Mobilygen Corporation
2900 Lakeside Drive #100
Santa Clara, CA 95054
Tel: (408) 869-4000
Fax: (408) 980-8044
email: info@mobilygen.com

MG1264

User Manual

Document Version: 1.1

Low Power H.264 and AAC Codec

Mobilygen Corporation
2900 Lakeside Drive #100
Santa Clara, CA 95054

Telephone 1 (408) 869-4000
FAX 1 (408) 980 8044

www.mobilygen.com

About This Document

This manual provides a complete reference for the MG1264 Low Power H.264 and AAC Codec for Mobile Devices User Manual.

Audience

This document assumes that the reader has knowledge of:

- Mobile Video product architectures
- Video Standards

Conventions

The following conventions were used in this manual:

Notation	Example	Meaning and Use
Courier typface	`.ini` file	Code Listings, names of files, symbols, and directories, are shown in courier typeface.
Bold Courier typeface	**`install`**	In a command line, keywords are shown in bold, non-italic, Courier typeface. Enter them exactly as shown.
Italics	*Note:*	Notes about the subject are shown with a header in italics.
Bold Italics	***Important:***	Important information about the subject is shown with the header in bold Italics. This information should not be ignored.
Square Brackets	[version]	You may, but need not, select one item enclosed within brackets. Do not enter the brackets
Angle Brackets	<username>	You must provide the information enclosed within brackets. Do not enter the brackets
Bar	les | les.out	You may select one (but not more than one) item from a list separated by bars. Do not enter the bars.

When computer output listings are shown, an effort has been made not to break up the lines when at all possible. This is to improve the clarity of the printout; for this reason, some listings will be indented, and others will start at the left edge of the column.

Terms

H.264

This manual makes reference to the term H.264 and MPEG4 Part 10 Advanced Video Coding (AVC). The full name for the standard is ITU-T Rec. H.264 / ISO/IEC 11496-10, “Advanced Video Coding”, and information can be found on the standard at:

- http://www.iec.ch/

The H.264 standard was jointly developed by the Video Coding Experts Group (VCEG) of the International Telecommunications Union (ITU) and the MPEG committee of ISO/IEC. The two identical standards are ISO MPEG4 Part 10 of MPEG4, and ITU-T H.264, but it is commonly referred to as “Advanced Video Coding” or AVC.

AAC

AAC is the MPEG-4 Advanced Audio Coding standard. Information on AAC can be found at:

- http://www.aac-audio.com/

Table of Contents

List of Figures

List of Tables

Chapter 1. Overview

The MG1264 is a single-chip H.264 codec IC that enables mobile products to capture, play and share high quality digital video and audio. The MG1264 is a complete A/V codec solution including both a H.264 30 frame-per-second video codec, and a high fidelity two-channel AAC audio codec. Power consumption while encoding is 185 mW for the complete device including VGA 30fps video, 2-channel AAC audio, and all chip I/O functions.

Mobilygen has developed a unique chip architecture dedicated to low power video processing. The patented EVE (Enabling Video Everywhere) architecture was used to implement the MG1264 and includes the following key technologies:

- Dedicated hardware media processing engines that are active only when data is being processed
- A highly-optimized hardware multi-threaded embedded microcontroller with single cycle context switching that controls all media processing operations and allows for easy integration of customer differentiating features
- An advanced video pre-processor that greatly improves H.264 encoder efficiency and overall video quality
- An ultra-efficient video processing oriented memory controller with forward seeking transaction reordering capabilities that doubles memory efficiency allowing all functions to operate with a single 16-bit SDRAM
- Patented low-power H.264 video coding algorithms developed specifically to maximize video quality
- Easy to control through standard firmware APIs; no customer programming is required

The MG1264 is designed for use in video surveillance, Digital Video Recorders (DVRs), Personal Video Recorders (PVRs), Portable Media Players (PMPs), video IP streaming, still cameras, video cameras, peripheral products, and any other applications that require H.264 encoding and/or decoding capabilities with very low power consumption.

1.1 Architecture

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices is built of the following blocks as shown in Figure 1-1:

- MG1264 Codec Host Interface
- Video Input and Preprocessor (VPP)
- H.264 Video Codec
- Video Output Processor (VPU)
- AAC Audio CODEC

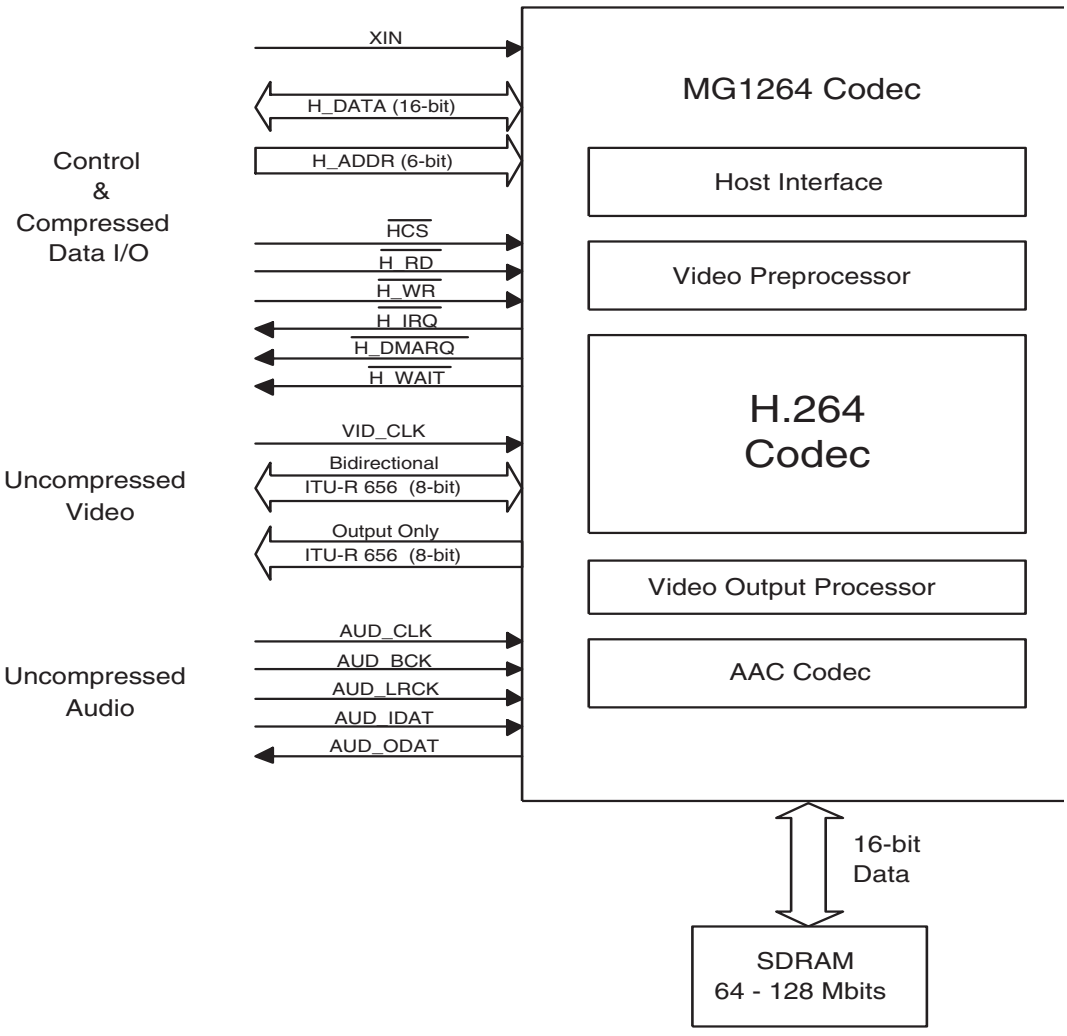


Figure 1-1 MG1264 Codec Block Diagram

1.2 MG1264 Codec Applications

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices is a VGA 30 fps H.264 and two-channel AAC Audio CODEC that enables Audio and Video (A/V) capture and playback functionality in mobile video products.

These include:

- Security cameras
- Digital Video Recorders (DVRs)
- Personal Video Recorders (PVRs)
- Video IP Streaming
- Digital Still Cameras
- Solid-State Camcorders
- Portable Media Players

The MG1264 Codec produces H.264 and AAC compliant bitstreams that can be decoded by any standard-compliant decoder such as software decoders on a PC.

The MG1264 Codec is designed for low power operation. Mobile video products based on the MG1264 Codec can play back any A/V content that it captures, just like a traditional tape based camcorder. The MG1264 Codec can also play back H.264 streams using the Tools shown in Figure 1-2. Figure 1-2 shows the MG1264 Codec's capabilities.

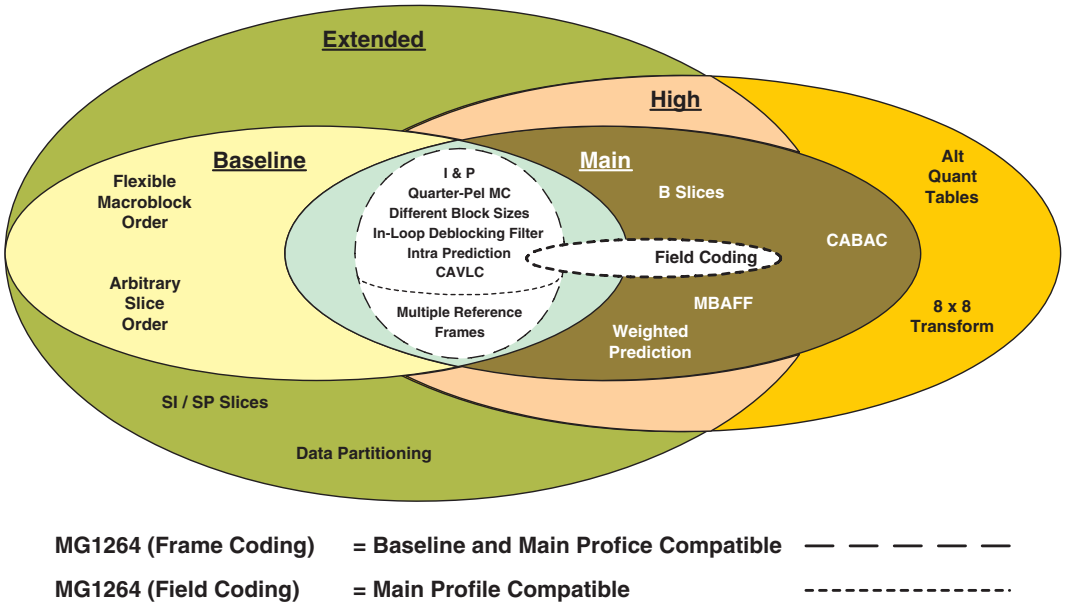


Figure 1-2 H.264/AVC Tools/Profiles

The MG1264 Codec is designed to be a coprocessor to a main System Host Processor and ASIC. Figure 1-3 is a camera system block diagram that shows how MG1264 Codec is integrated into a system. The main camera ASIC performs the traditional camera functions such as interface to the CCD, color processing, zoom lens control, LCD display, storage, etc.

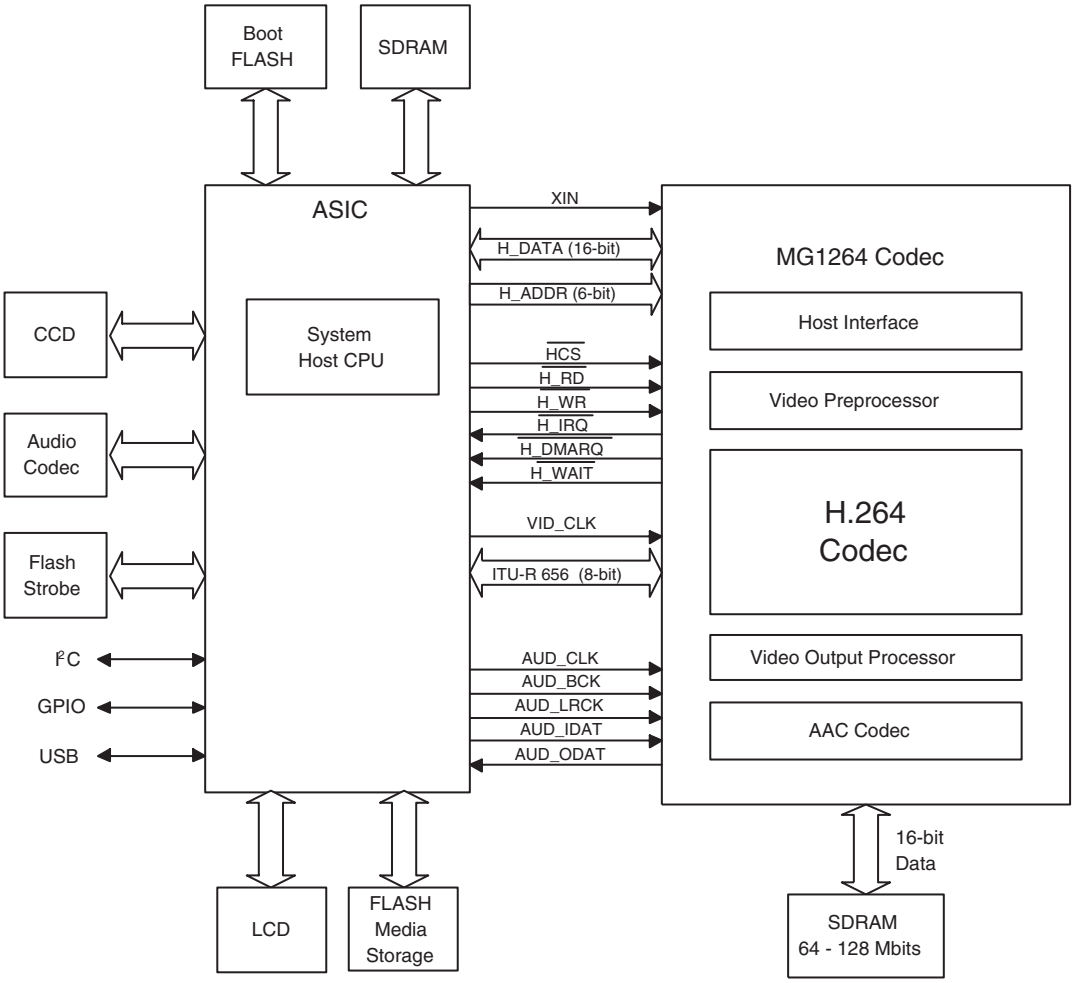


Figure 1-3 Camera System-Level Block Diagram

1.3 Features

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices has these features:

1.3.1 Modes Of Operation

Video compression applications require the user to manually select the mode of operation, typically video capture and playback. Depending upon the design, the MG1264 Codec does not need to be powered-on and initialized until the appropriate mode is selected.

1.3.2 Power-Up and Initialization

The MG1264 Codec is able to power-up and be ready to start encoding or decoding in less than one second. The System Host CPU is responsible for downloading the boot code to the MG1264 Codec and then initializing the MG1264 Codec. See “Firmware Loader” on page 119.

When the MG1264 Codec is actually powered-on and initialized is a design parameter of the system. It can be either when the system is turned on or when the Video Encode mode is selected.

1.3.3 Encode and Decode Mode

When the MG1264 Codec is active, it is ready to start encoding or decoding within one frame time.

1.3.4 MG1264 Codec Specifications

The MG1264 Codec implements a subset of H.264 Tools that achieves superior video quality with a low power budget. The MG1264 Codec does not implement the following H.264 tools: B-frames, CABAC, MAFF, Weighted Prediction, ASO, and FMO.

The MG1264 Codec can be best classified in the following way: If Frame mode coding is used, then the MG1264 Codec produces Baseline and Main Profile compatible streams (see Figure 1-2 on page 17). Baseline is the primary encoding mode for the MG1264 Codec, however the MG1264 Codec also supports Field mode coding. Streams coded as Field mode are technically Main Profile.

The MG1264 Codec decodes only streams created with the same subset of tools as listed above.

1.3.5 H.264 Encoder Target Performance

The MG1264 Codec is capable of encoding up to full D1 resolution (720 x 576). The MG1264 Codec is also capable of resolution down-sampling with excellent results at lower bitrates.

Table 1-1 lists target bitrates and corresponding resolutions for NTSC.

Table 1-1 Target H.264 Video Bitrates and Resolutions for NTSC

Video Bitrate (kbps)	Horizontal Resolution (Pixels)	Vertical Resolution (Pixels)	fps[1]	Notes Regarding The Source Video
300 - 768	320	240	30	QVGA, progressive, square pixel
1000 - 3000	640	480	30	VGA, progressive, square pixel
3000	800	600	25	SVGA, progressive, square pixel
300 - 768	352	240	30	SIF, progressive, rectangular pixel
1000 - 3000	720	480	30	D1, interlace, rectangular pixel

1. 30 fps is a shorthand representation for the traditional 29.976 NTSC frame rate. In applications where display on a traditional TV is required, the frame rate should be set accordingly.

1.3.6 PAL Resolution H.264

The MG1264 Codec is also capable of PAL encoding, as shown in Table 1-2.

Table 1-2 H.264 Video Bitrates and Resolutions for PAL

Video Bitrate (kbps)	Horizontal Resolution (pixels)	Vertical Resolution (Pixels)	fps	Notes Regarding The Source Video
300 - 768	352	288	25	QSIF, progressive, rectangular pixel
1000 - 3000	720	576	25	D1, interlace, rectangular pixel

1.3.7 SVGA 800x600 Video Resolution

The MG1264 Codec supports a maximum video resolution of 800x600 (SVGA). This resolution is intended for playback on PCs. This SVGA mode is intended to work with a standard 27 MHz video clock. The maximum frame rate is 25 fps.

1.3.8 Video Input and Output Scaling

The MG1264 Codec is capable of performing video scaling both on the input during encoding and on the output during decoding. This allows the MG1264 Codec to use alternate video resolutions to facilitate display on standard televisions. It also facilitates applications that make use of lower resolutions such as streaming over low bandwidth networks.

Input Video Scaling

The Input Video Scaler is designed to take a standard D1 resolution video input and generate the target encoding resolutions listed in Table 1-1. The MG1264 Codec supports a maximum horizontal resolution of 800 pixels.

The minimum picture size that can be encoded is 96 x 96. The resolution can be obtained by either setting the capture rectangle to that resolution, or by scaling a larger capture rectangle to that resolution. See the crop and scaling commands for more information.

However, note that you must use one slice per macroblock row for any horizontal resolution below 128, meaning that pictures that are 112 or 96 pixels wide must use one slice per row. See “Cropping“ and “Scaling” on page 187 for more information.

Output Video Scaling

The Output Video Scaler is designed to up-sample any resolution less than D1 for display on a standard television or down-sample for display on alternative displays. The Output Video Scaler also has the ability to perform square pixel to rectangular pixel conversion to support display of square pixel video correctly on a traditional TV display.

1.3.9 MG1264 Codec SDRAM Requirements by Function

Table 1-3 shows the SDRAM requirements for the most common applications.

Table 1-3 SDRAM Requirements by Function

Memory Requirements	Function
8 MBytes	Half Duplex (encode or decode) NTSC fully featured with no On-Screen Display (OSD)
16 MBytes	Half Duplex (encode or decode) PAL fully featured with OSD
	Full Duplex (encode and decode) NTSC with full-screen OSD
	Full Duplex (encode and decode) PAL with no OSD
32 MBytes	Full Duplex (encode and decode) PAL or NTSC with OSD

1.3.10 User Control of H.264 Encoder Features (Tools)

The encoder features are selectable. Each feature has settings and/or ranges that affect the overall compression efficiency accordingly. This section shows the key features and their associated target settings.

Picture Resolution

Table 1-1 shows the video resolutions. This selection uses the Input Video Scaler to produce the desired resolution.

Video Frame Rate

The primary target for the MG1264 Codec is natural motion frame rate like that of NTSC video at 30 fps. The following alternate frame rates are also supported:

- 25 fps (for PAL applications)
- 15 fps
- Any arbitrary bitrate between 1 and 30 fps

Video Bitrate

The target bitrates are listed in Table 1-1 for given resolutions. The maximum video data rate is 10 Mbps. The minimum video data rate is 56 kbps. The bitrate can be specified arbitrarily from 56 kbps to 10 Mbps.

Picture Type

The Picture Type refers to as Frame or Field coding. When Field mode is selected, all fields are encoded separately. The MG1264 Codec does not implement MBAFF mode.

GOP Structure

The MG1264 Codec uses I-frames and P-frames only. No B-frames. The GOP structure is user selectable from 1 to infinity. The default GOP length is 15.

On-the-Fly Parameter Changes

The following parameters can be changes at any time:

- Frame Rate
- Bit Rate
- Resolution
- GOP Length

1.3.11 The AAC Audio CODEC

The MG1264 Codec can encode two-channel AAC audio with 16-bit samples.

User Control of the AAC Encoder Features

The audio encoder features are selectable. Each feature has settings and/or ranges that affect the overall compression efficiency, accordingly. Table 1-4 shows the key features and their associated target settings.

Table 1-4 AAC Encoder Features

Feature	Options
Channels	Mono (1) or Stereo (2)
Sample rate	22.05, 24, 32, 44.1, or 48 kHz
Bitrate	8 - 384 kbps

1.3.12 I/O Control

The MG1264 Codec is intended to be a co-processor in a system with a basic architecture as shown in Figure 1-3. All system control is done by the System Host CPU, including booting and initializing the MG1264 Codec. All other I/O functions are controlled by the system host processor. I/O functions include: LCD control, camera sensor control, TV output, mass storage controllers, USB, Ethernet, audio codec, etc.

1.3.13 Full Duplex

The MG1264 can operate in Full Duplex mode, where it is encoding and decoding at the same time. Some limitations apply:

- VGA resolution (max)
- Frame coding only (no field coding).
- MPEG-1 Layer II audio, mono (no AAC)
- Requires 128 Mbits of SDRAM

Chapter 2. Pinlist and Packaging Information

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices is available in two RoHS compliant, **Pb-free** packages. The MG1264-169TFBGA is in a 169-pin Thin & Fine-Pitch Ball Grid Array package (TFBGA) that is 13mm x 13mm, with 0.8mm ball-pitch. The MG1264-156VFBGA is in a 156-pin Very Fine-Pitch Ball Grid Array package (VFBGA) that is 9mm x 9mm, with 0.5mm ball-pitch.

This chapter describes the mechanical specifications of the MG1264 Codec packages and provides a list of the pins for the device in each package. It also presents the solder profiles to be used for each of the packages, and the storage recommendations for the same package.

It is divided into these subsections:

2.1 Package Pinouts

2.1.1 169-Pin TFBGA Package

Figure 2-1 shows the pinout for the MG1264 Codec in the 169-Pin TFBGA package. This figure is continued on the next page.

	1	2	3	4	5	6	7	8
A	H_ADDR1	VIDOUT_DATA_1	VIDOUT_DATA_2	VIDOUT_DATA_3	VIDOUT_DATA_6	VIDOUT_VSYNC	VID_CLK	VID_DATA_5
B	H_ADDR2	$\overline{\text{HCS}}$	VIDOUT_DATA_0	VIDOUT_DATA_4	VIDOUT_DATA_5	VIDOUT_FIELD	VIDOUT_HSYNC	VID_DATA_6
C	H_ADDR4	H_ADDR3	IOVDD	CVDD	IOVDD	VIDOUT_DATA_7	IOVDD	VID_DATA_7
D	H_ADDR6	H_ADDR5	CVDD					
E	$\overline{\text{H\_WR}}$	IOVDD	IOVDD					
F	$\overline{\text{H\_WAIT}}$	$\overline{\text{H\_IRQ}}$	$\overline{\text{H\_RD}}$			GND	GND	GND
G	H_DATA0	$\overline{\text{H\_DMARQ}}$	IOVDD			GND	GND	GND
H	H_DATA3	H_DATA2	H_DATA1			GND	GND	GND
J	H_DATA4	H_DATA5	IOVDD			GND	GND	GND
K	H_DATA6	H_DATA7	H_DATA8			GND	GND	GND
L	H_DATA9	H_DATA10	H_DATA11					
M	H_DATA12	H_DATA13	CVDD					
N	H_DATA14	H_DATA15	TMS	CVDD	CVDD	MIOVDD	MIOVDD	MIOVDD
P	$\overline{\text{RESET}}$	SOUT	TDI	$\overline{\text{TRST}}$	AUD_IDAT	AUD_ODAT	AUD_BCK	SD_A_2
R	SIN	TCK	TDO	TMODE	AUD_CLK	AUD_LRCK	SD_A_10	SD_A_3

Figure 2-1 Pinout Diagram for the MG1264 Codec in the 169-pin TFBGA Package

9	10	11	12	13	14	15	
VID_DATA_4	VID_DATA_2	VID_FIELD	VID_HSYNC	IOVDD	AVDD	SD_CLK	**A**
VID_DATA_3	VID_DATA_1	VID_VSYNC	XIN	PFILTER	SD_DQ_15	SD_DQ_1	**B**
IOVDD	VID_DATA_0	IOVDD	CVDD	SD_DQ_0	SD_DQ_13	SD_DQ_2	**C**
				MIOVDD	MIOVDD	SD_DQ_14	**D**
				CVDD	SD_DQ_4	SD_DQ_3	**E**
GND	GND			MIOVDD	SD_DQ_12	SD_DQ_11	**F**
GND	GND			MIOVDD	SD_DQ_6	SD_DQ_10	**G**
GND	GND			MIOVDD	SD_DQ_9	SD_DQ_5	**H**
GND	GND			MIOVDD	SD_DQ_8	SD_CKE	**J**
GND	GND			CVDD	SD_DQM_1	SD_DQM_0	**K**
				MIOVDD	SD_A_11	SD_A_12	**L**
				MIOVDD	MIOVDD	SD_A_9	**M**
MIOVDD	MIOVDD	MIOVDD	MIOVDD	MIOVDD	SD_A_7	SD_A_8	**N**
SD_A_0	SD_BA_1	$\overline{\text{SD\_WE}}$	MIOVDD	$\overline{\text{SD\_CAS}}$	SD_A_6	SD_A_5	**P**
SD_A_1	$\overline{\text{SD\_CS}}$	SD_BA_0	$\overline{\text{SD\_RAS}}$	CVDD	SD_A_4	SD_DQ_7	**R**

Figure 2-1 Pinout Diagram for the MG1264 Codec in the 169-pin TFBGA Package (Continued)

2.1.2 156-Pin VFBGA Package

Figure 2-1 shows the pinout for the MG1264 Codec in the 156-pin VFBGA package. This figure is continued on the next page.

	1	2	3	4	5	6	7	8
A	H_ADDR1	VIDOUT_ DATA_0	VIDOUT_ DATA_2	VIDOUT_ DATA_4	VIDOUT_ DATA_6	VIDOUT_ FIELD	VIDOUT_ HSYNC	VID_DATA7
B	H_ADDR2	$\overline{\text{HCS}}$	VIDOUT_ DATA_1	VIDOUT_ DATA_3	VIDOUT_ DATA_5	VIDOUT_ DATA_7	VIDOUT_ VSYNC	VID_CLK
C	H_ADDR4	H_ADDR3	—	—	—	—	—	—
D	H_ADDR6	H_ADDR5	—	—	CVDD	—	—	—
E	IOVDD	$\overline{\text{H\_WR}}$	—	CVDD	—	—	—	—
F	$\overline{\text{H\_IRQ}}$	$\overline{\text{H\_RD}}$	—	—	—	—	IOVDD	IOVDD
G	$\overline{\text{H\_DMARQ}}$	$\overline{\text{H\_WAIT}}$	—	—	—	IOVDD	GND	GND
H	H_DATA1	H_DATA0	—	—	—	IOVDD	GND	GND
J	H_DATA2	H_DATA3	—	—	—	GND	GND	GND
K	H_DATA4	H_DATA5	—	—	—	GND	GND	GND
L	H_DATA6	H_DATA7	—	—	—	GND	MIOVDD	MIOVDD
M	H_DATA8	H_DATA9	—	CVDD	—	—	—	—
N	H_DATA10	H_DATA11	—	—	CVDD	CVDD	—	—
P	H_DATA12	H_DATA13	—	—	—	—	—	—
R	H_DATA14	$\overline{\text{RESET}}$	TMS	TDI	TDO	TMODE	AUD_CLK	AUD_LRCK
T	H_DATA15	SIN	SOUT	TCK	$\overline{\text{TRST}}$	AUD_IDAT	AUD_ODAT	AUD_BCK

Figure 2-2 Pinout Diagram for the MG1264 Codec in the 156-pin VFBGA Package

9	10	11	12	13	14	15	16	
VID_DATA6	VID_DATA4	VID_DATA2	VID_DATA0	VID_VSYNC	XIN	AVDD	PFILTER	A
VID_DATA5	VID_DATA3	VID_DATA1	VID_FIELD	VID_HSYNC	IOVDD	SD_CLK	SD_DQ_0	B
—	—	—	—	—	—	SD_DQ_1	SD_DQ15	C
—	—	—	CVDD	—	—	MIOVDD	SD_DQ_13	D
—	—	—	—	CVDD	—	SD_DQ_2	SD_DQ_14	E
GND	GND	GND	—	—	—	SD_DQ_4	SD_DQ_3	F
GND	GND	MIOVDD	—	—	—	SD_DQ_12	SD_DQ_11	G
GND	GND	MIOVDD	—	—	—	SD_DQ_6	SD_DQ_10	H
GND	GND	MIOVDD	—	—	—	SD_DQ_9	SD_DQ_5	J
GND	GND	MIOVDD	—	—	—	SD_DQ_8	SD_CKE	K
MIOVDD	MIOVDD	MIOVDD	—	—	—	SD_DQM_1	SD_DQM_0	L
—	—	—	—	CVDD	—	SD_A_11	SD_A_12	M
—	—	—	CVDD	—	—	MIOVDD	SD_A_9	N
—	—	—	—	—	—	SD_A_8	SD_A_7	P
SD_A_2	SD_A_1	$\overline{\text{SD\_CS}}$	SD_BA0	MIOVDD	$\overline{\text{SD\_CAS}}$	SD_A_6	SD_A_5	R
SD_A_10	SD_A_3	SD_A_0	SD_BA1	$\overline{\text{SD\_WE}}$	$\overline{\text{SD\_RAS}}$	SD_A_4	SD_DQ_7	T

Figure 2-2 Pinout Diagram for the MG1264 Codec in the 156-pin VFBGA Package (Continued)

2.2 Pin List

Table 2-1 shows the pin list sorted by interface. Table 2-2 shows the power and ground pins.

Table 2-1 MG1264 CODEC Host Interface Pins

Pin Name	Pin Number		Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
	169-pin TFBGA	156-pin VFBGA					
Clock Input							
XIN	B12	A14	I	—	3.3		Clock input: Clock Input to the internal PLL that is used to generate Core Clock. Supports 24 - 40 MHz. See “XIN Core Clock Considerations” on page 37 for more information.
Reset							
$\overline{\text{RESET}}$	P1	R2	I	—	3.3		Active low Reset pin.
Host Interface							
$\overline{\text{HCS}}$	B2	B2	I	—	3.3		Active low chip select. This pin is used to access the MG1264 internal registers, external memory and bit-stream read and write FIFO.
H_ADDR1	A1	A1	I	—	3.3		H-ADDR{6:1] - 6 bits of Host Bus Address
H_ADDR2	B1	B1	I	—	3.3		
H_ADDR3	C2	C2	I	—	3.3		
H_ADDR4	C1	C1	I	—	3.3		
H_ADDR5	D2	D2	I	—	3.3		
H_ADDR6	D1	D1	I	—	3.3		
$\overline{\text{H\_WR}}$	E1	E2	I	—	3.3		Active low, Write Enable
$\overline{\text{H\_RD}}$	F3	F2	I	—	3.3		Active low, Read Enable
$\overline{\text{H\_IRQ}}$	F2	F1	O	—	3.3	4	Active low, Host Interrupt Request
$\overline{\text{H\_WAIT}}$	F1	G2	O	—	3.3	4	Active Low wait signal. The MG1264 CODEC asserts this pin to extend the bus cycle until ti is able to accept data (during writes) or present data (during reads).
$\overline{\text{H\_DMARQ}}$	G2	G1	O	—	3.3	4	Active low, bitstream DMA Request. See “MG1264 Codec External Memory Interface Port 2 Registers” on page 70 and “MG1264 Codec Bitstream Interface Registers” on page 70 for more information.

Table 2-1 MG1264 CODEC Host Interface Pins

Pin Name	Pin Number		Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
	169-pin TFBGA	156-pin VFBGA					
H_DATA0	G1	H2	IO	—	3.3	4	H_DATA[15:0] - 16 bits Host Data Bus
H_DATA1	H3	H1	IO	—	3.3	4	
H_DATA2	H2	J1	IO	—	3.3	4	
H_DATA3	H1	J2	IO	—	3.3	4	
H_DATA4	J1	K1	IO	—	3.3	4	
H_DATA5	J2	K2	IO	—	3.3	4	
H_DATA6	K1	L1	IO	—	3.3	4	
H_DATA7	K2	L2	IO	—	3.3	4	
H_DATA8	K3	M1	IO	—	3.3	4	
H_DATA9	L1	M2	IO	—	3.3	4	
H_DATA10	L2	N1	IO	—	3.3	4	
H_DATA11	L3	N2	IO	—	3.3	4	
H_DATA12	M1	P1	IO	—	3.3	4	
H_DATA13	M2	P2	IO	—	3.3	4	
H_DATA14	N1	R1	IO	—	3.3	4	
H_DATA15	N2	T1	IO	—	3.3	4	

1. I = Input, IU = Input w/ Internal Pull-Up, IS = Input w/ Schmitt Trigger, IO = Bidirectional, O = Output, OT = Output w/ Tri-state

Table 2-1 MG1264 CODEC SDRAM Interface Pins

Pin Name	Pin Number 169-pin TFBGA	Pin Number 156-pin VFBGA	Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
SDRAM Interface							
SD_A_12	L15	M16	O	—	2.5 or 3.3	4	SDRAM address - SD_A[12:0]
SD_A_11	L14	M15	O	—	2.5 or 3.3	4	
SD_A_10	R7	T9	O	—	2.5 or 3.3	4	
SD_A_9	M15	N16	O	—	2.5 or 3.3	4	
SD_A_8	N15	P15	O	—	2.5 or 3.3	4	
SD_A_7	N14	P16	O	—	2.5 or 3.3	4	
SD_A_6	P14	R15	O	—	2.5 or 3.3	4	
SD_A_5	P15	R16	O	—	2.5 or 3.3	4	
SD_A_4	R14	T15	O	—	2.5 or 3.3	4	
SD_A_3	R8	T10	O	—	2.5 or 3.3	4	
SD_A_2	P8	R9	O	—	2.5 or 3.3	4	
SD_A_1	R9	R10	O	—	2.5 or 3.3	4	
SD_A_0	P9	T11	O	—	2.5 or 3.3	4	
SD_DQM_1	K14	L15	O	—	2.5 or 3.3	4	SDRAM mask bits - SD_DQM[1:0]
SD_DQM_0	K15	L16	O	—	2.5 or 3.3	4	
SD_BA_1	P10	T12	O	—	2.5 or 3.3	4	SDRAM bank select - SD_BA[1:0]
SD_BA_0	R11	R12	O	—	2.5 or 3.3	4	
$\overline{\text{SD\_WE}}$	P11	T13	O	—	2.5 or 3.3	4	Active low SDRAM write enable
$\overline{\text{SD\_CAS}}$	P13	R14	O	—	2.5 or 3.3	4	Active low SDRAM CAS
$\overline{\text{SD\_RAS}}$	R12	T14	O	—	2.5 or 3.3	4	Active low SDRAM RAS
$\overline{\text{SD\_CS}}$	R10	R11	O	—	2.5 or 3.3	4	Active low SDRAM chip select
SD_CKE	J15	K16	O	—	2.5 or 3.3	4	SDRAM clock enable
SD_DQ_15	B14	C16	IO	—	2.5 or 3.3	4	Bidirectional SDRAM data pins SD_DQ{[15:0]
SD_DQ_14	D15	E16	IO	—	2.5 or 3.3	4	
SD_DQ_13	C14	D16	IO	—	2.5 or 3.3	4	
SD_DQ_12	F14	G15	IO	—	2.5 or 3.3	4	
SD_DQ_11	F15	G16	IO	—	2.5 or 3.3	4	
SD_DQ_10	G15	H16	IO	—	2.5 or 3.3	4	
SD_DQ_9	H14	J15	IO	—	2.5 or 3.3	4	
SD_DQ_8	J14	K15	IO	—	2.5 or 3.3	4	
SD_DQ_7	R15	T16	IO	—	2.5 or 3.3	4	
SD_DQ_6	G14	H15	IO	—	2.5 or 3.3	4	
SD_DQ_5	H15	J16	IO	—	2.5 or 3.3	4	
SD_DQ_4	E14	F15	IO	—	2.5 or 3.3	4	
SD_DQ_3	E15	F16	IO	—	2.5 or 3.3	4	
SD_DQ_2	C15	E15	IO	—	2.5 or 3.3	4	
SD_DQ_1	B15	C15	IO	—	2.5 or 3.3	4	
SD_DQ_0	C13	B16	IO	—	2.5 or 3.3	4	
SD_CLK	A15	B15	O	—	2.5 or 3.3	8	SDRAM clock. This pin provides the clock to the SDRAM

1. I = Input, IU = Input w/ Internal Pull-Up, IS = Input w/ Schmitt Trigger, IO = Bidirectional, O = Output, OT = Output w/ Tri-state

Table 2-1 MG1264 CODEC Video and Interface Pins

Pin Name	Pin Number		Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
	169-pin TFBGA	156-pin VFBGA					
Video Interface							
VID_HSYNC	A12	B13	I	Down	3.3		Option to use negative edge of VID_CLK
VID_VSYNC	B11	A13	I	Down	3.3		Option to use negative edge of VID_CLK
VID_FIELD	A11	B12	I	Down	3.3		Option to use negative edge of VID_CLK
VID_DATA_7	C8	A8	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_6	B8	A9	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_5	A8	B9	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_4	A9	A10	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_3	B9	B10	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_2	A10	A11	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_1	B10	B11	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_DATA_0	C10	A12	IO	—	3.3	4	Option to use negative edge of VID_CLK
VID_CLK	A7	B8	I	—	3.3		Video Clock: Used for both the VID_DATA and VIDOUT ports. Always input. See “VID_CLK Video Clock Considerations” on page 37 for more information.
VIDOUT_HSYNC	B7	A7	I	Down	3.3		Option to use negative edge of VID_CLK
VIDOUT_VSYNC	A6	B7	I	Down	3.3		Option to use negative edge of VID_CLK
VIDOUT_FIELD	B6	A6	I	Down	3.3		Option to use negative edge of VID_CLK
VIDOUT_DATA_7	C6	B6	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_6	A5	A5	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_5	B5	B5	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_4	B4	A4	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_3	A4	B4	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_2	A3	A3	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_1	A2	B3	O	—	3.3	4	Option to use negative edge of VID_CLK
VIDOUT_DATA_0	B3	A2	O	—	3.3	4	Option to use negative edge of VID_CLK
Audio Interface							
AUD_IDAT	P5	T6	I	Down	3.3		Audio serial input data
AUD_CLK	R5	R7	I	Down	3.3		Audio over sample clock 256*fs (LRCK)
AUD_ODAT	P6	T7	O	—	3.3	4	Audio serial output data
AUD_LRCK	R6	R8	IO	Down	3.3	4	Audio left/right clock (48, 44.1, 32, 24, 22.05 MHz) This pin should be software-configured as an output when unused.
AUD_BCK	P7	T8	IO	Down	3.3	4	Audio bit clock, 32 or 64 *fs (LRCK) This pin should be software-configured as an output when unused.

1. I = Input, IU = Input w/ Internal Pull-Up, IS = Input w/ Schmitt Trigger, IO = Bidirectional, O = Output, OT = Output w/ Tri-state

Table 2-1 MG1264 CODEC Test Pins

Pin Name	Pin Number		Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
	169-pin TFBGA	156-pin VFBGA					
Test Pins							
SIN	R1	T2	I	Down	3.3		UART receive data
SOUT	P2	T3	O	—	3.3	4	UART transmit data
TMS	N3	R3	IU	Down	3.3		JTAG test mode. This pin has an internal 20 kOhm - 150 kOhm (50 kOhm nominal) pull-up resistor.
TCK	R2	T4	IS	Down	3.3		JTAG test clock
TDI	P3	R4	IU	Down	3.3		JTAG test data input. This pin has an internal 20 kOhm - 150 kOhm (50 kOhm nominal) pull-up resistor.
TDO	R3	R5	OT	—	3.3	8	JTAG test data output
$\overline{\text{TRST}}$	P4	T5	IU	Down	3.3		Active low JTAG Reset. This pin has an internal 20 kOhm - 150 kOhm (50 kOhm nominal) pull-up resistor.
TMODE	R4	R6	I	Down	3.3		Manufacturer test mode

1. I = Input, IU = Input w/ Internal Pull-Up, IS = Input w/ Schmitt Trigger, IO = Bidirectional, O = Output, OT = Output w/ Tri-state

2.2.1 The SOUT and SIN Signals

The SOUT and SIN signals provide a UART monitor port that can be used for debug purposes. These are traditional asynchronous signals that can be used as a UART output and input respectively.

2.2.2 JTAG Signals

The TCK, TDI, TDO, TMS and $\overline{\text{TRST}}$ signals comprise a JTAG test port. Contact your Mobilygen Sales Representative for information regarding JTAG.

2.2.3 TMODE Signal

Setting the TMODE signal high puts the MG1264 Codec into factory test mode, and will cause erratic operation. Customers should always pull TMODE low.

Table 2-2 MG1264 CODEC Power and Ground Pin List

Pin Name	Pin Number 169-pin TFBGA	Pin Number 156-pin VFBGA	Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
Power And Ground							
CVDD	C4	D5			1.2		1.2V Core Power Supply
	C12	D12			1.2		1.2V Core Power Supply
	D3	E4			1.2		1.2V Core Power Supply
	E13	E13			1.2		1.2V Core Power Supply
	K13	M4			1.2		1.2V Core Power Supply
	M3	M13			1.2		1.2V Core Power Supply
	N4	N5			1.2		1.2V Core Power Supply
	N5	N6			1.2		1.2V Core Power Supply
	R13	N12			1.2		1.2V Core Power Supply
GND	F6	F9			GND		Ground
	F7	F10			GND		Ground
	F8	F11			GND		Ground
	F9	G7			GND		Ground
	F10	G8			GND		Ground
	G6	G9			GND		Ground
	G7	G10			GND		Ground
	G8	H7			GND		Ground
	G9	H8			GND		Ground
	G10	H9			GND		Ground
	H6	H10			GND		Ground
	H7	J6			GND		Ground
	H8	J7			GND		Ground
	H9	J8			GND		Ground
	H10	J9			GND		Ground
	J6	J10			GND		Ground
	J7	K6			GND		Ground
	J8	K7			GND		Ground
	J9	K8			GND		Ground
	J10	K9			GND		Ground
	K6	K10			GND		Ground
	K7	L6			GND		Ground
	K8	–			GND		Ground
	K9	–			GND		Ground
	K10	–			GND		Ground

Table 2-2 MG1264 CODEC Power and Ground Pin List

Pin Name	Pin Number		Input or Output[1]	Pullup or Pulldown when not in use	Voltage (V)	Function (drive Strength) (mA)	Description
	169-pin TFBGA	156-pin VFBGA					
IOVDD	C3	B14			3.3		3.3V IO Power Supply
	C5	E1			3.3		3.3V IO Power Supply
	C7	F7			3.3		3.3V IO Power Supply
	C9	F8			3.3		3.3V IO Power Supply
	C11	G6			3.3		3.3V IO Power Supply
	E3	H6			3.3		3.3V IO Power Supply
	A13	–			3.3		3.3V IO Power Supply
	E2	–			3.3		3.3V IO Power Supply
	G3	–			3.3		3.3V IO Power Supply
	J3	–			3.3		3.3V IO Power Supply
MIOVDD	D13	D15			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	F13	G11			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	G13	H11			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	H13	J11			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	J13	K11			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	L13	L7			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	M13	L8			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N6	L9			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N7	L10			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N8	L11			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N9	N15			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N10	R13			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N11	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N12	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	N13	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	D14	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
MIOVDD	M14	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
	P12	–			2.5 or 3.3		2.5V or 3.3V Memory IO Power Supply
AVDD	A14	A15			1.2		1.2V Analog VDD for PLL power. See Section 2.3.4, AVDD Power Supply Considerations
PFILTER	B13	A16			N/A		Analog PLL power supply filter. Do NOT ground this pin. See Section 2.3.4, AVDD Power Supply Considerations for more information

1. I = Input, IU = Input w/ Internal Pull-Up, IS = Input w/ Schmitt Trigger, IO = Bidirectional, O = Output, OT = Output w/ Tri-state

2.3 Design Considerations

The following should be taken into consideration when designing with the MG1264 Low Power H.264 and AAC Codec for Mobile Devices.

2.3.1 Ground Plane Considerations

"Pinout Diagram for the MG1264 Codec in the 169-pin TFBGA Package" on page 26 shows the location and identification of each Ground (GND) pin. All Ground pins should be tied together in a common plane.

2.3.2 XIN Core Clock Considerations

The XIN signal is input to an internal PLL that is used to general the internal Core Clock. The MG1264 Codec Core Clock can run up to 110 MHz maximum by programming the internal PLL accordingly. Generation of the Core Clock is subject to the restrictions described in "Phase Lock Loop Restrictions" on page 245.

See "Clock and Configuration Registers" on page 74 for more information regarding control of the PLL.

Note: XIN is independent of VID_CLK operation.

2.3.3 VID_CLK Video Clock Considerations

The VID_CLK signal drives both the VID_DATA and VIDOUT_DATA ports. A clock must always be provided to the VID_CLK signal. The MG1264 Codec does not generate VID_CLK in any mode. The MG1264 video ports, and VID_CLK signal, can operate up to 40 MHz. This is beyond the typical 27 MHz associated with traditional 656 style video ports. See Chapter 5 for more information related to the operation of the video ports.

Note: VID_CLK is independent of XIN operation, but is subject to the restrictions described in "Phase Lock Loop Restrictions" on page 245.

2.3.4 AVDD Power Supply Considerations

The AVDD signal requires a very low current of 1.3 mA maximum. PFILTER is the power supply pin for the Phase Lock Loop (PLL). This pin should **not** be grounded. The power supply filtering circuit shown in Figure 2-3 is recommended to minimize jitter on the PLL.

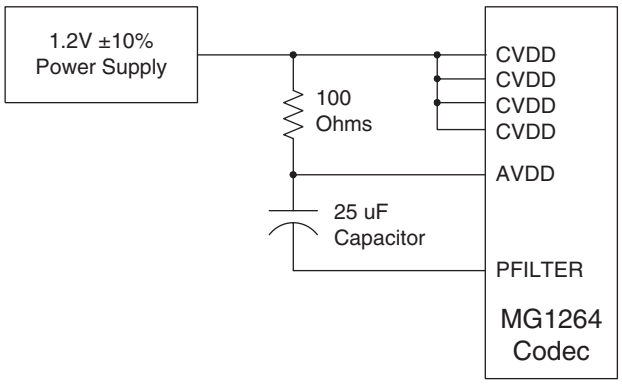


Figure 2-3 Switching Power Supply Decoupling

2.4 Package Dimensions

Figure 2-4 shows the package dimensions for the 169-pin RoHS compliant, **Pb-free**, 13mm x 13mm, 0.8mm ball-pitch TFBGA package. Figure 2-5 shows the package dimensions for the the 156-pin RoHS compliant, **Pb-free**, 9mm x 9mm, 0.5mm ball-pitch VFBGA package.

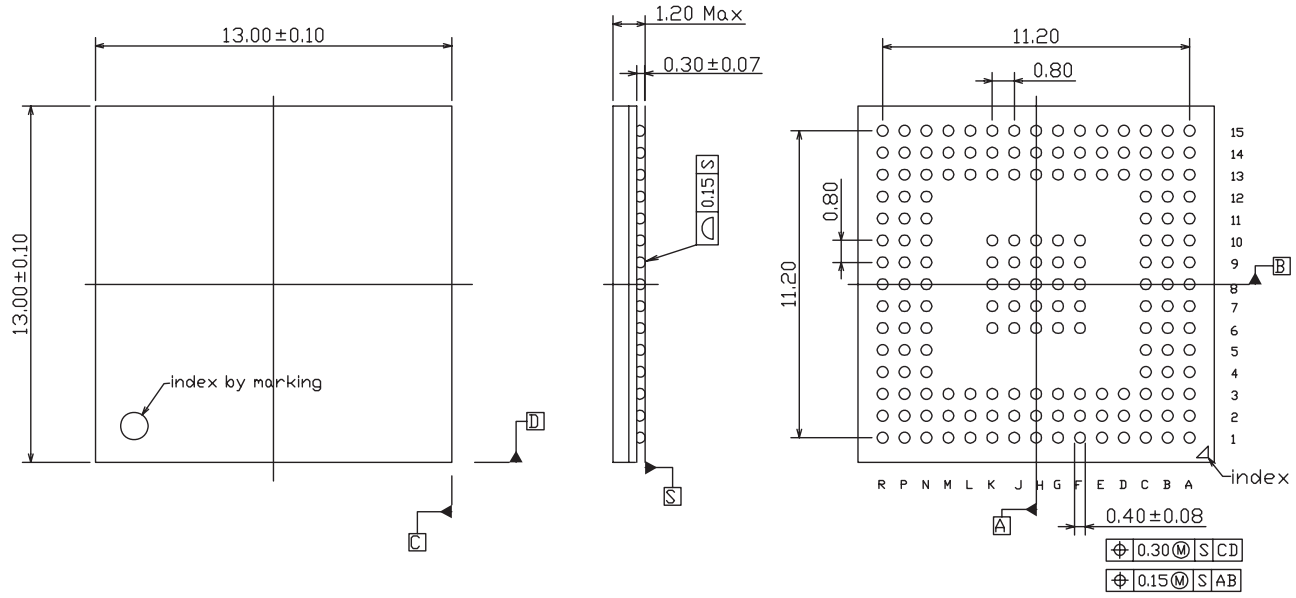


Figure 2-4 169-pin TFBGA Package Mechanical Dimensions

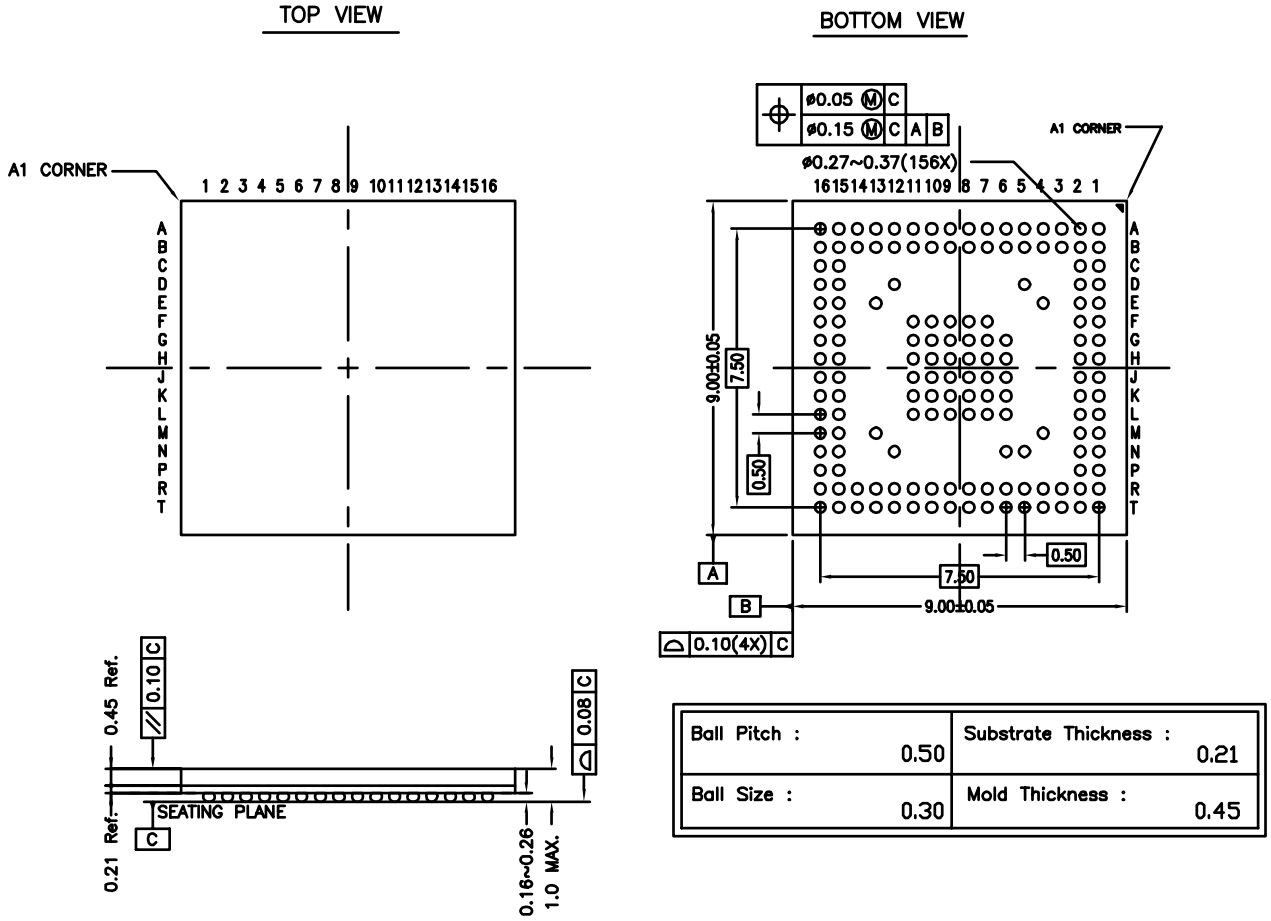


Figure 2-5 156-pin VFBGA Package Mechanical Dimensions

2.5 Ordering Information

Table 2-3 shows the part numbers to be used when ordering the MG1264 Low Power H.264 and AAC Codec for Mobile Devices.

Table 2-3 Ordering Information

Part Number	Description
MG1264-169TFBGA	MG1264-169TFBGA in a 169-pin Thin & Fine-Pitch Ball Grid Array package (TFBGA) that is 13mm x 13mm, with 0.8mm ball-pitch.
MG1264-156VFBGA	MG1264-156VFBGA in a 156-pin Very Fine-Pitch Ball Grid Array package (VFBGA) that is 9mm x 9mm, with 0.5mm ball-pitch

2.6 Solder Profile

Figure 2-6 shows the solder profile to be used when mounting the package. This specification applies to both the MG1264-169TFBGA and the MG1264-156VFBGA.

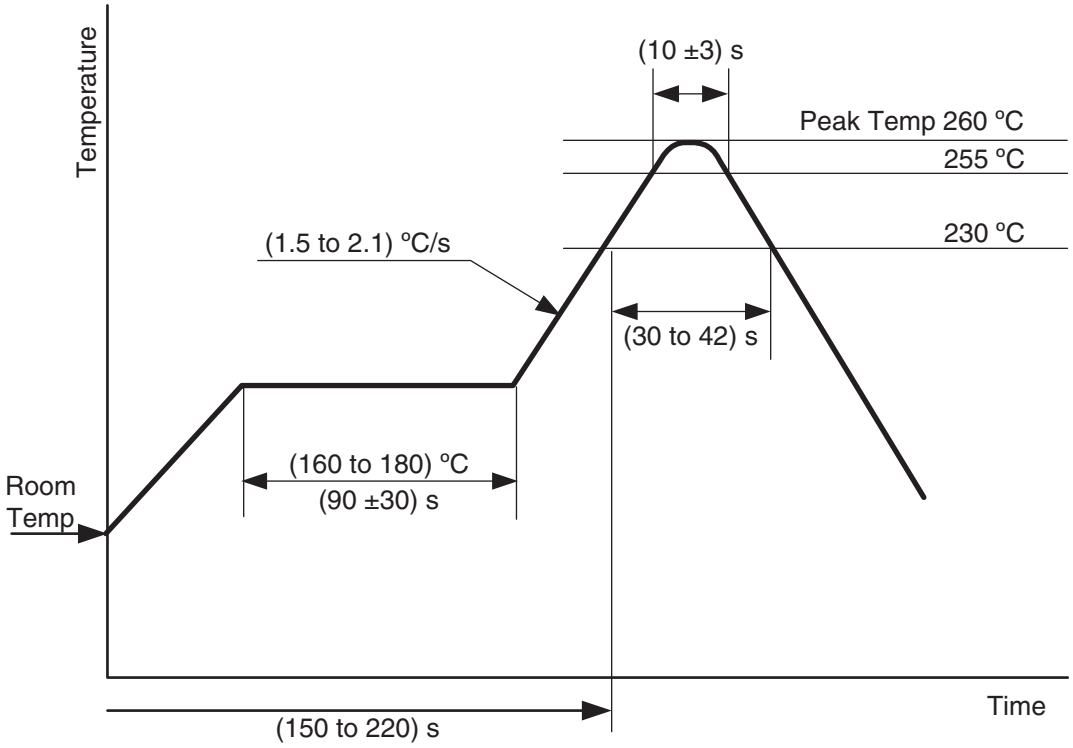


Figure 2-6 Temperature Profile (Body Temp) of Infrared Convection Reflow Soldering

Test Conditions

- Baked for 24 hours at 125º C
- Moisture soaking
  - Ta = (30 +/- 2)º C (Ta = Ambient Temperature)
  - RH = (70 +/- 5)% (RH = Relative Humidity)
  - 96 h
- Reflow Soldering: IRS
- Infra-red Reflow Soldering (IRS):
- Peak Temperature: 255º to 260º C for 10 (+/- 3) seconds
- Pre-heat: 70º (+/- 10º) for 90 (+/- 30) seconds

Reference Specifications: EIAJ ED-4701 A-133B

2.7 Storage Recommendations

1. Shelf life in sealed bag: 12 months at < 40º C and < 80% RH.

2: In the case of twice reflow process:

- Mounted within 96 hours for first reflow at factory conditions of below 30º C and below 70% RH, and
- Reflowed within 96 hours after first reflow at factory conditions of below 30º C and below 70% RH, or
- Stored at below 30% RH (SMD stocker).

3: In the case of one time reflow process:

- Mounted within 168 hours at factory conditions of below 30º C and below 60% RH (JEDEC Level3), or
- Stored at below 30% RH (SMD stocker).

4: Devices require baking before mounting if the moisture indicator inside the bag shows over 30% RH when the bag is opened or when (1) or (2) or (3) are not met.

5: If baking is required, the devices may be baked for 24 hours at 125º (+/- 5º) C.

Note: Stipulations about the handling of moisture-proof bags or moisture sensitive devices give priority to above cautions.

Chapter 3. Specifications

This chapter describes the electrical and mechanical specifications of the MG1264 Codec. It is divided into these subsections:

3.1 Electrical Characteristics

This section specifies the electrical characteristics of the MG1264 Codec.

3.1.1 Absolute Maximum Ratings

Table 3-1 gives the absolute maximum ratings. Exposure to stresses beyond those listed in this table may result in device unreliability, permanent damage, or both.

Table 3-1 Absolute Maximum Ratings

Parameter	Value	Units	Notes
CVDD	1.6	V	—
AVDD	1.6	V	—
IOVDD	4.5	V	—
MIOVDD	4.5	V	—
Maximum Input Voltage	IO_VDD + 0.3	V	Referenced to associated IOVDD
Storage Temperature Range	-40 to 150	°C	See “Storage Recommendations” on page 41.
Operating Temperature Range (case)	-20 to 125	°C	—

3.1.2 Operating Conditions

Table 3-2 specifies the operating conditions for the MG1264 Codec.

Table 3-2 Operating Conditions

Parameter	Minimum	Typical	Maximum	Units	Notes
CVDD	1.08	1.2	1.32	V	1.2V ±10%
VDDP	1.08	1.2	1.32	V	1.2V ±10%
IOVDD	2.97	3.3	3.63	V	3.3V ±10%
MIOVDD	2.25	2.5/3.3	3.63	V	2.5 / 3.3V ±10%
$T_{Ambient}$	-20		85	°C	

3.1.3 DC Characteristics

Table 3-3 defines the DC characteristics.

Table 3-3 DC Characteristics

Symbol	Parameters	Test Conditions	IOVDD and MIOVDD = 3.3V ±10%		MIOVDD = 2.5V ±10% [1]		Units
			Min	Max	Min	Max	
V_{IH}	Input High Level	V_{DD} = Maximum	2.0	—	1.7	—	V
V_{IL}	Input Low-Level Voltage	V_{DD} = Minimum	—	0.8	—	0.5	V
V_{OH}	Output High-Level Voltage	V_{DD} = Minimum, I_{OH} = –2, –4, –8 mA	2.4	—	1.9	—	V
V_{OL}	Output Low-Level Voltage	V_{DD} = Minimum, I_{OL} = –2, –4, –8 mA	—	0.4	—	0.3	V
I_{IH}	Input Leakage	V_{DD} = Maximum, V_{IN} = V_{DD}	–10	–10	–10	–10	µA
I_{IL}	Input Leakage	V_{DD} = Maximum, V_{IN} = 0V	–10	–10	–10	–10	µA
I_{OZ}	TriState Leakage	V_{DD} = Maximum, V_{IN} = 0V – IOVDD	–10	–10	–10	–10	µA
IDD_{Core}	Core Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	175	—	175	mA
IDD_{IO}	I/O Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	5	—	5	mA
IDD_{SD_IO}	SD I/O Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	20	—	20	mA
I_{PU}	Internal Pullup Current for pins of type IU	V_{DD} = Maximum, V_{IN} = 0V	–25	–165	–25	–165	µA
C_{PIN}	Capacitance[2]	—	—	5	—	5	pF

1.The MIOVDD = 2.5V columns only apply to the SDRAM interface when using 2.5V SDRAMs.
2.Not 100% tested.

3.1.4 Standby Power

Table 3-4 shows the standby power for each of the major elements when the MG1264 Codec is placed into powerdown mode and the PLL is stopped. The MG1264 Codec is placed into powerdown mode using the PLLPowerDown bit in the Clock Configuration Register as described in “Clock Configuration Register on page 74.

Table 3-4 Standby Power

Element	Min	Typ	Max	Units
Core	—	2	—	mW
DRAM	—	0.1	—	mW
I/O	—	1.9	—	mW
Total	—	4.0	—	mW

3.1.5 Power-Up and Power-Down Sequence

This section provides the recommended power-up and power-down sequences. In an ideal design, all of the power supplies become stable at the same time to prevent any direct feed-through current. In real designs, though, there is typically a time delay between when the various power supplies stabilize. This section explains the restrictions on the time differences between the power supplies.

Case 1: Power on: 1.2V Core Supply comes on First, 1.2V Core Supply goes off last
Refer to Figure 3-1, In this case, the restrictions are as follows:

$T_{LAG}1$, $T_{LAG}2 < 500$ ms.
T_{ON}, $T_{OFF} < 500$ ms.

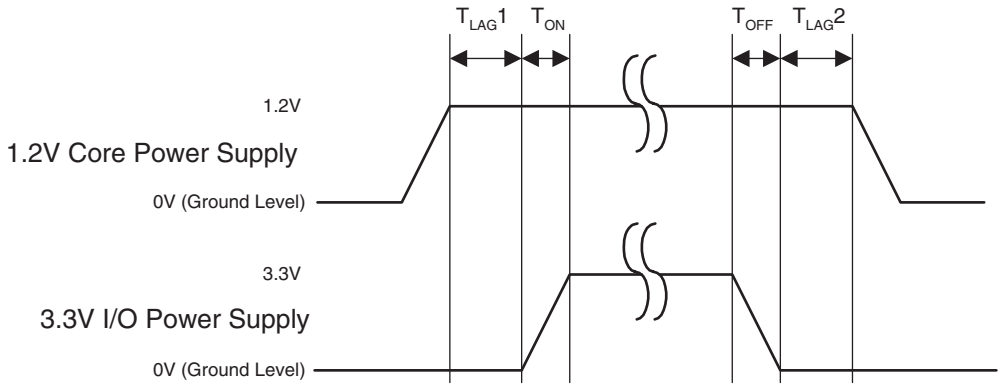


Figure 3-1 Power Supply Sequencing, Case 1

Case 2: Power on: 3.3V I/O Supply comes on First, 3.3V I/O Supply goes off last

Refer to Figure 3-2, In this case, the restrictions are as follows:

$T_{LAG}1$, $T_{LAG}2 < 500$ ms.

T_{ON}, $T_{OFF} < 500$ ms.

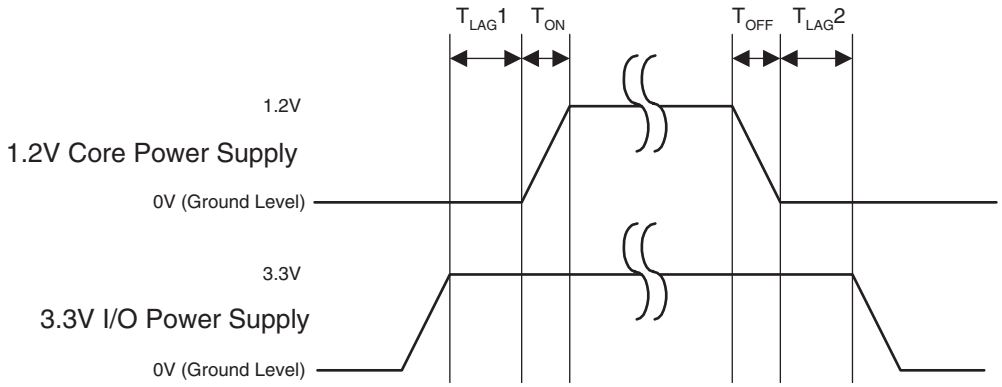


Figure 3-2 Power Supply Sequencing, Case 2

Other Cases

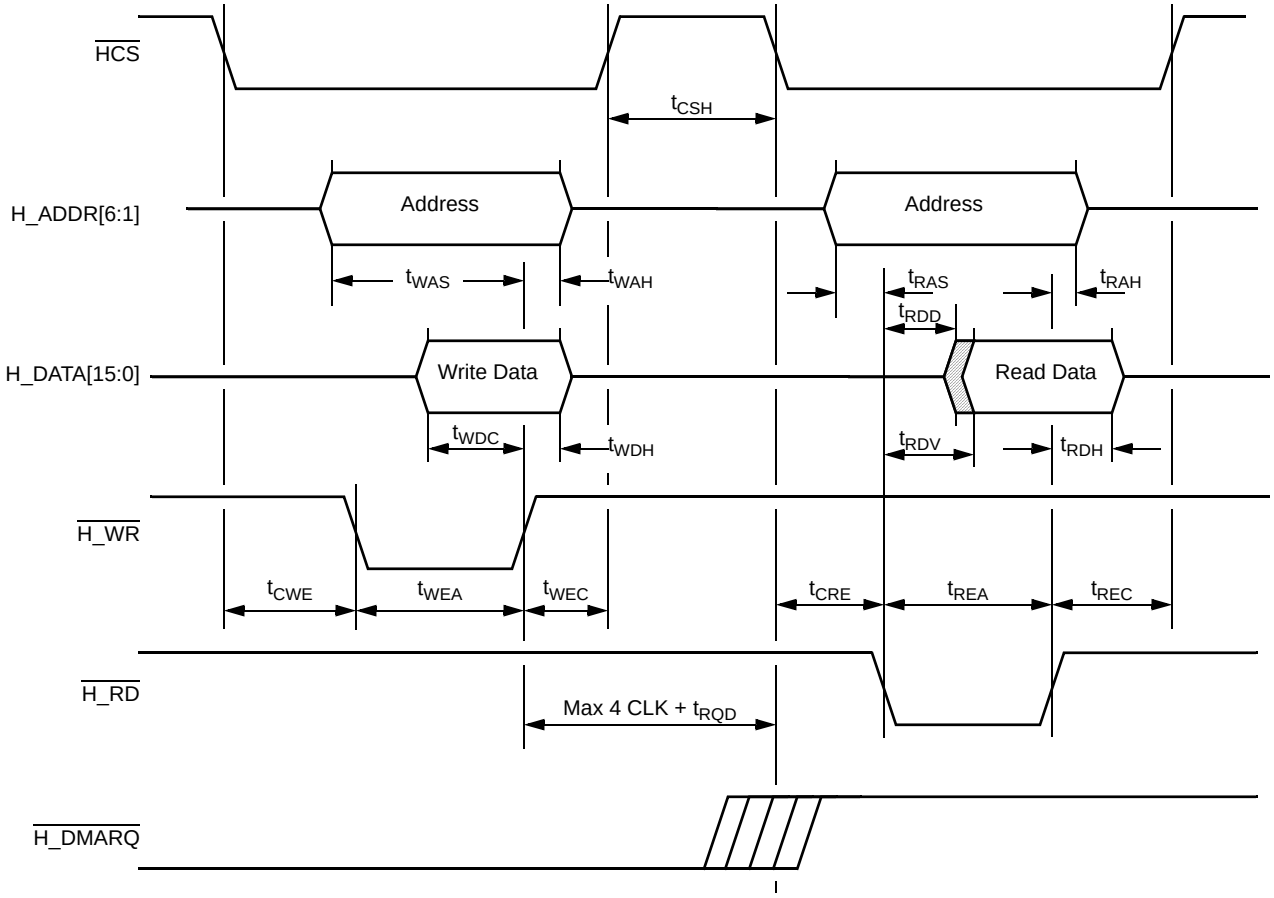
Follow the restrictions in Case 1 and Case 2. For example, if the 3.3V I/O supply powers up first, and then powers down first, you should follow Case 2 for power Up and Case 1 for power Down.

3.2 AC Timing

This section provides the AC timing for the MG1264 Codec's various interfaces. This section is divided into the following subsections:

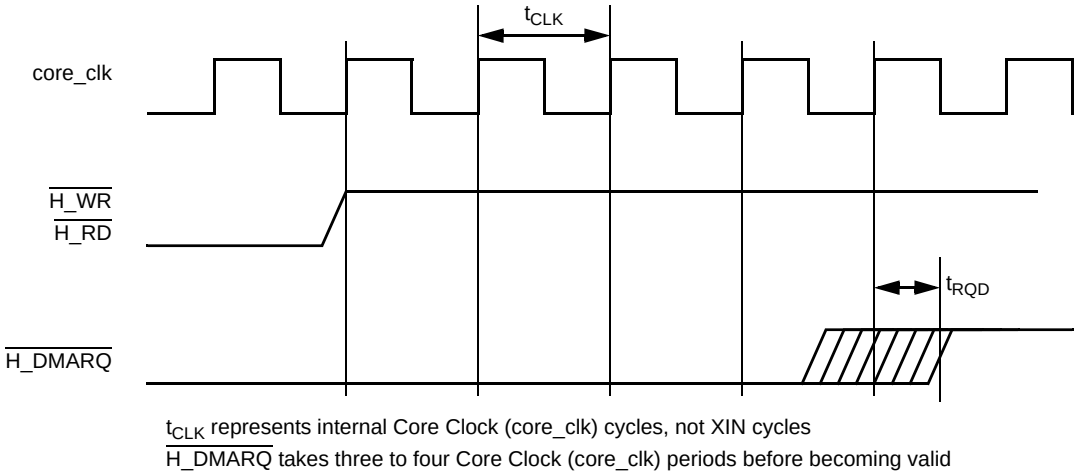
3.2.1 MG1264 Codec Host Interface Timing

Figure 3-3 shows the timing diagram for the MG1264 Codec Host Interface, Figure 3-4 shows the DMA Timing, Figure 3-5 shows the Wait timing, and Figure 3-6 shows the Interrupt Request timing. Table 3-5 lists the timing parameters for each of these diagrams.



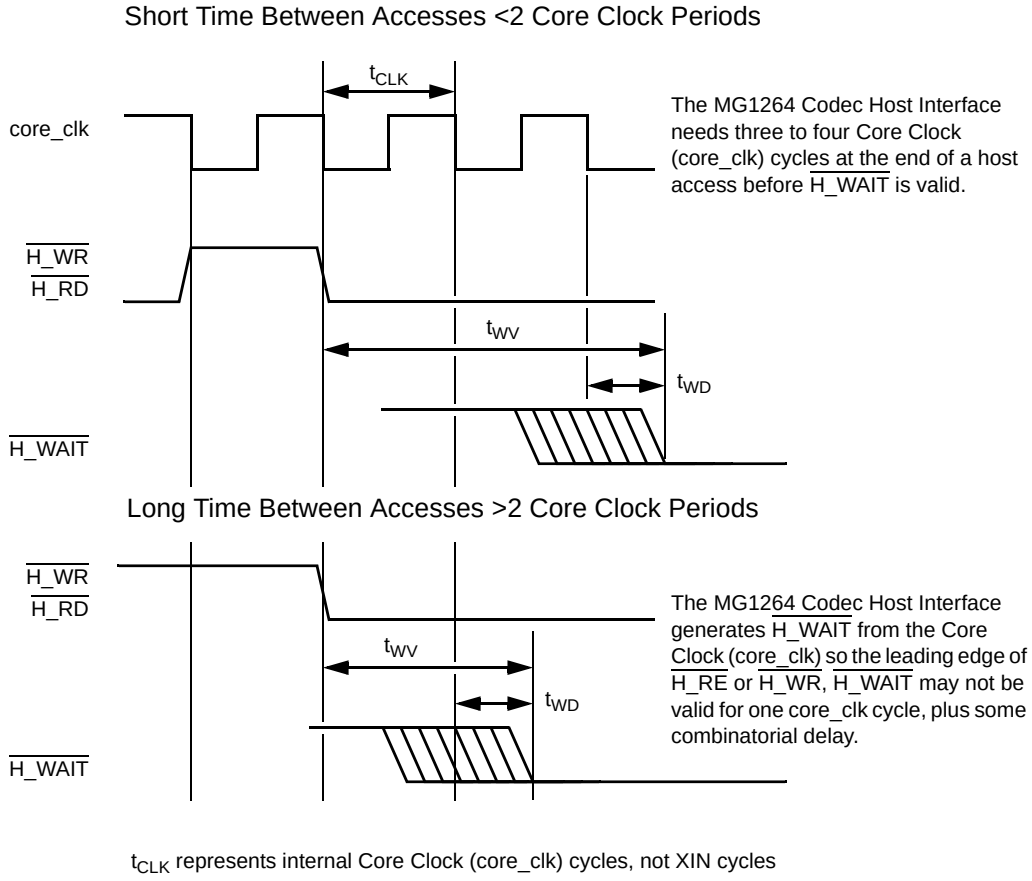
H_DMARQ takes three to four Core Clock (core_clk) periods before becoming valid

Figure 3-3 MG1264 Codec Host Interface AC Timing Waveform



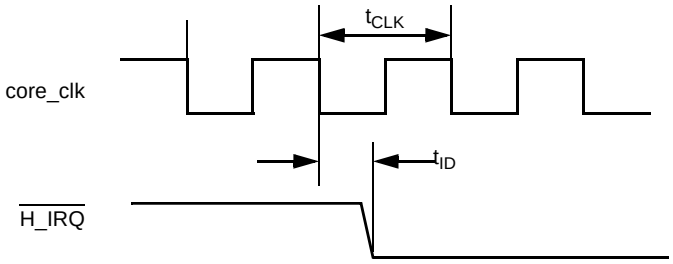
t_{CLK} represents internal Core Clock (core_clk) cycles, not XIN cycles
$\overline{H\_DMARQ}$ takes three to four Core Clock (core_clk) periods before becoming valid

Figure 3-4 MG1264 Codec $\overline{H\_DMARQ}$ Timing



t_{CLK} represents internal Core Clock (core_clk) cycles, not XIN cycles

Figure 3-5 $\overline{H\_WAIT}$ Timing



t_{CLK} represents internal Core Clock (core_clk) cycles, not XIN cycles

Figure 3-6 $\overline{\text{H\_IRQ}}$ Timing

Table 3-5 Host Interface Timing

Signal	Parameter	Description	Min	Max	Units
core_clk	t_{CLK}	Internal Core Clock: XIN x PLL Frequency [1]	—	110	MHz
H_ADDR[6:1]	t_{WAS}	H_ADDR setup to trailing edge $\overline{\text{H\_WR}}$ for write cycles	20	—	ns
	t_{WAH}	H_ADDR hold from trailing edge $\overline{\text{H\_WR}}$ for write cycles	3	—	ns
	t_{RAS}	H_ADDR setup to leading edge $\overline{\text{H\_RD}}$ for read cycles [2]	0	—	ns
	t_{RAH}	H_ADDR hold from trailing edge $\overline{\text{H\_RD}}$ for read cycles	0	—	ns
H_DATA[15:0]	t_{WDC}	H_DATA setup to trailing edge $\overline{\text{H\_WR}}$ for write cycles	20	—	ns
	t_{WDH}	H_DATA hold from trailing edge $\overline{\text{H\_WR}}$ for write cycles	3	—	ns
	t_{RDD}	H_DATA driven from leading edge $\overline{\text{H\_RD}}$ for read cycles	0	—	ns
	t_{RDV}	H_DATA valid from leading edge $\overline{\text{H\_RD}}$ for read cycles	—	15	ns
	t_{RDH}	H_DATA hold from trailing edge $\overline{\text{H\_RD}}$ for read cycles	2	15	ns
$\overline{\text{H\_WR}}$	t_{CWE}	$\overline{\text{HCS}}$ Active to $\overline{\text{H\_WR}}$ Active	0	—	ns
	t_{WEC}	$\overline{\text{H\_WR}}$ Inactive to $\overline{\text{HCS}}$ Inactive	3	—	ns
	t_{WEA}	$\overline{\text{H\_WR}}$ active time	37	—	ns
$\overline{\text{H\_RD}}$	t_{CRE}	$\overline{\text{HCS}}$ Active to $\overline{\text{H\_RD}}$ Active	0	—	ns
	t_{REC}	$\overline{\text{H\_RD}}$ Inactive to $\overline{\text{HCS}}$ Inactive	0	—	ns
	t_{REA}	$\overline{\text{H\_RD}}$ active time	$3*t_{CLK} + 8$	—	ns
$\overline{\text{HCS}}$	t_{CSH}	$\overline{\text{HCS}}$ inactive time between accesses	$2*t_{CLK}$	—	ns
$\overline{\text{H\_DMARQ}}$	t_{RQD}	$\overline{\text{H\_DMARQ}}$ valid from internal clock	—	8	ns
$\overline{\text{H\_IRQ}}$	T_{ID}	$\overline{\text{H\_IRQ}}$ valid from internal clock	—	8	ns
$\overline{\text{H\_WAIT}}$	t_{WD}	$\overline{\text{H\_WAIT}}$ valid from internal clock	—	8	ns
$\overline{\text{H\_WAIT}}$	t_{WV}	$\overline{\text{H\_WAIT}}$ valid from $\overline{\text{H\_RD}}$/ $\overline{\text{H\_WR}}$	—	12	ns

1.See “Phase Lock Loop Restrictions” on page 245 for information regarding Core Clock generation.
2.H_ADDR[6:1] must be stable before $\overline{\text{H\_RD}}$ is asserted. Make sure that delays caused by the printed circuit board layout are taken into account when programming the bus timings.

3.2.2 Video Interface AC Timing

Figure 3-7 and Table 3-6 show the AC timing parameters for the video interface.

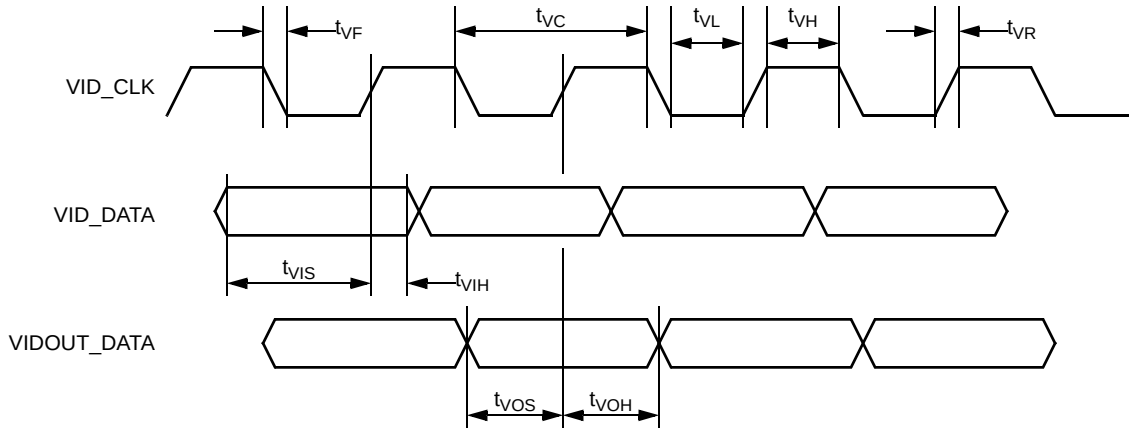


Figure 3-7 Video Interface Timing Diagram

Table 3-6 Video Interface AC Timing Values

Signal	Parameter	Description	Timing Value (ns.)		
			Min	Typ	Max
VID_CLK	t_{VC}	VID_CLK Cycle Time (27 MHz typical)	25	37	—
	t_{VH}	VID_CLK High Time	.4*t_{VC}	$t_{VC/2}$	.6*t_{VC}
	t_{VL}	VID_CLK Low Time	t_{VC} - t_{VH}		
	t_{VR}	VID_CLK Slew (Rise Time)	Not Applicable		
	t_{VF}	VID_CLK Slew (Fall Time)	Not Applicable		
VID_DATA	t_{VIS}	VID_DATA Set-up Time to VID_CLK	5.5	—	—
	t_{VIH}	VID_DATA Hold Time from VID_CLK	0	—	—
VIDOUT_DATA	t_{VOS}	VIDOUT_DATA Set-up Time to VID_CLK	16	—	—
	t_{VOH}	VIDOUT_DATA Hold Time from VID_CLK	6	—	—

3.2.3 Audio Interface AC Timing

This section gives the AC timing parameters for the MG1264 Codec's audio interface. Figure 3-8 shows the relationships between the three audio clocks. Figure 3-9 shows the timing waveforms. Table 3-7 lists the AC timing for Audio Operations.

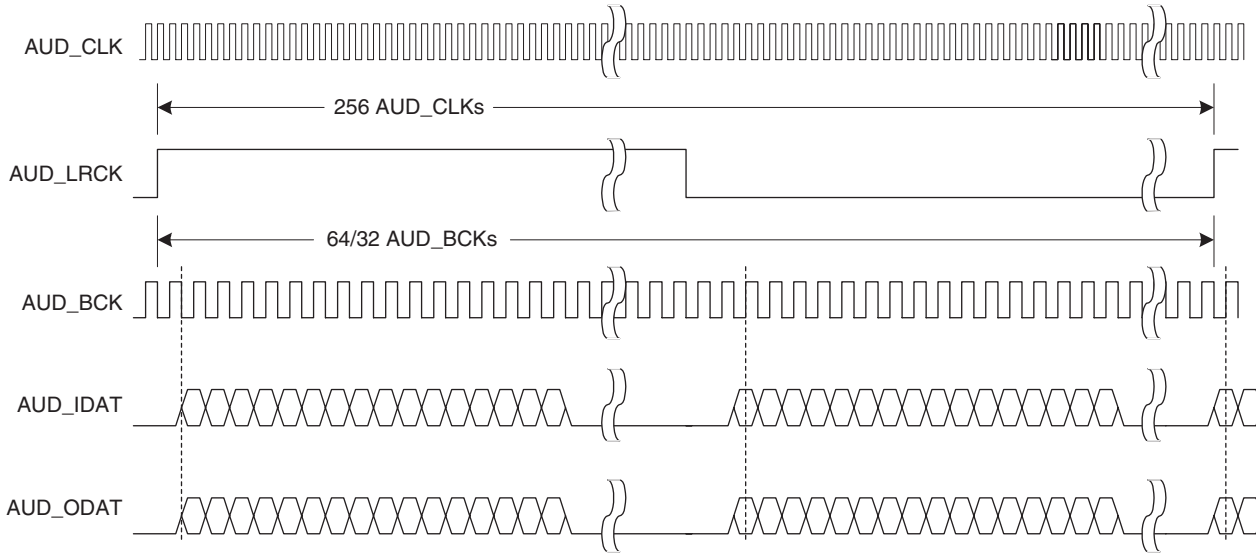


Figure 3-8 Audio Timing Diagram

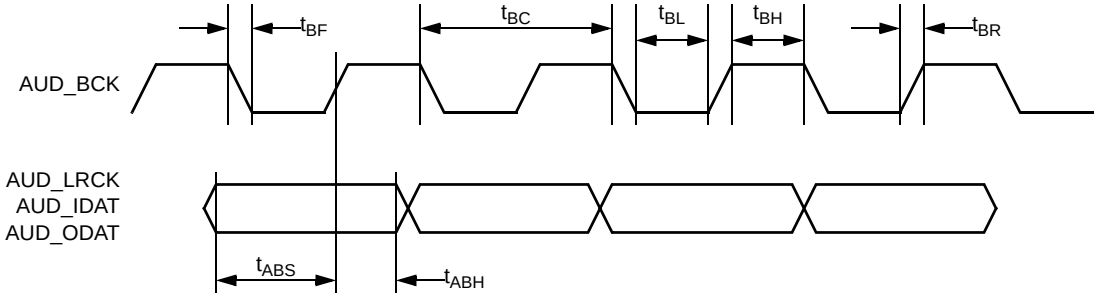


Figure 3-9 Audio Interface Timing Diagram

Table 3-7 Audio Interface AC Timing Values

Signal	Parameter	Description	Timing Value (ns.)		
			Min	Typ	Max
AUD_BCK	t_{BC}	AUD_BCK Cycle Time (Fs = 48 kHz, 64 BCK/Sample)	—	325	—
	t_{BC}	AUD_BCK Cycle Time (Fs = 48 kHz, 32 BCK/Sample)	—	651	—
	t_{BC}	AUD_BCK Cycle Time (Fs = 32 kHz, 64 BCK/Sample)	—	488	—
	t_{BC}	AUD_BCK Cycle Time (Fs = 32 kHz, 32 BCK/Sample)	—	977	—
	t_{BH}	AUD_BCK High Time	$.4*t_{BC}$	$t_{BC/2}$	$.6*t_{BC}$
	t_{BL}	AUD_BCK Low Time	$T_{BC} - T_{BH}$		
	t_{BR}	AUD_BCK Slew (Rise Time)	—	—	1.5
	t_{BF}	AUD_BCK Slew (Fall Time)	—	—	1.6
AUD_LRCK AUD_ODAT AUD_IDAT	t_{ABS}	Set-up Time to AUD_BCK	8	—	—
	t_{ABH}	Hold Time from AUD_BCK	3	—	—

3.2.4 SDRAM Interface AC Timing

The MG1264 Codec adheres to the JEDEC definition of timing for SDRAMs. Refer to the appropriate specifications when designing the SDRAM Interface.

Chapter 4. MG1264 Codec Host Interface

The System Host CPU controls the MG1264 Codec through the Host Interface. The MG1264 Codec Host Interface also serves as the compressed data interface. This interface allows for directly-addressable access to the MG1264 Codec DRAM, the MG1264 Codec Bitstream write FIFO, and the MG1264 Codec registers.

4.1 MG1264 Codec Host Interface Physical Description

The MG1264 Codec Host Interface is modeled on the commonly used generic asynchronous-style interface. It consists of a 16-bit data path (H_DATA[15:0], six bits of address (H_ADDR[6:1]), and control signals.

4.1.1 Connection Diagram

The MG1264 Codec Host Interface connection diagram is shown in Figure 4-1.

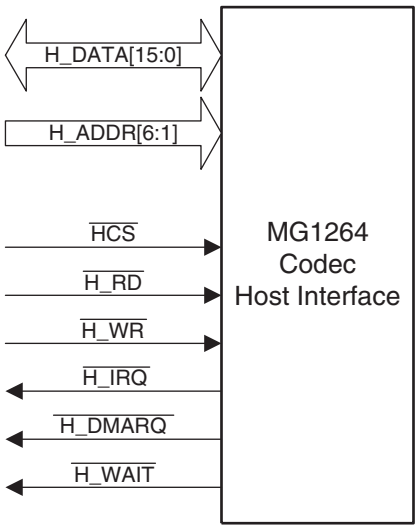


Figure 4-1 MG1264 Codec Host Interface Connection Diagrams

The MG1264 Codec Host Interface has a single chip select and six address lines. All of the device's resources reside in a single address space, and the registers that can be addressed by the six address lines are shown in Table 4-2.

4.1.2 MG1264 Codec Host Interface Signals

The signals that comprise the MG1264 Codec Host Interface are shown in Table 4-1.

Table 4-1 MG1264 Codec Host Interface Pin Descriptions

Pin Name	Signal Name	Direction	Description
H_DATA[15:0]	Data [15:0]	Bidirectional	16-bit Host Data Bus
H_ADDR[6:1]	Address [6:1]	Inputs	Six bits of Host Address
$\overline{\text{HCS}}$	$\overline{\text{Host Chip Select}}$	Input	Active Low Host Chip Select. This chip select is used to access the MG1264 Codec's Internal registers, External memory, bitstream read and write FIFO registers.
$\overline{\text{H\_RD}}$	$\overline{\text{RE}}$	Input	Active Low Read Enable
$\overline{\text{H\_WR}}$	$\overline{\text{WE}}$	Input	Active Low Write Enable
$\overline{\text{H\_IRQ}}$	$\overline{\text{Interrupt}}$	Output	Active Low Host Interrupt Request
$\overline{\text{H\_DMARQ}}$	$\overline{\text{Host DMA Request}}$	Output	Bitstream DMA Request associated with the Bitstream port
$\overline{\text{H\_WAIT}}$	$\overline{\text{Wait}}$	Output	Active low wait pin. The MG1264 Codec asserts this pin to extend the bus cycle until it is able to accept data (during a write cycle) or present data (during a read cycle). $\overline{\text{H\_WAIT}}$ can stay asserted or deasserted independently of HCS. If the H_WAIT signal is used in multi-chip designs, this must be accounted for by using an external multiplexer or other means to separate the different H_WAIT signals.

4.2 MG1264 Codec Host Interface Logical Description

The MG1264 Codec Host Interface works in two completely different modes:

- System Control
- Compressed Data I/O Interface

These are discussed in the sections that follow.

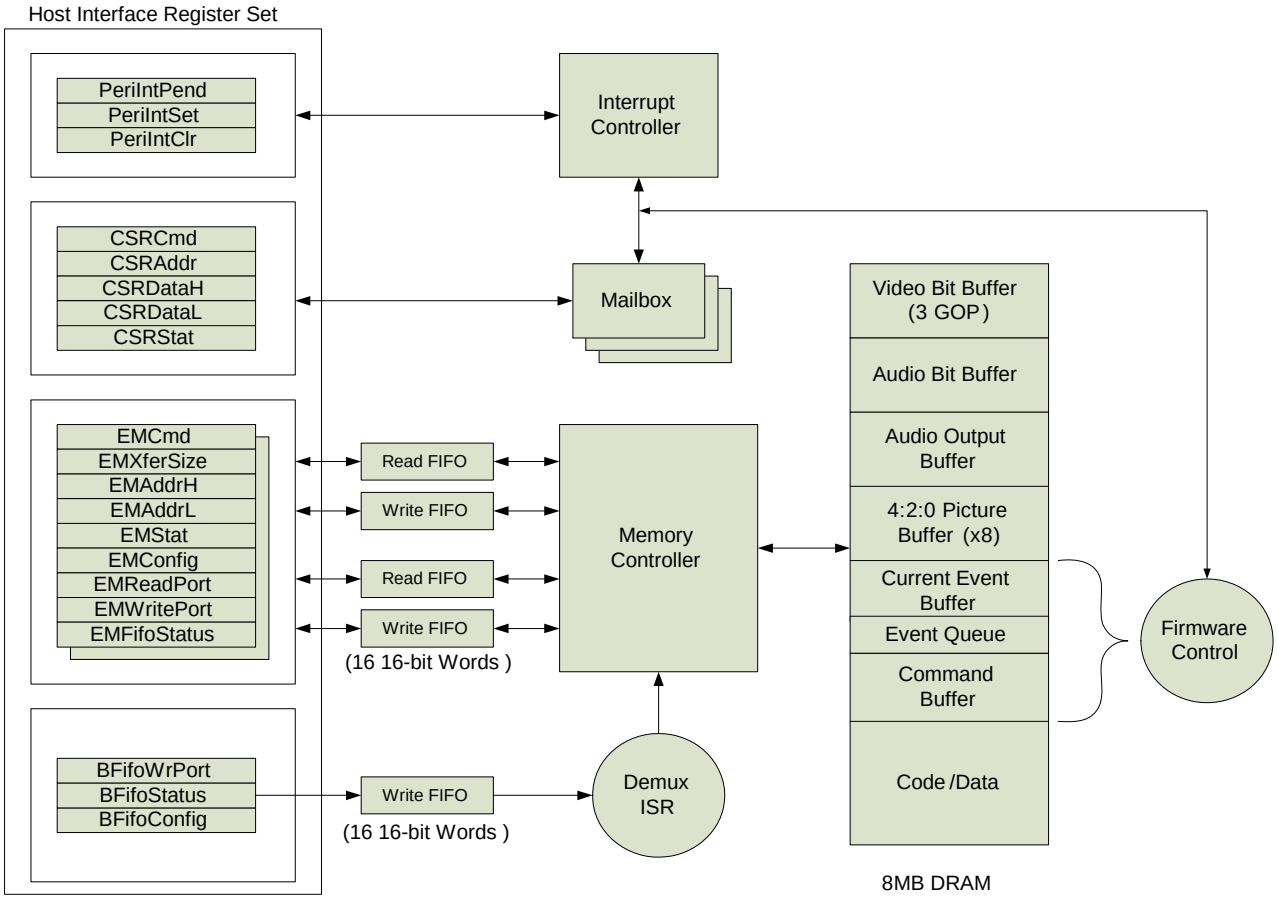


Figure 4-2 Register Logical View

4.2.1 System Control

The MG1264 Codec is controlled through the MG1264 Codec Host Interface. When the MG1264 Codec is powered up, the System Host CPU must first download the firmware through the MG1264 Codec Host Interface, and then initialize the MG1264 Codec. The System Host CPU controls the operation of the MG1264 Codec by reading and writing specific registers inside the MG1264 Codec.

The MG1264 Codec is able to accept new commands or requests from the System Host CPU at least once every frame period. Control commands such as start/stop/pause are executed within one frame time of being issued.

4.2.2 Compressed Data I/O Through the MG1264 Codec Host Interface

The MG1264 Codec Host Interface also transports compressed data in to (decoding) and out of (encoding) the MG1264 Codec. The System Host CPU can use Direct Memory Access (DMA) to facilitate these transfers.

4.2.3 Interrupts

There is a single interrupt pin defined: $\overline{\text{H\_IRQ}}$. The MG1264 Codec has four interrupt sources that are logically OR'd together internally to form the $\overline{\text{H\_IRQ}}$:

- CSRInt: Configuration Status Register Interrupt
- EMInt: External Memory Interrupt
- BMInt: Bitstream Memory Interrupt
- MBint: Mailbox Interrupt

For information on the Interrupt Registers, refer to “Peripheral Interrupt Registers” on page 73.

4.2.4 DMA Channels

The MG1264 Codec has two generic External Memory DMA engines. One is for System Host CPU access to the MG1264 Codec’s DRAM including the mailbox. You can find information on this DMA interface in the section “External Memory Access Registers” on page 79.

The other is for Bitstream transfers. The Bitstream DMA is used for reading a bitstream from, and writing a bitstream to the Bitstream Write FIFO. You can also find information on this DMA interface in the section “Bitstream Write FIFO Access Registers” on page 85.

4.2.5 Latency Considerations

Because internal operations such as DRAM and register access can incur a lot of latency, the MG1264 Codec’s Host Interface uses an indirect access method to access the internal MG1264 Codec’s processor resources. In this mode of operation, read and write accesses are deterministic and no Host Ready (or Wait) signaling is needed.

4.3 Read/Write Timing

This section provides generic timing information for the MG1264 Codec Host Interface. For specific timing information, refer to “Specifications” on page 43. For information on the programming sequence needed to read or write a register, refer to “Register Definitions” on page 71.

The Read/Write control signals are programmable, and can be set to work in either Read Enable and Write Enable mode (default) or Read/Write (RD/$\overline{WR}$) and Enable ($\overline{ENABLE}$) mode. The MG1264 Codec defaults to the separate Read Enable and Write Enable signalling as shown in Figure 4-3 and Figure 4-4.

To put the host interface into Read/Write and Enable mode (Figure 4-5 and Figure 4-6), the very first transaction on the read bus must be a Write transaction using the separate Enable and RD/$\overline{WR}$ signaling to register address 0x18. This register is not defined as a valid register and a write to it has no logical effect other than to put the chip into separate $\overline{ENABLE}$ and RD/$\overline{WR}$ mode. A data value of 0x0000 should be used.

4.3.1 Read Timing Sequence in Read Enable Mode

Figure 4-3 shows the timing for a System Host CPU read from the MG1264 Codec in Read Enable mode.

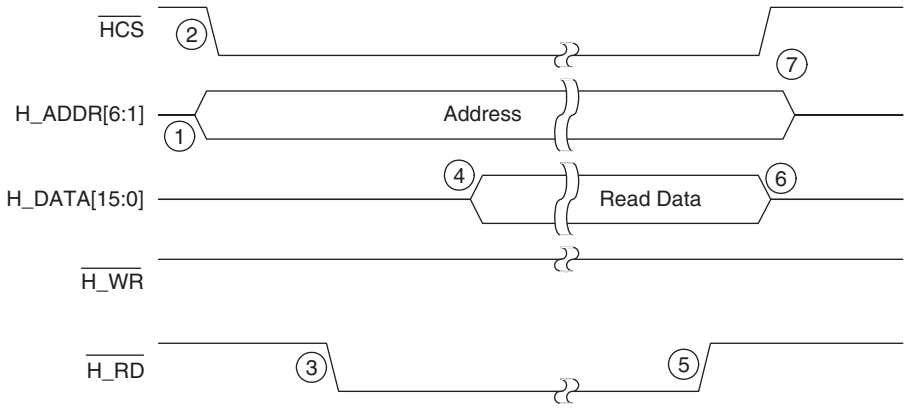


Figure 4-3 Read Access Timing in Read Enable Mode

1. The System Host CPU must assure that the address bus (H_ADDR[6:1]) is stable before asserting Host Chip Select ($\overline{\text{HCS}}$).
2: The System Host CPU asserts the Host Chip Select signal to inform the MG1264 Codec that a read is in process. When Host Chip Select ($\overline{\text{HCS}}$) is used, it accesses the MG1264 Codec's Internal registers and External memory.
3: The System Host CPU asserts the Host Read Enable ($\overline{\text{H\_RD}}$) signal to inform the MG1264 Codec that the operation will be a read.
4: The data becomes available on H_DATA[15:0].
5: Once the data has been taken, the System Host CPU de-asserts the Host Read Enable ($\overline{\text{H\_RD}}$) signal to indicate to the MG1264 Codec that the transaction is complete.
6: The MG1264 Codec removes the output data from the data bus (H_DATA[15:0]).
7: The System Host CPU then de-asserts the address bus (H_ADDR[6:1]) and the Host Chip Select to complete the transaction.

4.3.2 Write Data Timing in Write Enable Mode

Figure 4-4 shows the timing for a System Host CPU write to the MG1264 Codec in Write Enable mode.

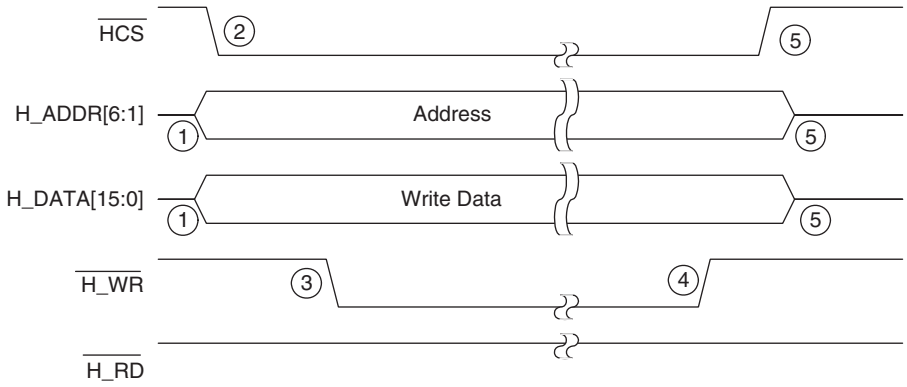


Figure 4-4 Write Access Timing in Write Enable Mode

1. The System Host CPU must assure that the address bus (H_ADDR[6:1]) and data to be written (on H_DATA[15:0]) are stable before asserting the Host Chip Select ($\overline{\text{HCS}}$).
2: The System Host CPU asserts the Host Chip Select signal to inform the MG1264 Codec that a write is in process. When the Host Chip Select ($\overline{\text{HCS}}$) is used, it accesses the MG1264 Codec's Internal registers and External memory.
3: The System Host CPU asserts the Host Write Enable ($\overline{\text{H\_WR}}$) signal to inform the MG1264 Codec that the operation will be a write.
4: The System Host CPU de-asserts the Host Write Enable ($\overline{\text{H\_WR}}$) signal to indicate to the MG1264 Codec that the write is complete.
5: The System Host CPU de-asserts the Address bus (H_ADDR[6:1]), Write Data bus (H_DATA[15:0]), and the Host Chip Select to indicate to the MG1264 Codec that the transaction is complete.

4.3.3 Read Timing Sequence in Read/Write and Enable Mode

Figure 4-3 shows the timing for a System Host CPU read from the MG1264 Codec in Read/Write mode.

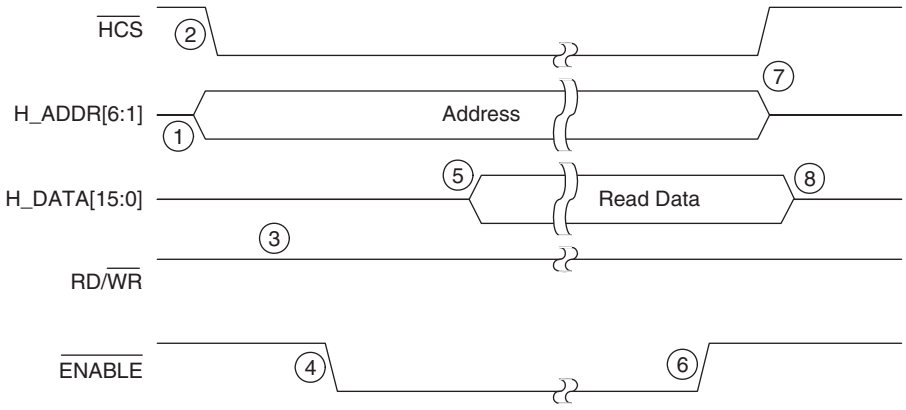


Figure 4-5 Read Access Timing in Read/Write and Enable Mode

1. The System Host CPU must assure that the address bus (H_ADDR[6:1]) is stable before asserting Host Chip Select ($\overline{\text{HCS}}$).
2: The System Host CPU asserts the Host Chip Select signal to inform the MG1264 Codec that a read is in process. When Host Chip Select ($\overline{\text{HCS}}$) is used, it accesses the MG1264 Codec's Internal registers and External memory.
3: The System Host CPU sets the Read/Write signal (RD/$\overline{\text{WR}}$) high to inform the MG1264 Codec that the operation will be a read.
4: The System Host CPU asserts the $\overline{\text{ENABLE}}$ signal to start the read cycle.
5: The data becomes available on H_DATA[15:0].
6: Once the data has been taken, the System Host CPU de-asserts the $\overline{\text{ENABLE}}$ signal to indicate to the MG1264 Codec that the transaction is complete.
7: The System Host CPU then de-asserts the address bus (H_ADDR[6:1]) and the Host Chip Select to complete the transaction.
8: The MG1264 Codec removes the output data from the data bus (H_DATA[15:0]).

4.3.4 Write Data Timing in Read/Write and Enable Mode

Figure 4-4 shows the timing for a System Host CPU write to the MG1264 Codec in Read/Write and Enable mode.

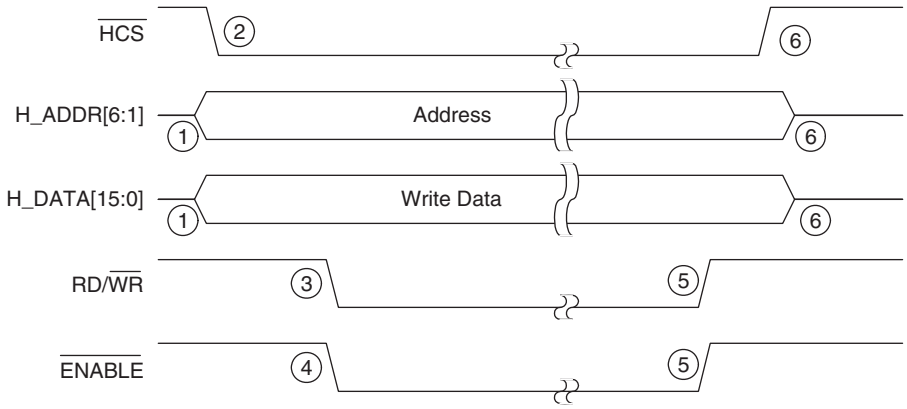


Figure 4-6 Write Access Timing in Read/Write and Enable Mode

1. The System Host CPU must assure that the address bus (H_ADDR[6:1]) and data to be written (on H_DATA[15:0]) is stable before asserting the Host Chip Select ($\overline{\text{HCS}}$).
2: The System Host CPU asserts the Host Chip Select signal to inform the MG1264 Codec that a write is in process. When the Host Chip Select ($\overline{\text{HCS}}$) is used, it accesses the MG1264 Codec's Internal registers and External memory.
3: The System Host CPU sets the Read/Write signal (RD/$\overline{\text{WR}}$) low to inform the MG1264 Codec that the operation will be a write.
4: The System Host CPU asserts the $\overline{\text{ENABLE}}$ signal to start the write cycle.
5: The System Host CPU de-asserts the RD/$\overline{\text{WR}}$ signal and $\overline{\text{ENABLE}}$ signals to indicate to the MG1264 Codec that the write is complete.
6: The System Host CPU de-asserts the Address bus (H_ADDR[6:1]), Write Data bus (H_DATA[15:0]), and the Host Chip Select to indicate to the MG1264 Codec that the transaction is complete.

4.4 DMA Transfers

The MG1264 Codec can be configured to do DMA transfers. When the MG1264 Codec is in DMA mode, the transfers on the external bus are a sequence of individual read and write transactions to a FIFO port mapped to a host interface register. See “Accessing External Memory Port 1 and Port 2” on page 77 for information on how to set up a DMA transfer.

When in DMA mode, the individual read or write transactions making up the DMA transactions must be paced. The MG1264 Codec signals the external host that it is ready to accept a read or write transaction. The pacing is accomplished using one of three mechanisms:

- The external $\overline{\text{H\_DMARQ}}$ pin
- A register bit (EMFifoRdReq/ EMFifoWrReq)
- The external $\overline{\text{H\_WAIT}}$ pin

4.4.1 Pacing using the $\overline{\text{H\_DMARQ}}$ Pin

The MG1264 Codec asserts the $\overline{\text{H\_DMARQ}}$ pin when a programmable threshold (EMDThresh, see page 83) is reached in the DMA transfer FIFO. For a read DMA, the MG1264 Codec asserts the $\overline{\text{H\_DMARQ}}$ pin when EMDThresh number of 16-bit words is available to be transferred to the System Host CPU. The MG1264 Codec deasserts the $\overline{\text{H\_DMARQ}}$ pin once the number of 16-bit words available to be read falls below EMDThresh.

For a write DMA, the $\overline{\text{H\_DMARQ}}$ pin is asserted when the MG1264 Codec is able to accept EMDThresh number of 16-bit words to be written. The $\overline{\text{H\_DMARQ}}$ pin is de-asserted once the number of 16-bit words available to be written falls below EMDThresh.

4.4.2 Pacing using the EMFifoRdReq/EMFifoWrReq Bits

The EMFifoRdReq or EMFifoWrReq Bits in the EMFifoStatus Register (see page 84) are reflections of the $\overline{\text{H\_DMARQ}}$ pin and are set accordingly if in read or write DMA mode.

4.4.3 Pacing using the $\overline{\text{H\_WAIT}}$ Pin

Pacing using the $\overline{\text{H\_WAIT}}$ pin is slightly different than in $\overline{\text{H\_DMARQ}}$ mode. In this case, the external host does not use the $\overline{\text{H\_DMARQ}}$ or the EMFifoRdReq/EMFifoWrReq mechanisms. In the case of a read DMA transaction, the System Host CPU initiates read transactions without monitoring the $\overline{\text{H\_DMARQ}}$ pin or the EMFifoRdReq bits. If the MG1264 Codec does not currently have data available for reading, it asserts the $\overline{\text{H\_WAIT}}$ signal during that individual read transaction until data is available. The transaction is not completed until $\overline{\text{H\_WAIT}}$ is deasserted.

In a write DMA transaction, the external host initiates write transactions without monitoring the $\overline{\text{H\_DMARQ}}$ pin or the EMFifoRdReq bits. If the MG1264 Codec is not currently able to accept write data, it asserts the $\overline{\text{H\_WAIT}}$ signal during that individual write transaction until it is able to accept data. The transaction is not completed until $\overline{\text{H\_WAIT}}$ is de-asserted.

4.5 MG1264 Codec Register Indirect Access

The System Host CPU processor can only indirectly access the MG1264 Codec's internal Configuration and Status (CSR) registers and Mailbox registers (see Figure 4-2). This is done through a set of registers mapped to the Host Chip Select ($\overline{\text{HCS}}$) over the MG1264 Codec Host Interface. These registers are not accessed during normal operation, and indirect addressing is typically only used by the bootloader.

4.5.1 Reading a Register

The procedure to read an MG1264 Codec register is:

1. Before accessing a register, set up the `PeriIntEn` register to enable the Configuration or Status Register (CSR) interrupt, if that is the preferred method for getting the "Access Done" message. This only needs to be done once for all CSR accesses.
2: Write the Address to the `CSRAddr` register.
3: Write the Command bits (`CSRAccess` = 0) to the `CSRCmd` register.
4: Poll the `CSRDone` bit in the `CSRStat` register, or wait for the interrupt.
5: Read the return data from the `CSRRdDataH` and `CSRRdDataL` registers.
6: Read the `CSRStat` register and check that it has the expected value.
7: Clear the `CSRInt` bit in the `PeriIntPend` register, if using interrupts or clear the `CSRDone` bit in the `CSRStatus` register, if polling.

4.5.2 Writing a Register

The procedure to write a MG1264 Codec register is:

1. Before accessing a register, set up the `PeriIntEn` register to enable the Configuration or Status Register (CSR) interrupt, if that is the preferred method for getting the "Access Done" message. This only needs to be done once for all CSR accesses.
2: Write the data to be written to the `CSRWrDataH` and `CSRWrDataL` registers.
3: Write the Address the `CSRAddr` register.
4: Write the Command bits (CSRAccess = 0) to the `CSRCmd` register.
5: Poll the `CSRDone` bit in the `CSRStat` register, or wait for the interrupt.
6: Read the `CSRStat` register and check that it has the expected value.

Usage Note: In some cases, it may be necessary to read `CSRRdData` to check a value returned by the internal processor if the operation is more complex than a simple register read or write.

7: Clear the `CSRInt` bit in the `PeriIntPend` register, if using interrupts or clear the `CSRDone` bit in the `CSRStatus` register, if polling.

4.6 Programming the MG1264 Codec Host Interface

4.6.1 Register Maps

This section provides information on the registers used to program the MG1264 Low Power H.264 and AAC Codec for Mobile Devices. These registers are addressed when the Host Chip Select ($\overline{HCS}$) signal is asserted.

Table 4-2 shows the MG1264 Codec Internal Configuration and Status Registers. These registers are discussed in detail in “Configuration, Data, and Status Registers” on page 71.

Table 4-2 MG1264 Codec Internal Configuration and Status Registers

Register	Offset	Access	Description	Page
CSRCmd	0x0020	R/W	Configuration/Status Register Command	71
CSRAddr	0x0022	R/W	Configuration/Status Register Address	71
CSRWrDataH	0x0024	R/W	Configuration/Status Register Write Data High	71
CSRWrDataL	0x0026	R/W	Configuration/Status Register Write Data Low	71
CSRRdDataH	0x0028	Read	Configuration/Status Register Read Data High	72
CSRRdDataL	0x002A	Read	Configuration/Status Register Read Data Low	72
CSRStat	0x002C	R/W	Configuration/Status Register Status	72
PeriIntPend	0x002E	R/W	Peripherals Interrupt Pending	73
PeriIntEnSet	0x0030	R/W	Peripherals Interrupt Enable - Set	73
PeriIntEnClr	0x0032	R/W	Peripherals Interrupt Enable - Clear	73
ClkConfig	0x0034	R/W	Clock Configuration Register	74
PLL Dividers	0x0036	R/W	PLL Dividers Register	75
ChipID	0x0038	R	Chip ID Register	76

Table 4-3 shows the MG1264 Codec External Memory Interface Port 1 Registers. These registers are discussed in detail in “Accessing External Memory Port 1 and Port 2” on page 77 and “Reading the MG1264 Codec’s External Memory” on page 77.

Table 4-3 MG1264 Codec External Memory Interface Port 1 Registers

Register	Offset	Access	Description	Page
EM1Cmd	0x0000	R/W	External Memory DMA Command	79
EM1XferSize	0x0002	R/W	External Memory DMA Transfer Size	79
EM1SrcAddrH	0x0004	R/W	External Memory DMA Source Address High or Starting Vertical/Y Source Address	80
EM1SrcAddrL	0x0006	R/W	External Memory DMA Source Address Low or Starting Horizontal/X Source Address	80
EM1DestAddrH	0x0008	R/W	External Memory DMA Destination Address High or Starting Vertical/Y Destination Address	80
EM1DestAddrL	0x000A	R/W	External Memory DMA Destination Address Low or Starting Horizontal/X Destination Address	80
EM1Status	0x000C	Read	External Memory DMA Status	82
EM1RemCount	0x000E	Read	External Memory DMA Transfer Remainder Count	82
EM1Config	0x0010	R/W	External Memory DMA Configuration	83
EM1FifoRdPort	0x0012	Read	External Memory DMA FIFO Read Port (from memory)	84
EM1FifoWrPort	0x0014	R/W	External Memory DMA FIFO Write Port (to memory)	84
EM1FifoStatus	0x0016	Read	Bitstream Memory DMA Status	84

Table 4-4 shows the MG1264 Codec External Memory Interface Port 2 Registers. These registers are also discussed in detail in "Accessing External Memory Port 1 and Port 2" on page 77 and "Reading the MG1264 Codec's External Memory" on page 77.

Table 4-4 MG1264 Codec External Memory Interface Port 2 Registers

Register	Offset	Access	Description	Page
EM2Cmd	0x0040	R/W	Bitstream Memory DMA Command	79
EM2XferSize	0x0042	R/W	Bitstream Memory DMA Transfer Size	79
EM2SrcAddrH	0x0044	R/W	Bitstream Memory DMA Source Address High or Starting Vertical/Y Source Address	80
EM2SrcAddrL	0x0046	R/W	Bitstream Memory DMA Source Address Low or Starting Horizontal/X Source Address	80
EM2DestAddrH	0x0048	R/W	Bitstream Memory DMA Destination Address High or Starting Vertical/Y Destination Address	80
EM2DestAddrL	0x004A	R/W	Bitstream Memory DMA Destination Address Low or Starting Vertical/Y Source Address	80
EM2Status	0x004C	Read	Bitstream Memory DMA Status	82
EM2RemCount	0x004E	Read	Bitstream Memory DMA Transfer Remainder Count	82
EM2Config	0x0050	R/W	Bitstream Memory DMA Configuration	83
EM2FifoRdPort	0x0052	Read	Bitstream Memory DMA FIFO Read Port (from memory)	84
EM2FifoWrPort	0x0054	R/W	Bitstream Memory DMA FIFO Write Port (to memory)	84
EM2FifoStatus	0x0056	Read	Bitstream Memory DMA FIFO Status	84

Table 4-5 shows the MG1264 Codec Bitstream Interface Registers. These registers are discussed in detail in "Bitstream Write FIFO Access Registers" on page 85.

Table 4-5 MG1264 Codec Bitstream Interface Registers

Register	Offset	Access	Description	Page
BFifoWrPort	0x0060	R/W	Bitstream FIFO Write Port (to Media Engine)	85
BFifoStatus	0x0062	Read	Bitstream FIFO Status Register	85
BFifoConfig	0x0064	R/W	Bitstream FIFO Command Register	85

4.7 Register Definitions

4.7.1 Configuration, Data, and Status Registers

Command/Status Register Command CSRCmd Offset: 0x0020

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSR Access	CSRLen			Reserved				CSRBlockID							

Reserved fields should be ignored (masked) when read, and only 0's should be written to them.	
CSRAccess	When a 0 is written to this field, it initiates a CSR read from the address provided in the CSRAddr register. When a 1 is written to this field, it initiates a CSR write to the address provided in the CSRAddr register with the data provided in the CSRWrData register.
CSRLen	000 = 4 byte (word) access 001 = 1 byte access 010 = 2 byte (halfword) access Other codes are reserved and should not be used.
CSRBlockID	Block ID for a register access

Command/Status Register Address CSRAddr Offset: 0x0022

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRAddr															

CSRAddr	Address (within a register block) for register access. Expected to be word-aligned (bits [1:0] are 0) for 4-byte access and half-word aligned (bit [0] is 0) for 2-byte access.

Command/Status Register Write Data High CSRWrDataH Offset: 0x0024

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRWrDataH															

CSRWrDataH	High 16-bit register from which the data for a CSR write is taken. Used with CSRWrDataL.

Command/Status Register Write Data Low CSRWrDataL Offset: 0x0026

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRWrDataL															

CSRWrDataL	Low 16-bit register from which the data for a CSR write is taken. Used with CSRWrDataH

Command/Status Register Read Data High ***CSRRdDataH*** ***Offset: 0x0028***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRRdDataH															

CSRRdDataH	High 16-bit register containing the data returned for a CSR read or the status information returned for a write. Used with CSRRdDataL This register is read-only.

Command/Status Register Read Data Low ***CSRRdDataL*** ***Offset: 0x002A***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRRdDataL															

CSRRdDataL	Low 16-bit register containing the data returned for a CSR read or the status information returned for a write. Used with CSRRdDataH. This register is read-only.

Command/Status Register Status ***CSRStat*** ***Offset: 0x002C***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CSRRespID								Res	CSRRespLen			Res		CSR-Err	CSR-Done

Reserved fields should be ignored (masked) when read and only 0’s should be written to them.	
CSRRespID	Block ID information from l_obid port when a CSR access is completed (which block responded). If it doesn't match the CSRBlockID originally programmed, then something is wrong. This field is read-only.
CSRRespLen	Length of the access actually done. For a write, it should be 1; for a read, it should match the CSRLen code originally programmed. If not, then something is wrong. This field is read-only.
CSRErr	If set to 1 when CSRDone is set, an error occurred in the access. This should never happen. This field is read-only.
CSRDone	Set to 1 after each CSRAccess completes. When the hardware sets this bit to 1, the read data (or write response status) is available in the CSRRdData register. It is not required to clear this bit before initiating a new access; however, software should clear it if it is polling this bit to determine when an access completes, instead of using the CSRInt interrupt.

4.7.2 Peripheral Interrupt Registers

Peripheral Interrupt Pending Register ***PeriIntPend*** ***Offset: 0x002E***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											Mbox 1Int	Mbox 0Int	BMInt	EMInt	CSR Int

Reserved fields should be ignored (masked) when read and only 0's should be written to them.
The bits in these registers are “sticky”; if an interrupt event occurs and sets a bit, the bit stays set until it is cleared. A bit can only be cleared by writing a 1 to it; writing a 0 to it has no effect (so the same value that was read from the register can be written back to clear only the interrupt bits that were previously set, not any new ones).

Mbox1Int	This bit is a logical OR of the Mbox1RdyCPU0Int and Mbox1ReadCPU0Int field of the MboxIntCPU0 QCC register.
Mbox0Int	This bit is a logical OR of the Mbox0RdyCPU0Int and Mbox0ReadCPU0Int field of the MboxIntCPU0 QCC register.
BMInt	Bitstream Read Memory DMA transfer is done (BMBusy goes from 1 to 0)
EMInt	External Memory DMA transfer is done (EMBusy goes from 1 to 0)
CSRInt	CSR Access is done.

Peripheral Interrupt Enable Set Register ***PeriIntEnSet*** ***Offset: 0x0030***

The Peripheral Interrupt Enable function is implemented with separate “Set” and “Clear” register addresses, allowing each interrupt enable bit to be set or cleared independently of the other bits, so that no read-modify-write cycles are required.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											PeriIntEnSet				

Reserved fields should be ignored (masked) when read and only 0's should be written to them.

PeriIntEnSet	Writing a 1 to a bit at the address for PeriIntEnSet sets the corresponding bit to 1 in PeriIntEn; writing a 0 has no effect. Reading the register at the address for PeriIntEnSet returns the current value for PeriIntEn.

Peripheral Interrupt Enable Clear Register ***PeriIntEnClr*** ***Offset: 0x0032***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											PeriIntEnClr				

Reserved fields should be ignored (masked) when read and only 0's should be written to them.

PeriIntEnClr	Writing a 1 to a bit at the address for PeriIntEnClr clears the corresponding bit in PeriIntEn; writing a 0 has no effect. Reading the register at the address for PeriIntEnClr returns the current value for PeriIntEn.

4.7.3 Clock and Configuration Registers

Clock Configuration Register ***ClkConfig*** ***Offset: 0x0034***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved													Vclk Invert	PLL Power Down	ClkEn

Reserved fields should be ignored (masked) when read and only 0's should be written to them.	
VclkInvert	Internally inverts VID_CLK. This allows for sampling of video pins on the negative edge of VLK. It is very useful for solving setup and hold issues on the video bus. 0: video_clk = VID_CLK (default) 1: video_clk = VID_CLK
PLLPowerDown	The PLL is put in powerdown mode. Note: ClkGate must be enabled (set to 0) first (separate register programming transactions) before setting PLLPowerDown to 1. PLLPowerDown must be set to 0 before clearing (set to 1) ClkGate. 0: Normal Operation 1: PLL is in powerdown (default)
ClkEn	This register glitchlessly turns off core_clk, video_clk, and audio_aclk and holds them low. 0: Clocks are gated off and held low (default) 1: Clocks are active

Phase Lock Loop Dividers ***PLLDividers*** ***Offset: 0x0036***

The Core Clock frequency (core_clk) is generated using an internal Phase Lock Loop (PLL) from the clock input on the XIN pin. The Core Clock frequency is calculated using the following equation:

$$\text{core\_clk} = XIN \times \frac{M}{X}$$

where M is set using the PLLFeedBackDivider field and X is set using the PLLOutputDivider field of the PLLDivider register (see below).

The maximum frequency for the MG1264 Codec Core Clock is 110 MHz. at worse case conditions. However, the MG1264 Codec has a restriction on the relationship between the clock input on the VID_CLK pin (video Input Clock) and the Core Clock. The relationship can best be described as follows: The maximum Core Clock frequency of the MG1264 Codec is one PLL resolution below four times the clock on the VID_CLK pin. (See “Phase Lock Loop Restrictions” on page 245.)

For instance, if VID_CLK = 27 MHz, the Core Clock must be less than 4 x 27 MHz (108 MHz.), and 104.625 MHz. is the highest Core Clock frequency below the 4 x 27 MHz (108 MHz.) limit. The equation for generating a 104.625 MHz Core Clock is:

$$104.625MHz = 27MHz \times \frac{31}{8}$$

Where the M/X ratio of 31/8 meets the requirement of being one PLL resolution below four times the clock on the VID_CLK pin.

When programming the PLL dividers, the ClkEn bit in the Clock Configuration register must be set to 0 before setting the dividers or PLLBypass. Once programmed, the PLL must be given time (0.5 ms.) to lock before setting ClkEn = 1. When programming PLLBypass, the PLL does not need time to lock and ClkEn can be set to 1 immediately.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PLL Bypass	Reserved							PllFeedBackDivider						PLLOutput Divider	

Reserved fields should be ignored (masked) when read and only 0's should be written to them.	
PLLBypass	The register bypasses the PLL and sets the pll_clk = XIN. 0: PLL is in normal mode (default) 1: PLL is bypassed.
PLLFeedBack Divider	The PLL feedback divider *M*. The default=31 Restriction: 2<=*M*<=37 for 27 MHz input clock.
PLLOutput Divider	00: The PLL output divider X is set to 8 (Default). 01: The PLL output divider X is set to 1. 10: The PLL output divider X is set to 2. 11: The PLL output divider X is set to 4.

Chip ID Register ***ChipID*** ***0x0038***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ProductID								TapeOutRev				MaskID			

this is a Read-only register	
ProductID	8’b00000001
TapeOutRev	4’b0001
MaskID	4’b0000

4.7.4 Accessing External Memory Port 1 and Port 2

The System Host CPU accesses the MG1264 Codec's external DRAM through a set of registers mapped to the Host Chip Select ($\overline{\text{HCS}}$) pin over the MG1264 Codec Host Interface. The base address of this device, and the offset for each of these registers is listed in Table 4-2. These registers are explained in detail in the sections that follow.

Two generic External Memory DMA engines have been implemented in the MG1264 Codec. The first one (EM1) is intended for generic System Host CPU access to the DRAM, including the mailbox. It is selected by asserting the $\overline{\text{HCS}}$ pin and register addresses 0x0000 to 0x0016. The other (EM2) is intended for compressed bitstream transfers and is selected by asserting $\overline{\text{HCS}}$ and register addresses 0x0040 to 0x0056. These interfaces are identical designs.

Usage Note: While these two interfaces are identical in design, the MG1264 Codec only brings the DMA request signal from the device when H_ADDR[6] is high (Bitstream write) out to a pin. $\overline{\text{H\_DMARQ}}$ is a logical OR of the DMA requests for External Memory Port 1 and 2. When the EMCmd register is written with an active value, the $\overline{\text{H\_DMARQ}}$ signal represents the request generated from the External memory access logic. Otherwise, it represents the request signal generated from the Bitstream FIFO logic.

During initialization, the System Host CPU can use the $\overline{\text{HCS}}$ pin and H_ADDR = 1 to do a block-level DMA of a DRAM image into the MG1264 Codec's DRAM. However, during normal operating mode, it is envisioned that the modes when H_ADDR[6] is high will only be used for Bitstream transfers to the MG1264 Codec. The $\overline{\text{HCS0}}$ device is used mainly for mailbox messaging those transactions can happen on a polled IO basis.

4.7.5 Reading the MG1264 Codec's External Memory

The procedure to read a block of the MG1264 Codec's memory is:

1. Verify that the EMBusy bit in the EMStatus register is set to 0; otherwise, wait until it is.
2: If necessary, update the MG1264 Codec's DMA engine configuration in the EMConfig register.
3: Store the address to be accessed in the EMSrcAddrH and EMSrcAddrL registers.
4: Write the transfer length to the EMXferSize register.
5: Write the "read" command to the EMCmd register (set the EMCmd field to 0b01).
6: Set up the System Host CPU to DMA the data from the EMFifoRdPort to a buffer in the System Host CPU's memory
   or
   Loop through enough loads from EMFifoRdPort to read the specified number of words. You must check the EMFifoStatus in this case. Refer to "Checking the FIFO Status" on page 78 for additional information.
7: Optionally, check the EMBusy bit in the EMStatus register or use EMInt to determine when the DMA engine is finished (for a "read" operation, the DMA engine for the System Host CPU can generate an interrupt when the DMA is complete).

Writing the MG1264 Codec's External Memory

The procedure to write to a block of the MG1264 Codec's memory is:

1. Verify that the EMBusy bit in the EMStatus register is set to 0; otherwise, wait until it is.
2: If necessary, update the MG1264 Codec's DMA engine configuration in the EMConfig register.
3: Setup the address in the EMDestAddrH and EMDestAddrL registers.
4: Write the transfer length to the EMXferSize register.
5: Write the "write" command to the EMCmd register (set the EMCmd field to 0b10).
6: Set up the System Host CPU to DMA the data from a buffer in the System Host CPU's memory to the EMFifoWrPort
   or
   Loop through enough stores to EMFifoWrPort to write the specified number of words. You must check the EMFifoStatus in this case. Refer to "Checking the FIFO Status" on page 78 for additional information.
7: Optionally, check the EMBusy bit in the EMStatus register or use EMInt to determine when the DMA engine is finished (for a "write" operation, the DMA engine for the System Host CPU can generate an interrupt when the DMA is complete from the System Host CPU's point of view, but the MG1264 Codec may still be working on it).

4.7.6 Checking the FIFO Status

The interface logic asserts a DMA request to the System Host CPU (by asserting $\overline{\text{H\_DMARQ}}$) when it has available at least EMDThresh words of data in its Read FIFO or when it can accept at least EMDThresh words of data into its Write FIFO, depending upon the direction of the transfer programmed in the EMCmd register. If the System Host CPU DMA engine is not used, individual words can be read (loaded) from or written (stored) to this port, but software must check the status of the FIFO after every EMDThresh word.

4.7.7 External Memory Access Registers

These registers are used to access the external memory.

External Memory Command Register ***EM1Cmd*** ***Offset: 0x0000***
Bitstream Memory Command Register ***EM2Cmd*** ***Offset: 0x0040***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMCmd		EMM arb Priority	EM Endian Swap	Reserved											

Reserved fields should be ignored (masked) when read and only 0's should be written to them.
This register should not be modified while EMBusy is 1.

EMCmd	00 = Idle: no operation is performed 01 = Read: Initiate transfer from MG1264 Codec Memory, starting at EMSrcAddr, to the Memory Read FIFO, which can be read by the System Host CPU (Static Bus) via the EMFifoRdPort. 10 = Write: Initiate transfer from the Memory Write FIFO to MG1264 Codec Memory, starting at EMDestAddr; the Memory Write FIFO is filled by the System Host CPU (Static Bus) via the EMFifoWrPort. 11 = Reserved For all operations, the transfer length is given by EMXferSize.
EMMarbPriority	0 = set to 0 when both EM ports are expected to be simultaneously active. 1 = set to 1 for optimal transfers when only 1 of the 2 EM ports are expected to be active.
EmEndianSwap	0 = Byte order is preserved (default) 1 = Bytes 0 and 1 are swapped during the transfer.

External Memory DMA Transfer Size Register ***EM1XferSize*** ***Offset: 0x0002***
Bitstream Memory DMA Transfer Size Register ***EM2XferSize*** ***Offset: 0x0042***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMXferSize															

Reserved fields should be ignored (masked) when read and only 0's should be written to them.
This register should not be modified while EMBusy is 1

EMXferSize	Number of 32-bit data words to transfer. A zero means no words will be transferred; EM-Busy will not be set. For Frame Mode, this is interpreted as: EMYSize[5:0] = EMXferSize[15:10] - Vertical size of the block to transfer (number of "rows") EMXSize[9:0] = EMXferSize[9:0] - Horizontal size (in bytes) of the block to transfer (size of "row")

External Memory DMA Source Address High Register EM1SrcAddrH Offset: 0x0004
Bitstream Memory DMA Source Address High Register EM2SrcAddrH Offset: 0x0044

This pair of registers changes function depending on the type of operation where it is being used. During DMA Operations, these registers are interpreted as follows:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMSrcAddrH															
EMSrcAddrH	Source address for a “read” (System Host CPU <- MG1264 Codec) or “copy” (MG1264 Codec -> MG1264 Codec) operation. Used with EMSrcAddrL. This register should not be modified while the EMBusy bet is set to 1. During the operation, the hardware updates this register as it progresses.														

External Memory DMA Source Address Low Register EM1SrcAddrL Offset: 0x0006
Bitstream Memory DMA Source Address Low Register EM2SrcAddrL Offset: 0x0046

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMSrcAddrL															
EMSrcAddrL	Source address for a “read” (System Host CPU - MG1264 Codec) or “copy” (MG1264 Codec - MG1264 Codec) operation. Used with EMSrcAddrH. This register should not be modified while the EMBusy bet is set to 1. During the operation, the hardware will update this register as it progresses.														

During Frame Buffer Access (EMMode = 00 or 01), these registers are interpreted as follows:

External Memory Y Source Address Register EM1SrcYAddr Offset: 0x0004
Bitstream Memory Y Source Address Register EMSrcYAddr Offset: 0x0044

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMSrcYAddr															
EMSrcYAddr	Starting Vertical/Y source address														

External Memory X Source Address Register EM1SrcXAddr Offset: 0x0006
Bitstream Memory X Source Address Register EMSrcXAddr Offset: 0x0046

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMSrcXAddr															
EMSrcXAddr	Starting Horizontal/X source address														

External Memory DMA Destination Addr. High Register EM1DestAddrH Offset: 0x0008
Bitstream Memory DMA Destination Addr. High Register EM2DestAddrH Offset: 0x0048

This pair of registers changes function depending on the type of operation where it is being used. During DMA Operations, these registers are interpreted as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMDestAddrH															

EMDestAddrH	Destination address for a “write” (System Host CPU - MG1264 Codec) or “copy” (MG1264 Codec - MG1264 Codec) operation. Used with EMDestAddrL. This register should not be modified while the EMBusy bet is set to 1. During the operation, the hardware will update this register as it progresses.

External Memory DMA Destination Addr. Low Register EM1DestAddrL Offset: 0x000A
Bitstream Memory DMA Destination Addr. Low Register EM2DestAddrL Offset: 0x004A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMDestAddrL															

EMDestAddrL	Destination address for a “write” (System Host CPU - MG1264 Codec) or “copy” (MG1264 Codec - MG1264 Codec) operation. Used with EMDestAddrH. This register should not be modified while the EMBusy bet is set to 1. During the operation, the hardware will update this register as it progresses.

During Frame Buffer Access (EMMode=00 or 01), this register is interpreted as:

External Memory Y Destination Addr. Register EM1DestYAddr Offset: 0x0008
Bitstream Memory Y Destination Addr. Register EMDestYAddr Offset: 0x0048

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMDestYAddr															

EMDestYAddr	Starting Vertical/Y destination address

External Memory X Destination Addr. Register EM1DestXAddr Offset: 0x000A
Bitstream Memory X Destination Addr. Register EMDestXAddr Offset: 0x004A

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMDestXAddr															

EMDestXAddr	Starting Horizontal/X destination address

External Memory Status Register ***EM1Status*** ***Offset: 0x000C***
Bitstream Memory Status Register ***EM2Status*** ***Offset: 0x004C***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EM-Busy	Reserved														

Reserved fields should be ignored (masked) when read. This register is read-only.	
EMBusy	0 = No operation is in progress; other registers may be changed. 1 = A DMA operation is in progress; the EMCmdParams, EMSrcAddr, EMDestAddr, and EMConfig registers may not be changed.

External Memory Remaining Count ***EM1RemCount*** ***Offset: 0x000E***
Bitstream Memory Remaining Count ***EM2RemCount*** ***Offset: 0x004E***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMRemCount															

Reserved fields should be ignored (masked) when read. This register is read-only.	
EMRemCount	Number of 32-bit data words remaining to be transferred. In frame mode this field is interpreted similar to EMXferSize: EMRemY[5:0] = EMRemCount[15:10] - Remaining number of blocks to transfer (number of "rows") EMRemX[9:0] = EMRemCount[9:0] - Remaining number (in bytes) of block to transfer (size of "row"). This field should be an even number, i.e. EMRemX[0] always equals 0.

External Memory Configuration Register ***EM1Config*** ***Offset: 0x0010***
Bitstream Memory Configuration Register ***EM2Config*** ***Offset: 0x0050***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EM-Wait	EMDThresh				EM Burst	EMMode		EMBaseId							

Reserved fields should be ignored (masked) when read and only 0's should be written to them.	
EMWait	0 = no H_WAIT signal generated (default) 1 = H_WAIT signal is generated. EMDThresh should be set to 1 when EMWait is set to 1
EMDThresh	The interface logic asserts a DMA request to the System Host CPU (by asserting H_DMARQ) when it has available at least EMDThresh words of data in its Read FIFO or when it can accept at least EMDThresh words of data into its Write FIFO, depending upon the direction of the transfer programmed in the EMCmd register.
EMBurst	Number of 16-bit words per internal MG1264 Codec Memory burst access. A DMA operation is broken into sequential MG1264 Codec memory requests of the specified burst size. This parameter must be set to a value less than (usually half of) the MG1264 Codec MMU buffer for the System Host CPU. Code: 0 = 8 16-bit words 1 = 16 16-bit words (default) This field is not used when EMMode is set for Frame Buffer access. The entire DMA operation is sent as one internal MG1264 Codec Memory operation (using EMYSize, EMXSize, EMY*Addr, and EMX*Addr). The software must take care not to attempt a request larger than the MG1264 Codec Memory subsystem can handle (the request must be no larger than the MMU buffer size allocated to the MG1264 Codec Host Interface).
EMMode	Use EMMode to control the MG1264 Codec MMU Transaction Mode 00 = Frame Buffer - frame access 01 = Frame Buffer - field access 10 = Linear (default) 11 = reserved; do not use.
EMBaseId	EMSrcAddr and EMDestAddr specify addresses (offsets) relative to the MG1264 Codec Memory Subsystem identified by EMBaseId. (default: 0)

External Memory Access FIFO Read Port ***EM1FifoRdPort*** ***Offset: 0x0012***
Bitstream Memory Access FIFO Read Port ***EM2FifoRdPort*** ***Offset: 0x0052***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMFifoRdPort															

EMFifoRdPort	A read from this port removes and returns a 16-bit data word from the Memory Read FIFO that was read from the MG1264 Codec's memory. DO NOT WRITE TO THIS REGISTER! DATA WILL BE LOST!

External Memory Access FIFO Write Port ***EM1FifoWrPort*** ***Offset: 0x0014***
Bitstream Memory Access FIFO Write Port ***EM2FifoWrPort*** ***Offset: 0x0054***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EMFifoWrdPort															

EMFifoWrPort	16-bit data from the “Static Bus” written to this port's address is placed into the Memory Write FIFO to be sent to the MG1264 Codec's memory. Reading from this address returns 0's.

External Memory FIFO Status Port ***EM1FifoStatus*** ***Offset: 0x0016***
Bitstream Memory FIFO Status Port ***EM2FifoStatus*** ***Offset: 0x0056***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved														EM Fifo RdReq	EM Fifo WrReq

Reserved fields should be ignored (masked) when read and only 0’s should be written to them.	
EMFifoRdReq	0 = no more words are available for reading beyond the current burst of eight 1 = at least EMDThresh more 16-bit words are available in the Memory Read FIFO If the System Host CPU’s DMA engine is being used, then flow control is done by the DMA request line; in this case, it is not necessary for software to check this bit.
EMFifoWrReq	0 = no more words can be accepted beyond the current burst of eight 1 = at least EMDThresh more 16-bit words can be accepted by the Memory Write FIFO If the System Host CPU’s DMA engine is being used, then flow control is done by the DMA request line; in this case, it is not necessary for software to check this bit.

4.7.8 Bitstream Write FIFO Access Registers

The System Host CPU sends a bitstream to the MG1264 Codec's external DRAM through a set of registers. These registers are explained in detail in the sections that follow.

Bitstream FIFO Write Port ***BFifoWrPort*** ***Offset: 0x0060***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BFifoWrPort															

BFifoWrPort	16-bit data from the "Static Bus" written to this port's address is sent to the System Input Stream Controller of the Media Engine. Reading from this address returns 0's.

Bitstream FIFO Status Register ***BFifoStatus*** ***Offset: 0x0062***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															BFifo WrReq

Reserved fields should be ignored (masked) when read and only 0's should be written to them.

BFifoWrReq	0 = no more words can be accepted beyond the current burst of DBThresh 1 = at least BBurst more 16-bit words can be accepted by the Bitstream FIFO If the System Host CPU's DMA engine is being used, then flow control is done by the DMA request line; in this case, it is not necessary for software to check this bit.

Bitstream FIFO Configuration Register ***BFifoConfig*** ***Offset: 0x0064***

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved											BThresh				Res

Reserved fields should be ignored (masked) when read and only 0's should be written to them.

BThresh	When this number of 16-bit words are left in the FIFO, the DMA request signal or the BFifoWrReq bit in the BFIFOStatus register is deasserted.

The interface logic asserts the DMA request to the System Host CPU by driving $\overline{\text{H\_DMARQ}}$ high) when it can accept at least BThresh words of data into its FIFO. If the System Host CPU's DMA engine is not used, individual words can be written (stored) to this port, but software must check the status of the FIFO after every BThresh word.

Chapter 5. Video Interface

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices is able to both send and receive digitized raw video. This video can be either interlaced or “progressive”. Common resolutions are shown in Table 5-1.

Table 5-1 Input Video Resolutions

Horizontal	Vertical	Frame Rate	Description
800	600	25 fps	SVGA (square pixel)
768	576	25 fps	square pixel PAL
720	576	25 fps	rectangular pixel PAL
720	480	30 fps	rectangular pixel NTSC
640	480	30 fps	VGA (square pixel NTSC)
320	240	30 fps	QVGA

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices video interface supports both 656 video and 601 video. For 656 video, the MG1264 Codec reads the AV codes from the data stream to derive the timing, and for 601 video, the MG1264 Codec receives the sync data on the input pins.

5.1 Video Interface Usage

The pages that follow show the MG1264 Codec in various video applications.

5.1.1 Interlaced ITU-R BT.656 Video Interfaces

The MG1264 Codec has video input and output interfaces for interlaced video that are ITU-R BT.656-compliant. In NTSC interlaced mode, the video interface requires that each frame of video contain exactly 858 Horizontal samples and 525 Lines, as shown in Figure 5-1. The Horizontal blanking and Vertical blanking can be adjusted to adapt to a target resolution of active video, but the total number of samples in each frame must be maintained.

In PAL interlaced mode, the video interface requires that each frame of video contain exactly 864 Horizontal samples and 625 Lines, as shown in Figure 5-2. The Horizontal blanking and Vertical blanking can be adjusted to adapt to a target resolution of active video, but the total number of samples in each frame must be maintained.

Figure 5-1 and Figure 5-2 show the timing and blanking for conventional 656-compliant video. For both NTSC and PAL video, the Horizontal Blanking has a minimum value of 64 samples and the Vertical Blanking has a minimum value of four lines when using adjustable timing.

In interlaced applications, the video frame is created by taking a line from each of the top and bottom video fields in sequence as shown in Figure 5-1 for NTSC video and Figure 5-2 for PAL video.

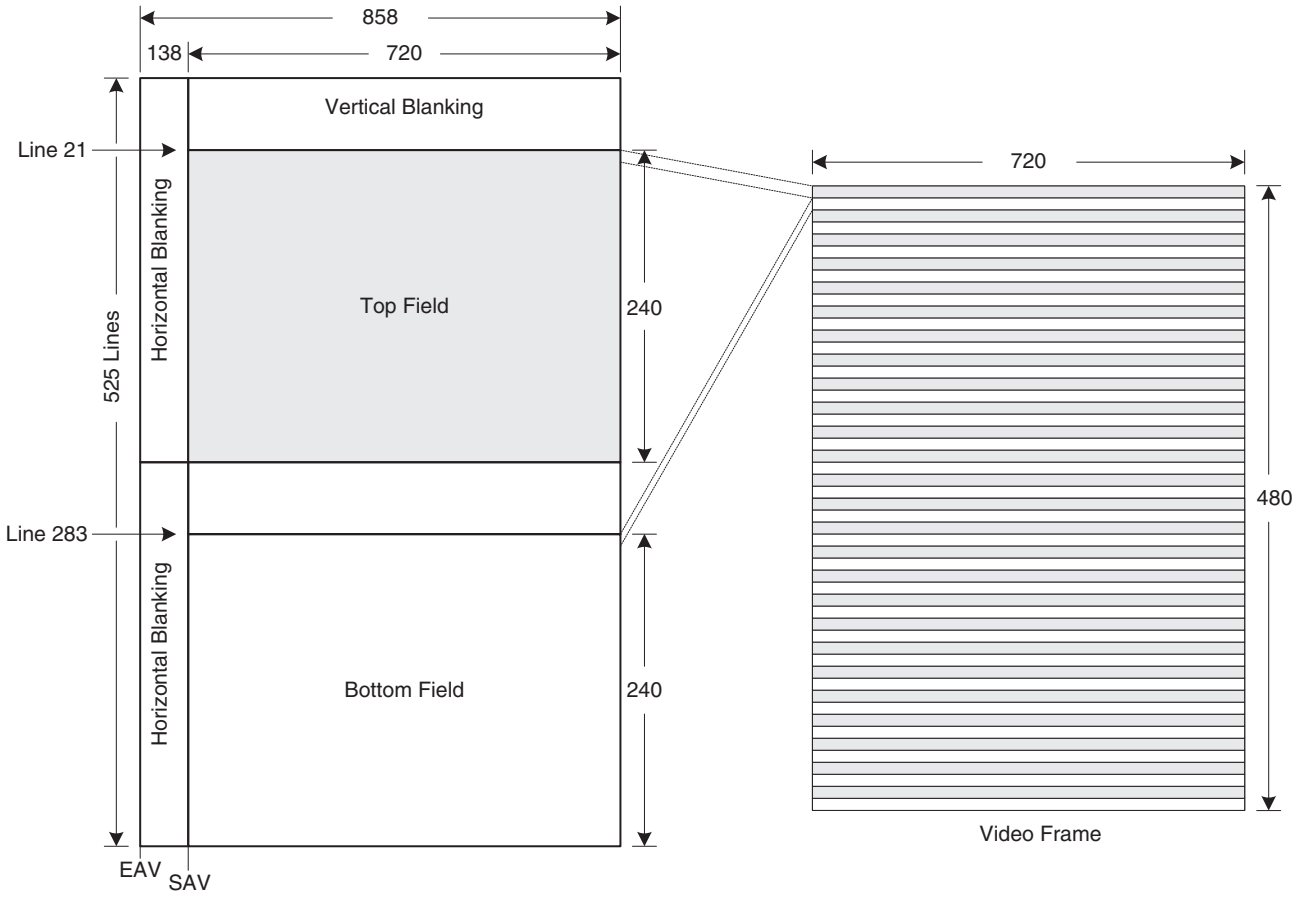


Figure 5-1 ITU-R BT.656 NTSC Interlaced Video Standard

For example:

1. Line 1 from the Top Field
2: Line 1 from the Bottom Field
3: Line 2 from the Top Field
4: Line 2 from the Bottom Field
5: Line 3 from the Top Field
6: Line 3 from the Bottom Field

. . .

479: Line 240 from the Top Field
480: Line 240 from the Bottom Field

A similar sequence is followed for PAL interlaced video, except that a greater number of lines have to be interlaced.

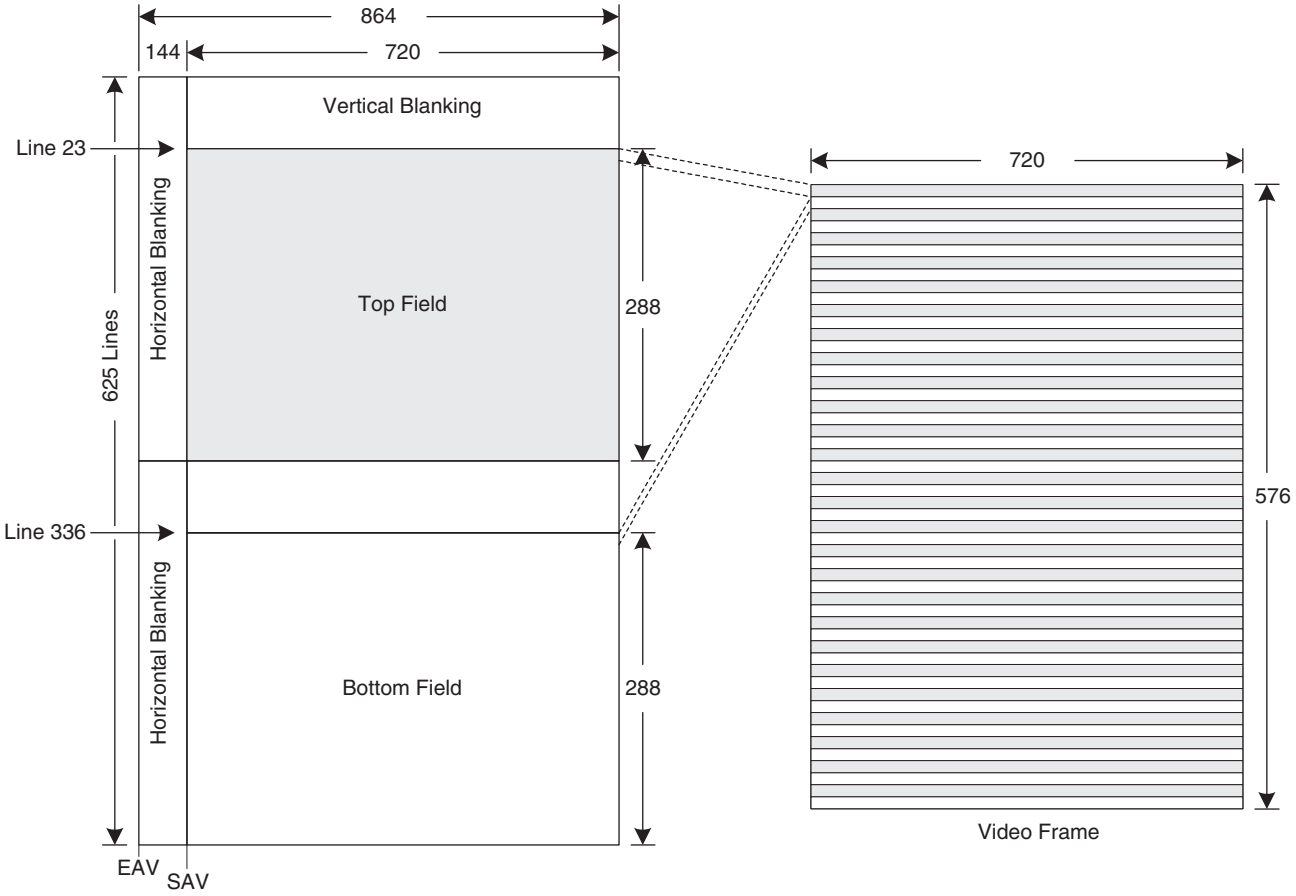


Figure 5-2 ITU-R BT.656 PAL Interlaced Video Standard

1. Line 1 from the Top Field
2: Line 1 from the Bottom Field
3: Line 2 from the Top Field

4: Line 2 from the Bottom Field
5: Line 3 from the Top Field
6: Line 3 from the Bottom Field

. . .

573: Line 287 from the Top Field
574: Line 287 from the Bottom Field
575: Line 288 from the Top Field
576: Line 288 from the Bottom Field

5.1.2 Progressive Video Interface in Free-run Mode

There is no digital transmission standard for progressive video. Because of this, the timings are adjustable as shown in Figure 5-3. This is called Free-run Mode.

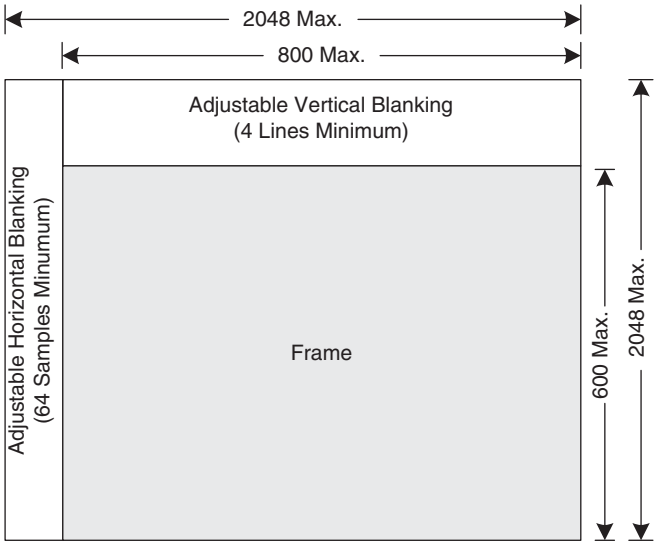


Figure 5-3 Progressive Video with Adjustable Timing

The actual parameters are set in the Firmware Configuration file. Contact the Mobilygen Field Application group for details and support in determining the appropriate values for your application.

5.2 Video Interface Signals

This section describes the signals used to interface the MG1264 Codec into a system. Table 5-2 shows the signals and Figure 5-4 shows the connections.

Table 5-2 Video Interface Signals

SIGNAL	Dir	# Bits	Description
VID_CLK	I	1	**Video Clock:** This is primarily used when the MG1264 Codec is slaved to the Video Clock. Optionally, the MG1264 Codec can master the Video Clock.
VID_DATA_[7:0]	IO	8	**Video Data:** This bidirectional bus is an input by default. It must be configured in software to be used as an output. Contact Mobilygen Technical Support for information.
VIDOUT_DATA_[7:0]	O	8	**Video Output Data:** Data is output on this bus when the MG1264 Codec is sourcing the video data (decoding). During full duplex operation, the bidirectional Video Data port is the input, and the Video Output Data is the output.

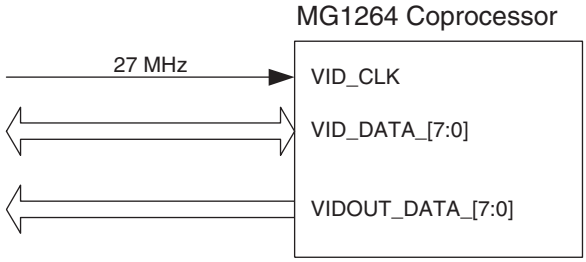


Figure 5-4 Video Interface Connections

5.3 Video Interface Timing

The video interface is 656 in nature, and the signal pins consist of a video clock (VID_CLK) and video data (VID_DATA_[7:0]) as shown in Figure 5-5. The data is either the timing code (EAV/SAV) or the actual video data. The timing for the interface is specified in the 656 Interface Specification.

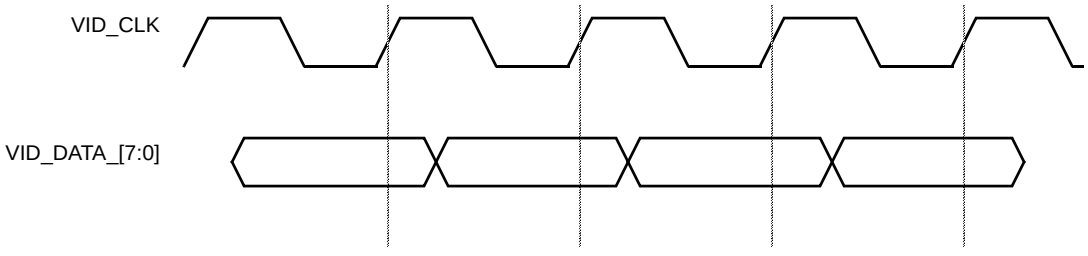


Figure 5-5 Video Interface Timing

5.4 Working With CMOS Sensors

The MG1264 Codec's VID_DATA port is a bidirectional ITU-R.BT656 style interface. It is designed to be flexible and interface to any device that implements the 656 standard. The VID_DATA port can support clock speeds other than 27 MHz up to 40 MHz.

Some CMOS sensors output ITU-R.BT656 or 601 signals directly (known as YUV sensors vs. RGB Bayer sensors). For the MG1264 Codec's VID_DATA port to operate correctly, the video source must provide active Horizontal and Vertical blanking signals, even for non-active video data. Some CMOS sensors are known to suppress blanking signals in non-active video regions.

Table 5-5 shows a list of CMOS sensors that are known to work with MG1264:

Table 5-3 Compatible CMOS Sensors

Company	Part Number	URL
Micron	MT9V111	http://download.micron.com/pdf/flyers/mt9v111_(mi-soc-0360)_mobile_flyer.pdf
ST	VS6524	http://www.st.com/stonline/books/ascii/docs/11157.htm
OnmiVision	OV7710	http://www.ovt.com/data/parts/pdf/web_Brief7710%20security%20V2.8.pdf
OnmiVision	OV7720	http://www.ovt.com/products/app2_table.asp?id=9

Because there is a great deal of variance between different sensors with respect to video clock gating, compliance, etc., we strongly recommend that you contact Mobilygen Technical Support before starting a design that includes a CMOS sensor.

5.5 Video Pre-Processing Filters

The MG1264 Codec has four specific video pre-processing filters that can be enabled of disabled to improve the encoded picture quality of source video.

5.5.1 Vertical Impulse Noise Reduction

The Vertical impulse Noise Reduction filter is a three-line adaptive median filter that reduces the presence of horizontal line streaks and line drops. This filter should be used only under extremely noisy conditions because it can generate non-linear artifacts.

5.5.2 Horizontal Impulse Noise Reduction

The Horizontal Impulse Noise Reduction is a three-tap adaptive median filter that reduces the presence of salt-and-pepper (Gaussian) noise and random single stuck-on pixels.

5.5.3 Horizontal Edge-Preserving Noise Reduction Filter

The Horizontal Edge-Preserving Noise Reduction filter reduces high frequency noise while preserving edges and high contrast picture details. The amount of high frequency filtered is determined by a programmable 7-tap FIR symmetrical filter. The types of edges preserved are determined by a set of edge transition thresholds.

5.5.4 Motion Adaptive Temporal Recursive Filter

The Motion Adaptive Temporal Recursive Filter reduces picture noise according to the amount of motion detected in a neighborhood of pixels around every pixel in the picture. When pixels belong to still areas of the picture, they are strongly filtered recursively across many frames, i.e., with a long temporal constant. Conversely, pixels belonging to areas of the picture with motion are lightly filtered with a short temporal constant.

Chapter 6. SDRAM Interface

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices requires one 8 Meg x 16 SDRAM, and supports both regular SDRAMs with a 3.3V interface or Mobile SDRAMs with a 2.5V interface. We believe that most customers will use Mobile SDRAM because they are packaged in a fine-pitched VFBGA package suitable for mobile designs. Another reason is that an equivalent 3.3V Mobile SDRAM draws less power than an equivalent 3.3V normal SDRAM.

The option of 2.5V volt support is very important to some customers. It offers tremendous system power savings. In the Field Encode mode, the saving are >100 mW, including the MG1264 Codec DRAM IO and the DRAM part itself.

6.1 The SDRAM Interface

The MG1264 Codec connects to the SDRAM as shown in Figure 6-1. Table 6-1 lists the connections and describes their functions.

Table 6-1 DRAM Interface Signal List

SIGNAL	Dir	# Bits	Description
SD_CLK	O	1	SDRAM Clock. This signal provides the clock to the SDRAM.
SD_DQ_[15:0]	IO	16	SDRAM Data. These signals are the 16-bit data port between the SDRAM and the MG1264 Codec.
SD_A_[12:0]	O	13	SDRAM Address. This bus provides the multiplexed row and column address information to the SDRAM.
SD_BA_[1:0]	O	2	SDRAM Bank Address. These lines select the bank that is being addressed within the DRAM.
SD_DQM_[1:0]	O	2	SDRAM Data Mask. These bits provide a byte-mask signal for data being written to the DDR SDRAM. Two MDQM bits are provided to mask the lower and upper bytes of 16-bit wide SDRAMs. In a typical system SD_DQM[0] is connected to LDQM and SD_DQM[1] is connected to UDQM on 16-bit wide SDRAMs.

Table 6-1 DRAM Interface Signal List

SIGNAL	Dir	# Bits	Description
SD_CKE	O	1	SDRAM Clock Enable. This signal is the Clock Enable Output for the DRAMs.
$\overline{\text{SD\_CS}}$	O	1	SDRAM Chip Select
$\overline{\text{SD\_RAS}}$	O	1	SDRAM RAS. This signal is the row access strobe to the SDRAM.
$\overline{\text{SD\_CAS}}$	O	1	SDRAM CAS. This signal is the column access strobe to the SDRAM.
$\overline{\text{SD\_WE}}$	O	1	SDRAM Write Enable

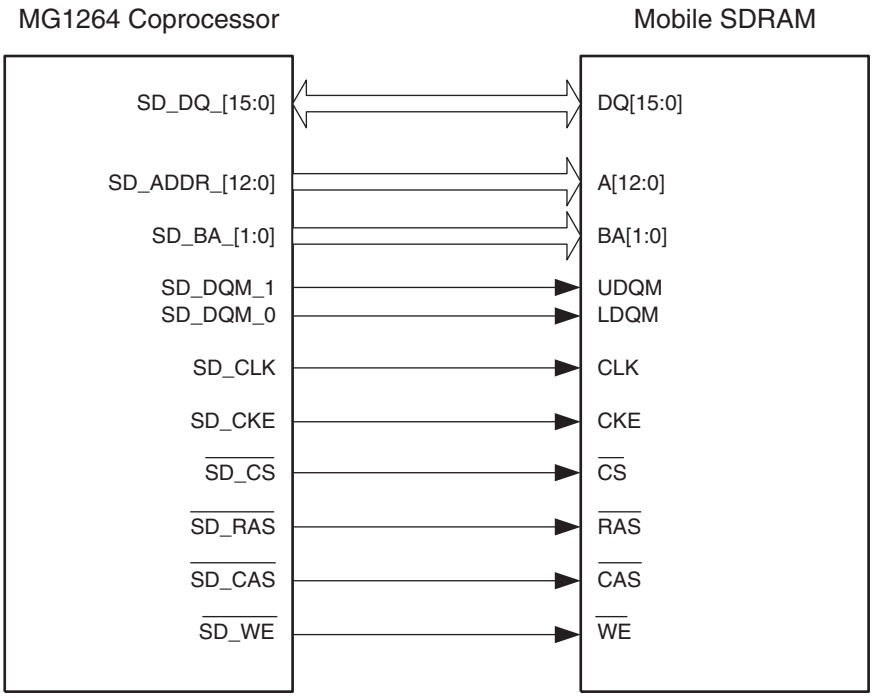


Figure 6-1 MG1264 Codec SDRAM Interface

6.2 Mobile SDRAM Features

Features that are implemented in the Mobile SDRAM that are not in the normal SDRAM include:

- Support for 3.3 and 2.5 Volt Operation (Core and I/O)
- Temperature Compensated Self-Refresh
- Partial Array Self Refresh
- Deep Power Down
- Drive Strength Control

6.2.1 Voltage Operation (3.3V and 2.5V)

The main benefit that the MG1264 Codec will get from the Mobile SDRAM is low-voltage operation. While Normal SDRAMs are limited to 3.3V, Mobile SDRAMs allow for the option of supporting 2.5V as well. The MG1264 Codec supports both the 3.3V and 2.5V options.

6.2.2 Temperature Compensated Self-Refresh

Mobile SDRAMs have a mechanism for saving self-refresh power based upon the operating temperature. The Controller enables this mechanism by programming the External Mode Register (EMR) bits A4 and A3. The Controller must have an external temperature sensor to know the value to program into the EMR.

6.2.3 Deep Power Down

The MG1264 Codec does not use a DPD mode. Instead, the MG1264 Codec uses an external Voltage Regulator to switch the power completely off to the SDRAM.

6.2.4 Drive Strength Control

Mobile SDRAMs are typically designed assuming a 30 pF load with a risetime and/or falltime target of 1 nS. However, two bits exist within the Extended Mode Register of the DRAM that allow for control of the Drive Strength (DS) to tailor it to lower loading scenarios.

Chapter 7. Audio Interface

7.1 Audio Interface Overview

The audio interface on the MG1264 Codec is responsible for receiving a PCM audio stream from an audio Analog-to Digital convertor in either left-justified mode or as an I^2S audio Slave device. It then writes the audio samples to the external memory via the memory subsystem. This module can support one or two channels (left and right) per sample.

The MG1264 Codec accepts input audio for AAC compression and generates output audio from decompressed AAC bitstreams. It accepts audio sample rates (fs or AUD_LRCK) of 48, 44.1, 32, 24, and 22.05 kHz.

The MG1264 Codec encodes two-channel AAC audio encoding with 16-bit samples at both the 32 kHz and 48 kHz sample rates. The target audio bitrate is 10% of the associated video bitrate, with an appropriate sample rate.

User Control of the AAC Encoder Features

The audio encoder features are selectable. Each feature has settings and/or ranges that affect the overall compression efficiency accordingly. Table 7-1 shows the key features and their associated target settings.

Table 7-1 AAC Encoder Features

Feature	Options
Channels	Mono (1) or Stereo (2)
Sample rate	22.05, 24, 32, 44.1, or 48 kHz
Bitrate	8 kbps - 384 kbps

7.2 Audio Interface Signals

The audio interface is a modification of the inter-IC sound (I^2S) bus; a serial link especially for digital audio. To minimize the number of pins required and to keep wiring simple, a four-line serial bus is used. The signals consist of an input for two time-multiplexed data channels, an output for two time-multiplexed data channels, a word select line, and a clock line. These signals are shown in Table 7-2.

Table 7-2 Audio Interface Signal List

SIGNAL	Dir	# Bits	Description
AUD_CLK	I	1	Audio Oversample Clock, 256 fs (LRCK) [1]
AUD_BCK	IO	1	Audio Bit Clock, 32 or 64 fs (LRCK) [2]
AUD_LRCK	IO	1	Audio Left/Right Clock (48, 44.1, 32, 24, 22.05 kHz) [2]
AUD_IDAT	I	1	Audio Serial Input Data [1]
AUD_ODAT	O	1	Audio Serial Output Data [3]

1.This signal should be pulled down if not used.
2.This pin should be configured in software as an output and left unconnected if not used.
3.This pin should be left unconnected if not used.

The MG1264 Codec requires that the audio clock must be supplied from an external source (the MG1264 Codec is an audio Slave). The clocks can be supplied by either the System Host CPU (refer to Figure 7-1) or the audio DAC/ADC (refer to Figure 7-2). The MG1264 Codec can use the AUD_LRCK and AUD_BCK signals acting as either a slave or a master.

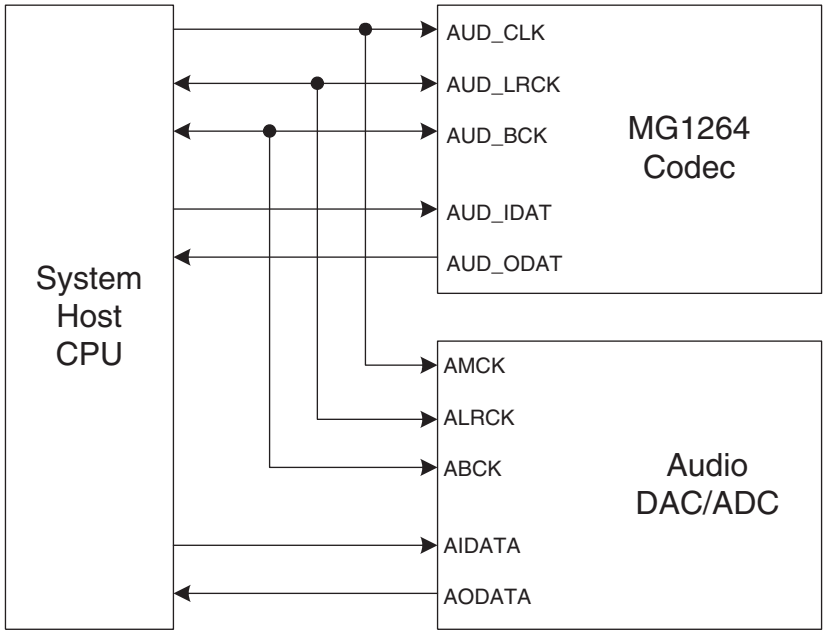


Figure 7-1 Audio Interface with the System Host CPU as the Audio Clock Master

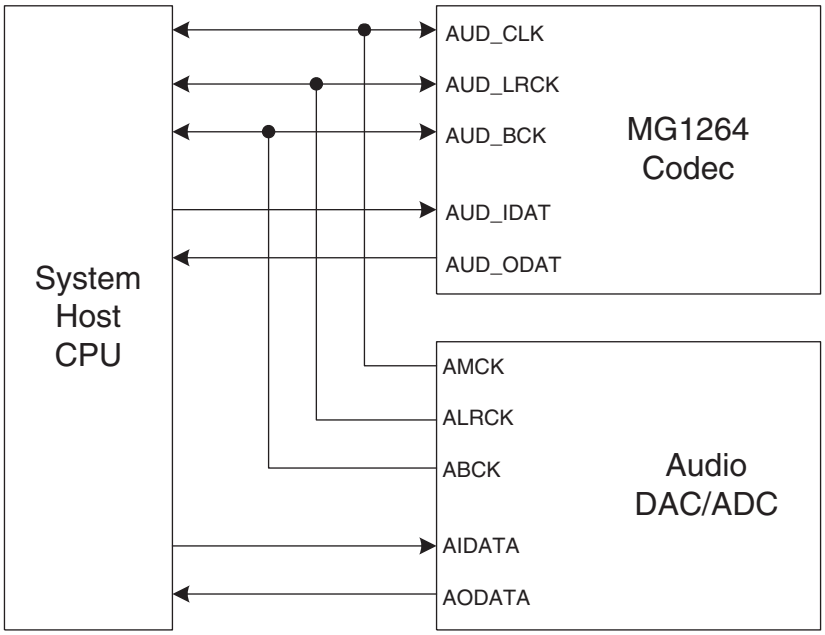


Figure 7-2 Audio Interface Connections with the DAC/ADC as the Audio Clock Master

7.3 I²S Audio Waveforms

A sample waveform for I²S audio is shown in Figure 7-3. Note that AUD_LRCK (Left Right Clock) changes one clock before the MSB is transmitted. This allows the slave transmitter to derive synchronous timing for the serial data that will be set up for transmission. It also allows the receiver to store the previous word and clear the input for the next word.

- LRCK = 0; channel 0 (left)
- LRCK = 1; channel 1 (right)

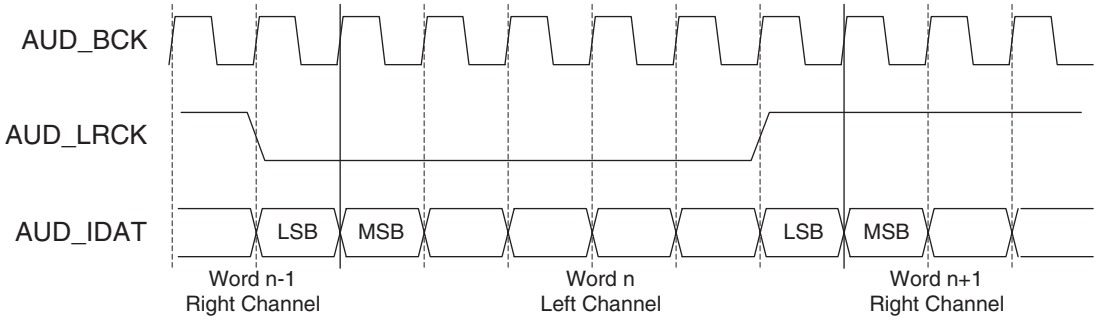


Figure 7-3 I²S Left-justified Audio Waveform

7.4 Left Justified Audio Waveform

A sample waveform for Left Justified audio shown in Figure 7-4. Note that AUD_LRCK (Left Right Clock) changes on the same cycle as when the MSB is transmitted.

- LRCK = 1; channel 0 (left)
- LRCK = 0; channel 1 (right)

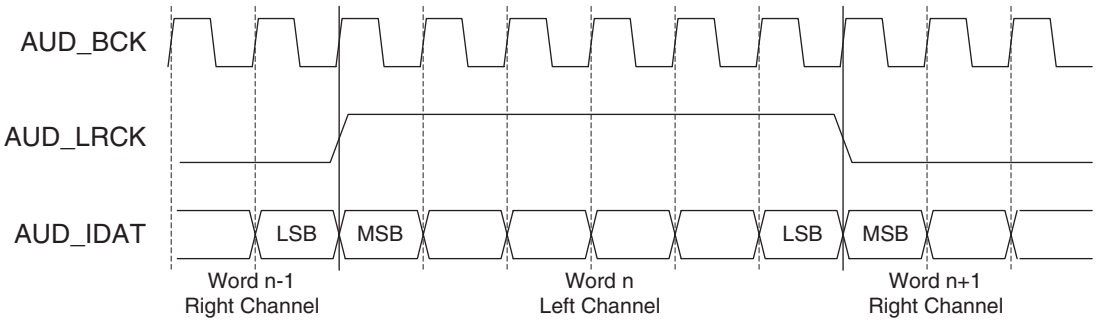


Figure 7-4 Left-justified Audio Waveform

7.5 16, 20, 24, 32-Bit Left Justified Audio Waveform

Sample waveforms for 16, 20, 24, and 32-bit Left Justified audio are shown in Figure 7-5. Note that AUD_LRCK stays high/low for 32 cycles and AUD_CLK is 64 cycles per channel. The MSB for each audio sample is aligned with the AUD_LRCK's transition. The Audio Input Interface ignores the data bus after the LSB for each sample.

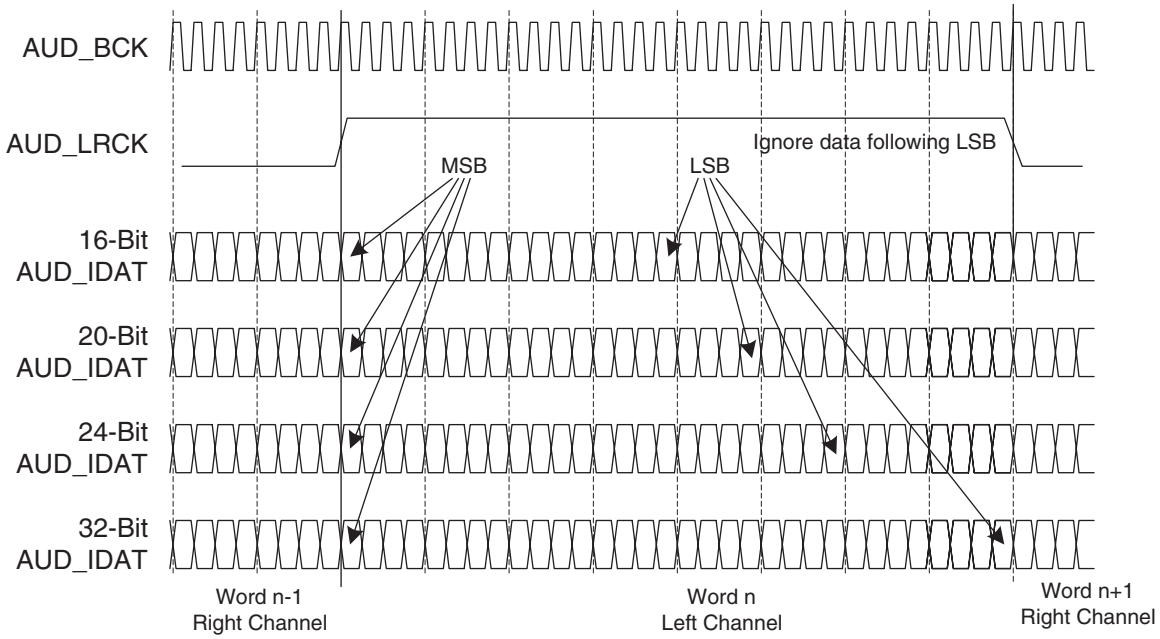


Figure 7-5 16, 20, 24, and 32-Bit Left Justified Audio Waveform

Chapter 8. Bringing up the MG1264 Codec

This chapter provides suggestions for bringing up the MG1264 Low Power H.264 and AAC Codec for Mobile Devices decoder and encoder functions for the first time.

8.1 Decoder Bringup

This section describes the phases needed to bring up the AVC decoder in the MG1264 Codec. The phases are as follows.

1. Send a video elementary bitstream to the decoder that is smaller than the decoder's bitbuffer and confirm that it decodes.
2: Send a video elementary bitstream to the decoder that is larger than the decoder's bitbuffer and confirm it decodes. Since the stream is larger than the bitbuffer, this phase tests the software flow control.
3: Send a “QBOX” video stream to the decoder and confirm that it decodes. A QBOX video stream is a video elementary stream that has a Mobilygen QBOX header prior to each video access unit. More information about the QBOX is contained “Phase 3: Decoding A QBOX Stream” on page 110.

8.1.1 Phase 1: Decoding a Small Elementary NAL Video Stream

The goal for this step is to decode a video elementary AVC stream that is smaller than the MG1264 Codec bitbuffer.

Step 1: Configuring the Bitstream Type

The MG1264 Codec firmware can decode several bitstream formats called BitstreamTypes. In this part of the bringup we will be using the “video elementary stream.” This type of stream corresponds to Annex B of the ISO/IEC 14496-10 where there is a startcode preceding each Network Abstraction Layer (NAL) unit. The size of each NAL unit is not located in the stream and can only be detected by searching for startcodes. Streams encoded by the MG1264 Codec will have a 32-bit startcode of 0x00000001, although the decoder can also handle 24 bit startcodes of 0x000001.

The default bitstream type for the MG1264 Codec firmware is the video elementary stream. This bitstream type can be forcibly selected by sending a configuration command to the

video decoder control object. This is done with the following command, which is only valid when the decoder is in IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_TYPE_ELEM_VIDEO;
cmd.arguments[2] = 0;
```

Step 2: Configuring the Bitstream Source

The MG1264 Codec firmware can receive bitstream data using three different methods. These methods are:

- Bitstream push using hardware flow control
- Bitstream pull using software flow control
- Memory pull using software flow control.

The bitstream push method sends data to the bitstream FIFO device in the MG1264 Codec host interface. This FIFO is internally connected to a MG1264 Codec device called the System Input Stream Controller (SISC). This datapath has complete hardware flow control in that, if the internal bitstream buffer is full, the bitstream FIFO on the host interface will assert the WAIT signal (or de-assert the $\overline{\text{H\_DMARQ}}$ signal) indicating to the host that no more data can be sent.

In normal playback operation the bitstream buffer will almost always be full, meaning that the WAIT signal will be asserted for up to 20 ms. until a video frame is decoded. When the decoder is in the PAUSE state, the WAIT signal will be continuously asserted. If the host system architecture has a DMA engine that is not shared with other applications and can be blocked for an indefinite period of time, then this is the best option as it requires no software interaction for flow control.

The bitstream pull method also sends data to the bitstream FIFO in the host interface, except that the host is required to send a command to request the size of data that can be safely sent without filling the bitbuffer. If the host sends less than this amount, then the WAIT signal will never be asserted for long periods of time (or indefinitely in the case of the pause state).

The memory pull interface is not covered in this document, as either the bitstream push or pull methods are sufficient for this application.

The bitstream source is set to bitstream push by default. The bitstream source can be forcibly selected with the following configuration command, which is only valid when the decoder is in the IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_SOURCE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_SOURCE_SISC_PUSH;
cmd.arguments[2] = 0;
```

For this phase of the bringup we will use the SISC_PUSH method because the size of the bitstream will be smaller than the bitbuffer.

Step 3: Putting the Decoder into the PLAY State

The decoder must be placed into the PLAY state before any streaming is done. The host must ensure that the PLAY command returns with the COMMAND_DONE interrupt before streaming otherwise some data at the start of the stream could be lost.

The decoder is put into the PLAY state with the following command.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_TYPE_ELEM_VIDEO;
cmd.arguments[2] = 0;
```

Step 4: Streaming the Bitstream

Sending the bitstream is done using the QHAL bitstream (bs) module. Because the bitstream contains startcodes and there is no parsing or demultiplexing required on the host, the host can simply read the bitstream in fixed sized blocks and send them to the host interface one at a time. The only restriction is that the transfer size must be 4-byte aligned.

Here is sample code that can be used to send data.

```
#include <stdio.h>
#include <errno.h>
#include "qhal_bs.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define NDATAPERTX (256*1024) // transfer in 256k byte chunks

char buf[NDATAPERTX];

int main(int argc,char *argv[])
{   int fd;
    qhalbs_handle_t handle;
    int err,ntx;

    switch(argc)
    {   case 1:
            fd=0;
            break;
        case 2:
            fd=open(argv[1],O_RDONLY);
            break;
        default:
            fprintf(stderr,"Error: too many arguments, syntax is %s
[<file>]\n",argv[0]);
            return -1;
    };
    if(fd<0)
    {   perror("Error");
        return errno;
```

```
    };
    handle=qhalbs_open();
    while(1)
    {   ntx=read(fd,buf,NDATAPERTX);
        if(ntx==0) break;
        if(ntx<0)
        {   perror("Error");
            return errno;
        };
        if((ntx%4) && (ntx>4))
        {   lseek(fd,-(ntx%4),SEEK_CUR);
            ntx-=ntx%4;
        } else if(ntx%4)
        {   bzero(buf+ntx,4-ntx%4);
            ntx+=4-ntx%4;
        };
        if((err=qhalbs_write(handle,buf,ntx))<0)
        {   fprintf(stderr,"Error: qhal returned error %d\n",err);
            return err;
        };
    };
}
```

Decoding and presentation should begin shortly after streaming has started.

Note that this code adds padding to the buffer if it is not a multiple of four bytes. It relies on the fact that this will only happen at the end of the file, since the read function always returns the number of bytes requested if there are that many left (or more) in the file. Also, this code has no checks for flow control. This is added in the next phase.

It is important to understand the endian-ness of the AVC bitstream and how it affects streaming. The AVC stream is big-endian and should be read as a byte stream into an internal buffer and then sent to MG1264 Codec. Little endian hosts need to be aware of this and not swap bytes when reading into the internal buffer.

8.1.2 Phase 2: Decoding a Large Elementary NAL Video Stream with Software Flow Control

The goal for this phase is to decode a bitstream that is larger than the size of the internal bit buffer. If the host can use the PUSH method, then sending a large file is exactly the same as sending a small one because the hardware takes care of the flow control. The data streaming code from the previous section continues to work as the `qhalbs_write` function will block until the streaming operation is complete. Assuming that streaming is done in a separate thread, then the system will continue to run.

If the host uses the PULL method, meaning that it cannot have the DMA operations stall for indefinite periods of time, then the following steps should be followed. The key section is in streaming where we introduce software flow control.

Step 1: Setting the Bitstream Type

This step is the same as "Step 1: Setting the Bitstream Type" on page 107.

Step 2: Configuring the Bitstream Source

We have to set the bitstream source to PULL because of the software flow control. This is done using the following configure command, which is only valid when the decoder is in the IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_SOURCE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_SOURCE_SISC_PULL;
cmd.arguments[2] = 0;
```

Step 3: Putting the Decoder into the PLAY State

This step is the same as "Step 3: Putting the Decoder into the PLAY State" on page 105.

Step 4: Streaming the Bitstream

Software flow control is achieved by sending a command to MG1264 Codec that returns the number of bytes remaining in the bit buffer. The host must ensure that it does not send more than this amount of data before it asks again how much data is available. The command to obtain how much data remains is shown here.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_AVD_CMD_NEXT_BS_SIZE;
```

The MG1264 Codec firmware returns the number of bytes free in the return values section of the command.

```
cmd.returnValues[0];
```

Here is sample code that can be used to send data. The code reads the amount of space left in the bit buffer and continuously transfers data in blocks until it has no space left. It then re-reads the amount of space left and waits until the space left is greater than the block size.

```
#include <stdio.h>
#include <errno.h>
#include "qhal_bs.h"
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#define NDATAPERTX (256*1024)

char buf[NDATAPERTX];

int main(int argc,char *argv[])
{   int fd;
    qhalbs_handle_t handle;
    int err,ntx;
    int i;
    int space;
    int pendingXfer;

    switch(argc)
    {   case 1:
            fd=0;
            break;
        case 2:
            fd=open(argv[1],O_RDONLY);
            break;
        default:
            fprintf(stderr,"Error: too many arguments, syntax is %s
[<file>]\n",argv[0]);
            return -1;
    };
    if(fd<0)
    {   perror("Error");
        return errno;
    };
    handle=qhalbs_open();

    // initialization
    pendingXfer = 0;
    ntx = 1;

    while(ntx != 0)
    {

        space = readnumleft();  // - host implements command to read data left

        while (ntx != 0)
        {

            // read one buffer
            if (pendingXfer == 0)
            {
                ntx=read(fd,buf,NDATAPERTX);
            }
```

```
            if (ntx+4 > space)
            {
                pendingXfer = 1;
                break;
            }

            if (ntx != 0)
            {
                if((ntx%4) && (ntx>4))
                {
                    lseek(fd,-(ntx%4),SEEK_CUR);
                    ntx-=ntx%4;
                }
                else if(ntx%4)
                {
                    bzero(buf+ntx,4-ntx%4);
                    ntx+=4-ntx%4;
                }
            }

            if((err=qhalbs_write(handle,buf,ntx))<0)
            {
                fprintf(stderr,"Error: qhal returned error %d\n",err);
                return err;
            }

            space -= ntx;
            pendingXfer = 0;
        }

        // sleep 15 ms
        sleep(); // -- host specific
    }

}
```

8.1.3 Phase 3: Decoding A QBOX Stream

A QBOX is a Mobilygen proprietary header that includes information about the data it contains, specifically audio or video compressed streams. For example, a flag in the header indicates if the contained data is audio or video data. It is expected that if the host does MP4 multiplexing and demultiplexing then it will stream QBOX data to the MG1264 Codec for decoding.

The QBOX header is as follows.

```
typedef struct {
     uint32_t box_size;
     uint32_t box_type; // "qbox"
     uint32_t box_flags; // (version << 24 | box_flags)
     uint16_t sample_stream_type;
     uint16_t sample_stream_id;
     uint32_t sample_flags;
     uint32_t sample_cts;
     uint8_t sample_data[];
} QBox;
```

`sample_stream_type` is set to 0x0001 for AAC audio, and 0x0002 for AVC video.

`sample_stream_id` is currently set to the same value as `sample_stream_type`.

`box_flags` has two flags. Bit 0 is set if there is sample data after the header and bit 1 is set if this is the last sample in the stream.

`sample_flags` is a 32-bit value. Bit 0 is set if the data contains configuration information for the decoder. Bit 1 is set if the CTS field is present and valid. Bit 2 is set if the video frame is a synchronization point (meaning I frame for H.264), and bit 3 is set if the frame is disposable (meaning a B frame in H.264). Bit 4 is set if the audio or video sample is the result of a MUTE command sent to the AV encoder. Bits 30-31 represent the number of leading padding bytes in the QBox (0-3) that are skipped by the MG1264 Codec demultiplexer.

This 24-byte structure is at the start of each bitstream block when the system has the stream type of QBOX. Additionally, when in QBOX mode, startcodes are not used and instead the AVC bitstream follows part 15 of ISO/IEC-14496 (AVC File Format). The net effect of this mode compared to the previous mode is that the length of the following NAL unit replaces the 4-byte start code of 0x00000001.

The first QBOX sent by the MG1264 Codec when encoding, and the first QBOX that is expected to be received when decoding, contains two NAL units, one with the sequence parameter set and the other with the picture parameter set. Subsequent QBOX's contain one NAL unit with a single AVC access unit.

For example, here is the first QBOX header of AVC video:

```
0000002D  Size of QBOX is 2D bytes including the size field.
71626F78  "qbox" in ASCII
00000001  Sample data is present
00020002  AVC video
00000000  sample flags
00000000  sample CTS (not implemented yet)
```

The next data set is the sequence parameter set proceeded by the NAL unit size. For example:

```
00000009   NAL size (not including this field)
6742E01E   Sequence parameter data
```

```
DA02D0F4   Sequence parameter data
40         Sequence parameter data
00000004   NAL size
68CE3E80   Picture parameter data
```

Totalling all of the data bytes gives 0x2D which is the size of the QBOX given at the beginning.

Step 1: Setting the Bitstream Type

This step is the same as “Step 1: Setting the Bitstream Type” on page 107.

The default bitstream type for the MG1264 Codec firmware is the video elementary stream. In order to use QBOX we must switch the type to QBOX. This must be done only once for the decoder at startup (it must be done for the encoder at startup as well).

This is done with the following command, which is only valid when the decoder is in IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_TYPE_QBOX;
cmd.arguments[2] = 0;
```

Step 2: Configuring the Bitstream Source

There are no additional requirements that QBOX streaming put on the bitstream source. If the host is using PUSH, then push should be used here; if the host is using PULL then it should be used here as well.

Step 3: Putting the Decoder into the PLAY State

This step is the same as “Step 3: Putting the Decoder into the PLAY State” on page 107.

Step 4: Streaming the Bitstream

If the stored bitstream consists of QBOXes, then the streaming is done exactly the same as in the previous phases. A QBOX stream is available to test this mode. Contact your Mobilygen sales representative for a copy.

However, it is likely that the bitstream will be stored in an MP4 file, and the host must convert it to QBOX format on the fly. This operation is quite simple and involves prepending the 24-byte QBOX header to the bitstream data (and possibly updating the size of the NAL unit as well).

8.2 Encoder Bringup

This section describes the phases needed to bringup the AVC encoder in the MG1264 Codec. The phases are as follows.

1. Record a video elementary bitstream which is smaller than the encoder's bitbuffer and confirm that it decodes.
2: Record a video elementary bitstream which is larger than the encoder's bitbuffer and confirm it decodes. Since the stream is larger than the bitbuffer this tests the software flow control.
3: Record a "QBOX" video stream and confirm it decodes. A qbox video stream is a video elementary stream that has a Mobilygen QBOX header prior to each video access unit. More information about the QBOX is contained in this document.

8.2.1 Phase 1: Recording a Small Elementary NAL Video Stream

The goal for this step is the decoding of a video elementary AVC stream that is smaller than the MG1264 Codec bitbuffer.

Step 1: Configuring the Bitstream Type

The MG1264 Codec firmware can decode several bitstream formats called BitstreamTypes. In this part of the bringup we will be using the “video elementary stream.” This type of stream corresponds to Annex B of the ISO/IEC 14496-10 where there is a startcode preceding each Network Abstraction Layer (NAL) unit. The size of each NAL unit is not located in the stream and can only be detected by searching for startcodes. Streams encoded by the MG1264 Codec will have a 32-bit startcode of 0x00000001, although the decoder can also handle 24-bit startcodes of 0x000001.

The default bitstream type for the MG1264 Codec firmware is the video elementary stream. This bitstream type can be forcibly selected by sending a configuration command to the video encoder control object. This is done with the following command, which is only valid when the encoder is in IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVE_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVE_CFP_BITSTREAM_TYPE_ELEM_VIDEO;
cmd.arguments[2] = 0;
```

Step 2: Subscribing to the BITSTREAM_BLOCK_READY Event

The MG1264 Codec firmware sends BITSTREAM_BLOCK_READY events to the host to indicate that there is new data to store. These events must first be subscribed. This subscription must be done only once at startup.

Subscription is done through the following command.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_SUBSCRIBE_EVENT;
cmd.arguments[0] = Q_AVE_EV_BITSTREAM_BLOCK_READY
cmd.arguments[2] = 0;
```

Step 3: Putting the Encoder into the RECORD state

The encoder must be placed into the RECORD state. The encoder is put into the RECORD state with the following command.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVE_CMD_OPCODE_RECORD;
cmd.arguments[0] = 0;
```

Step 4: Receiving the Bitstream

Receiving the bitstream is done by processing the bitstream block ready events. The AV encoder generates bitstream block ready events each time enough data has been accumulated in its internal bit buffers.

The structure of a generic event is as follows:

```
typedef struct
{
    CONTROLOBJECT_ID    controlObjectId;
    EVENT_ID            eventId;
    unsigned int        timestamp;
    unsigned int        payload[MAX_EVENT_PAYLOAD];
} EVENT;
```

The timestamp field is measured in microseconds. The timestamp corresponds to the PTS of the access unit in the event (if an access unit is present).

The bitstream block ready has specific meanings assigned to the payload fields. Up to six blocks of data can be sent in a single event. The structure of the bitstream block ready events follows.

```
typedef struct
{
    CONTROLOBJECT_ID    controlObjectId;
    EVENT_ID            eventId;
    unsigned int        timestamp;
    unsigned int        numAndType;
    unsigned int        reserved0;
    unsigned int        reserved1;
    unsigned int        Addr0;
    unsigned int        Size0;
    unsigned int        Addr1;
    unsigned int        Size1;
    unsigned int        Addr2;
    unsigned int        Size2;
    unsigned int        Addr3;
    unsigned int        Size3;
    unsigned int        Addr4;
    unsigned int        Size4;
} STRUCT_Q_AVE_EV_BITSTREAM_BLOCK_READY;
```

The field `numAndType` contains information about the data in the event. The lower 16-bits of this field contains the number of data blocks, which will be either 1 - 5. The upper 16-bits contains one 3-bit field per access unit that describes its content. Access unit 0's information is stored in bits 16-18, access unit 1 in 19-21 etc. The following values are currently allocated:

1: AVC Video Elementary Stream

2: QBox

In this phase, the encoder is creating AVC video elementary streams, so the value of this field will be (for example, if five blocks are sent per event) 0x12490005.

The bitstream should be read using the `qhalem_read_bytes()` method using a block Id of 64 with the address and data from the event.

Because the bitstream blocks are not being acknowledged by the host, the bitstream events will stop arriving once the video bit buffer is full.

Step 5: Decoding the Bitstream

Once stored, this bitstream should decode. Follow the steps in the decoder bringup of small video elementary streams to check.

8.2.2 Phase 2: Recording a Large Elementary NAL Video Stream with Software Flow Control

The goal for this phase is to record a bitstream that is larger than the size of the internal bit buffer. This is done by the host acknowledging buffers that it has read from, and that can be reused by the encoder.

Step 1: Configuring the Bitstream Type

This step is the same as “Step 1: Configuring the Bitstream Type” on page 112.

Step 2: Putting the Encoder into the RECORD State

This step is the same as “Step 3: Putting the Encoder into the RECORD state” on page 113.

Step 3: Receiving the bitstream

Software flow control is achieved by having the host send a command to the MG1264 Codec that contains the same information as the event it just processed. That is, once the host has read all the data that the event contains (one to six data blocks), then it sends the `BITSTREAM_BLOCK_DONE` command. Note that since the maximum number of arguments in a command is six, the host might have to send two commands. The list of blocks that are acknowledged is done by setting the address to zero.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVD_CMD_BITSTREAM_BLOCK_DONE;
cmd.arguments[0] = Addr0;
cmd.arguments[1] = Size0;
cmd.arguments[2] = Addr1;
cmd.arguments[3] = Size1;
cmd.arguments[4] = Addr2;
cmd.arguments[5] = Size2;

COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVD_CMD_BITSTREAM_BLOCK_DONE;
cmd.arguments[0] = Addr3;
```

```
cmd.arguments[1] = Size3;
cmd.arguments[2] = Addr4;
cmd.arguments[3] = Size4;
cmd.arguments[4] = 0;
```

Step 4: Stopping Recording

Stopping the recording is done with the FLUSH command. The following command performs this operation.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVD_CMD_FLUSH;
cmd.arguments[0] = 0;
```

8.2.3 Phase 3: Recording a QBOX Stream

A QBOX is a Mobilygen proprietary header that contains information about its contained data, specifically audio or video compressed streams. For example, a flag in the header indicates if the contained data is audio or video data. It is expected that if the host does MP4 multiplexing and demultiplexing, then it will stream QBOX data to the MG1264 Codec for decode.

The QBOX header is as follows.

```
typedef struct {
     uint32_t box_size;
     uint32_t box_type; // "qbox"
     uint32_t box_flags; // (version << 24 | box_flags)
     uint16_t sample_stream_type;
     uint16_t sample_stream_id;
     uint32_t sample_flags;
     uint32_t sample_cts;
     uint8_t sample_data[];
} QBox;
```

`sample_stream_type` is set to 0x0001 for AAC audio, and 0x0002 for AVC video.

`sample_stream_id` is currently set to the same value as `sample_stream_type`.

`box_flags` has two flags. Bit 0 is set if there is sample data after the header and bit 1 is set if this is the last sample in the stream.

`sample_flags` has three flags. Bit 0 indicates whether configuration information is contained in the sample. Bit 1 indicates if CTS is meaningful, bit 2 indicates if this is a sync point (I-frame).

This 24-byte structure is at the start of each bitstream block when the system has the stream type of QBOX. Additionally, when in QBOX mode, startcodes are not used and the AVC bitstream follows part 15 of ISO/IEC-14496 (AVC File Format) instead. The net effect of this mode compared to the previous mode is that the length of the following NAL unit replaces the 4-byte start code of 0x00000001.

The first QBOX sent by the MG1264 Codec when encoding, and the first QBOX that is expected to be received when decoding, contains two NAL units, one with the sequence parameter set and the other with the picture parameter set. Subsequent QBOX's contain one NAL unit with a single AVC access unit.

For example, here is the first QBOX header of AVC video.

```
0000002D  Size of QBOX is 2D bytes including the size field.
71626F78  "qbox" in ASCII
00000001  Sample data is present
00020002  AVC video
00000000  sample flags
00000000  sample CTS (not implemented yet)
```

The next data set is the sequence parameter set preceded by the NAL unit size. For example

```
00000009   NAL size (not including this field)
6742E01E   Sequence parameter data
DA02D0F4   Sequence parameter data
40         Sequence parameter data
00000004   NAL size
68CE3E80   Picture parameter data
```

Totalling all of the data bytes gives 0x2D which is the size of the QBOX given at the beginning.

Step 1: Configuring the Bitstream Type

This step is the same as “Step 1: Configuring the Bitstream Type” on page 112.

The default bitstream type for the MG1264 Codec firmware is the video elementary stream. In order to use QBOX, we must switch the type to QBOX. This must be done only once for the encoder at startup (it must be done for the decoder at startup as well).

This is done with the following command, which is only valid when the encoder is in IDLE state.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVE_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVE_CFP_BITSTREAM_TYPE_QBOX;
cmd.arguments[2] = 0;
```

Step 2: Putting the Encoder into the RECORD State

This step is the same as “Step 3: Putting the Encoder into the RECORD state” on page 113.

Step 4: Storing the bitstream

Handling the bitstream block ready events is done the same as in the previous phase except that the QBOX header should be examined for the timestamp (CTS) and sample flags to help the host multiplexer.

Step 5: Stopping the bitstream

Stopping the recording is done with the FLUSH command. The following command performs this operation.

```
COMMAND cmd;

cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVD_CMD_FLUSH;
cmd.arguments[0] = 0;
```

However, the key difference in QBOX recording is that the firmware will continue to send the buffered bitstream until the host receives the QBOX that has the last sample in stream (bit 1 of box_flags).

Chapter 9. Firmware Loader

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices contains a proprietary media processor that controls all of operations of the MG1264 Codec, as well as executing the Application programmers Interface. Because the MG1264 Codec has no non-volatile storage attached (such as Flash or ROM), the System Host CPU must initialize the MG1264 Codec. This initialization process involves

- Resetting the MG1264 Codec
- Writing a set of internal MG1264 Codec registers (called Configuration/Status Registers, or CSR registers)
- Downloading the firmware to the MG1264 Codec DRAM, and
- Writing a second set of MG1264 Codec CSR registers.

The first set of register writes initializes hardware modules such as the memory controller. The second set of register writes starts the media processor's execution.

All of the information required to initialize the MG1264 Codec firmware is contained in a binary file provided by Mobilygen. This binary file is referred to as the “Firmware Image”. This chapter describes the format of the binary image and how to read it.

It is important to note that the binary image is stored in a little endian format. Big-endian System Host CPUs will likely have to byte-reverse the image before storing it in their own Flash memory.

9.1 Firmware Image Format

The binary firmware image provided by Mobilygen starts with a header and then one or more sections in sequence. Each section consists of a 32-bit word that contains the section ID, followed by a variable number of 32-bit words. All fields in each section are always 32-bit words to make parsing easier. These fields are in little endian format and can be converted to big endian by reversing the four bytes in the 32-bit word (byte 3 switches with byte 0, byte 2 switches with byte 1, byte 1 switches with byte 2, byte 0 switches with byte 3.).

Note: The System Host CPU should read and process each section in order.

9.1.1 Header

The Header of the binary image contains two 32-bit words. The first word contains the characters “MBY0” and the second word contains the firmware version. The first three bytes are the version number and the last byte is the product code. For example, if the version field is 0x010204AA, then the version is 1.2.4, with the product code AA:

```
unsigned char[4] header = “MBY0”;
unsigned int32 version;
```

9.1.2 Global Pointer Block

The GPB section contains a single word whose value is the address of the “Global Pointer Block” for the firmware image. The Global Pointer Block is a structure that contains the address of the command block, the current event address, and status areas for the encoder, decoder, and system control. The address of this block can change between firmware builds. Therefore the System Host CPU must obtain the current Global Pointer Block address by parsing the firmware binary image.

The structure of the Global Pointer Block contains two 32-bit words. The first word is the section ID and has a value of four. The second 32-bit word is the Global Pointer Block.

```
unsigned int32 sectionId = 4;
unsigned int32 globalPointerBlockAddress;
```

In order to process this section, the System Host CPU must read and locally store the value of the Global Pointer Block address.

9.1.3 Pre-download CSR

There are two Configuration/Status Register sections in the binary image. The first CSR section is referred to as the “Pre-download” section and it is executed before downloading the firmware. The second CSR section is referred to as the “Post-download” section, and it is executed after downloading the firmware. Each CSR section has the same format; they are different only in their position in the file. As is expected, the Pre-download CSR section comes before the firmware download sections, and the Post-download CSR section comes after the firmware download sections.

The structure of the CSR section consists of the section ID with a value of two, the number of register writes, and then four 32-bit words per register write. The words per register are the block number, register address, register data, and register size. Register size will either be 1, 2 or 4 corresponding to an 8, 16 or 32-bit register. In all cases, the register data is a 32-bit field with the data always starting at bit 0:

```
unsigned int32 sectionId = 2;
unsigned int32 numRegisters;
repeat numRegisters

{
unsigned int32 blockId;
unsigned int32 address;
unsigned int32 data;
unsigned int32 size;
}
```

In order to process this section, the System Host CPU must write each register in order with the correct address, data, and size parameters.

9.1.4 Firmware

Boot

There are two firmware sections in the binary image; the Boot section and the Main section. The Boot firmware section contains a small amount of boot code for the MG1264 Codec that must be put into a different DRAM address from the Main firmware section. Each firmware section has the same format; they differ only in the location in the binary image.

The structure of the firmware section contains the section ID with value of one, the size of the firmware data to be downloaded in bytes, the start address of the firmware data, the partition ID of the firmware data, followed by the firmware data itself. The size of the firmware data will always be a multiple of four.

The Boot section is small, and is typically 1024 bytes of firmware data:

```
unsigned int32 sectionId = 1;
unsigned int32 firmwareSize;
unsigned int32 firmwareAddress;
unsigned int32 firmwarePartition;
repeat firmwareSize/4

{
unsigned int32 firmwareData;
}
```

In order to process this section, the System Host CPU must copy the firmware data to the address specified in the firmware section.

Main

The Main firmware section uses the same format as the Boot section, but is typically much larger and is stored at a different address using a different partition. In order to process this section, the System Host CPU must copy the firmware data to the address specified in the firmware section.

9.1.5 Uninitialized Data

The MG1264 Codec firmware requires that a section of the MG1264 Codec DRAM be set to zero before execution begins. This section is called the BSS section.

The structure of the BSS section is similar to the firmware section, except that there is no firmware data. It consists of the section ID with a value of three, the size of the area to be zeroed in bytes, the start address of the zero data, and the partition ID to use. The size of the BSS area will always be a multiple of four:

```
unsigned int32 sectionId = 3;
unsigned int32 bssSize;
unsigned int32 bssAddress;
unsigned int32 bssPartition;
```

In order to process this section, the System Host CPU must zero-out the MG1264 Codec DRAM starting at the given address for the specified number of bytes.

9.1.6 End

The End section consists simply of the section ID with a value of five. This section is at the end of the binary image, and can be used by the System Host CPU to indicate that the file was parsed successfully.

9.2 Sample Code

Mobilygen provides sample code for the firmware loader. This code assumes that the System Host CPU is the same endian structure as the binary image. Since the binary image is originally little endian, a big endian host will have to swap the data within the file, with the exception of the first MBY0 string, which is a character string that does not need swapping.

Pseudocode for the sample code follows, assuming that the System Host CPU is little endian. Byte reversal can be done using the macro:

```
#define SWAP_ENDIAN(A) (((A & 0xff000000) >> 24) | \
                        ((A & 0x00ff0000) >> 8 ) | \
                        ((A & 0x0000ff00) << 8 ) | \
                        ((A & 0x000000ff) << 24))
```

The pseudocode contains the functions “CopyToDram”, “ZeroDram”, and “WriteRegister”. These are functions that copy a block of local memory to the MG1264 Codec memory, zero-out a block of MG1264 Codec memory, and write to a CSR register. Mobilygen also provides a driver layer for the MG1264 Codec Host Interface called the Hardware Abstraction Layer (QHAL) which contains code to perform these functions. It is expected that these calls are implemented using real QHAL calls:

```
int qmmLoadAndRun(char *imageBuffer, int imageSize)
{

    // set current position of the firmware image to the start
    currentPos = imageBuffer;

    // read the first 4 bytes and check against the magic number and
    // fail if they do not match
    if ((imageBuffer[0] != 'M') || (imageBuffer[1] != 'B') ||
        (imageBuffer[2] != 'Y') || (imageBuffer[3] != '0'))
    {
        printf(“bad magic number\n”);
```

```
        return(0);
    }

    // move past the header to the version field and retrieve the version
    currentPos++;
    version = *currentPos++;

    // Continue in a loop processing each section as it is found.
    // In order to handle corrupted images, the loop exits as
    // soon as the current firmware image pointer goes past the
    // size of the firmware image.
    while (currentPos - imageBuffer < imageSize)
    {
        // read the id of the current section and move to the next field
        sectionId = *currentPos++;

        switch (sectionId)
        {
            case QMM_LOAD_SECTION:

                // read the size, address, and partition of the firmware
                // data to be downloaded.
                size = *currentPos++;
                addr = *currentPos++;
                partition = *currentPos++;

                // copy the firmware data to codec memory
                CopyToDram(addr, size, (char *)currentPos, partition);

                // move to next section
                currentPos = (int*)((char *)currentPos + size);


                break;

            case QMM_CSR_SECTION:
                // get number of registers to write
                numRegisters = *currentPos++;

                // iterate across the set of registers, writing each one as they
                // are read.
                for (i = 0; i < numRegisters; i++)
                {
                    csrBlock = *currentPos++;
                    csrAddr = *currentPos++;
                    csrData = *currentPos++;
                    csrSize = *currentPos++;

                    // write the register
                    WriteRegister(csrBlock, csrAddr, csrSize, csrData);
                }

                break;
```

```
            case QMM_BSS_SECTION:

                // read the size, address and partition of the bss section
                size = *currentPos++;
                addr = *currentPos++;
                partition = *currentPos++;

                // clear codec memory as specified
                ZeroDram(addr, size, partition);
                break;

            case QMM_GPB_SECTION:

                // retrieve the GPB address for this image
                gpb = *currentPos++;
                break;

            case QMM_END_SECTION:

                // Flag that the end section has been found
                currentPos++;
                break;

        }
    }

}
```

Chapter 10. Application Programming Interface

The MG1264 Low Power H.264 and AAC Codec for Mobile Devices is designed for use for mobile and wall-powered applications. The MG1264 Codec integrates the Media Processor Multi-threaded Microcontroller along with specialized hardware modules that are responsible for the real-time encoding and decoding of video and audio streams. This processing is done under the control of firmware running on the micro controller that presents a programming interface to the System Host CPU.

This chapter describes the Application Programming Interface (API) for the Media Processor firmware and how the Media Processor responds to its API calls. It is the functional specification for the firmware and a programming manual for the System Host CPU-based software.

The API is partitioned into five types of interface elements that are used by the System Host CPU to control the firmware. They are:

- The Firmware State Machine
- Commands sent from the System Host CPU to the firmware that change the state of the firmware.
- Configuration information sent from the System Host CPU to the firmware that change parameters that control how the firmware operates in the various states.
- Asynchronous notifications sent from the firmware to the System Host CPU to inform the System Host CPU of specific events.
- Status information made available by the firmware that can be polled by the System Host CPU to obtain information about how the firmware is operating. This status information is state- and bitstream-dependent and changes over time, often in response to an asynchronous notification.

Taken together, these elements comprise the logical interface of the firmware. Three additional interface elements must be described to complete the picture of how the firmware is controlled. These elements are:

- How to send commands and read status and events from the System Host CPU.
- How to format bitstreams so that they are properly decoded by the Media Processor firmware.

- How to read encoded bitstreams from the Media Processor firmware.

All eight of these interface elements are described in this document. The physical connection between the System Host CPU and the Media Processor Controller is presented first, followed by the logical interface of the firmware, and then the bitstream interfaces for the encoder and decoder.

10.1 Host Interface and the Hardware Abstraction Layer

The MG1264 Codec interfaces with an external System Host CPU through its MG1264 Codec Host Interface, which is accessed through a 16-bit SRAM-like asynchronous bus. In this configuration, the System Host CPU is the bus Master, and the MG1264 Codec is the Slave.

The MG1264 Codec Host Interface provides the System Host CPU with the ability to read/write the MG1264 Codec's DRAM, read/write the MG1264 Codec's Configuration/Status Registers (CSR), and send bitstream data to the decoder. The MG1264 Codec Host Interface is also used to implement an inter-processor communication protocol using special mailbox registers and the System Host CPU interrupt signal.

The QHAL is Mobilygen's Hardware Abstraction Layer that implements the control logic required to use the host bus effectively. The QHAL is meant to be ported and executed on the System Host CPU, and is written in ANSI-C.

The QHAL is made up of the external memory driver (*qhal_em*), the CSR register driver (*qhal_qcc*), the bitstream transfer driver (*qhal_bs*), the mailbox control driver (*qhal_mbox*), and the host bus register driver (*qhal_host*, also known as the low-level driver). The *qhal_host* driver is the only module that must change when moving between different host processors. Once the *qhal_host* is properly functioning, the rest of the QHAL modules will work. For the purposes of this document, *qhal_host* and *qhal_qcc* can be ignored. The firmware API can be implemented only with *qhal_em*, *qhal_bs*, and *qhal_mbox* calls. The *qhal_qcc* API is used primarily for booting the MG1264 Codec.

The structure of the QHAL is shown in Figure 10-1.

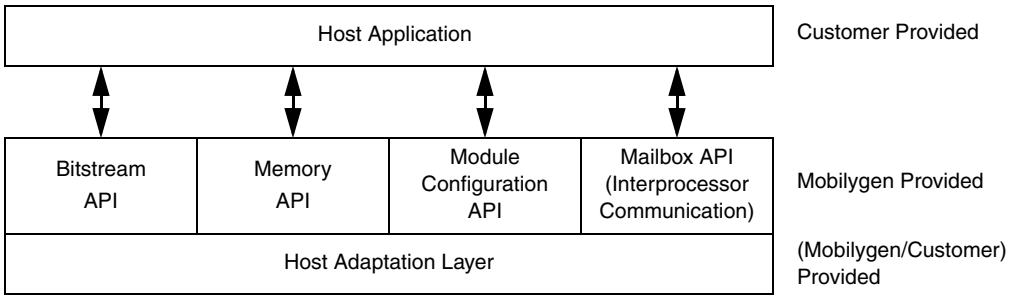


Figure 10-1 QHAL Structure

10.1.1 QHAL_EM

The *qhal_em* is the driver used to access the MG1264 Codec's external DRAM. This driver configures the memory channel and provides interfaces to the read/write blocks of memory.

The MG1264 Codec Host Interface provides two concurrent memory channels; one is used for bitstream data, and the other is used for command and control. Both channels can be used in PIO mode, but only the bitstream channel can be used with hardware flow-control DMA. In systems that do not have hardware flow-control DMA, only the command channel should be used.

There are two sets of read/write functions; they are 16-bit word read/write and byte-sized read/write functions. In either case, the total size read or written must be a multiple of 32 bits, but the word-size read/write functions do endianness conversion if required. The Media Processor processor is big-endian meaning that *qhalem_read_words* and *qhalem_write_words* will perform a byte-swap before writing the data if the System Host CPU is little endian.

> Note that swapping is typically only required for commands and events that are relatively small. Bitstreams are always transferred using the byte-sized functions (*qhalem_read_bytes*) that force the data to be big endian as required by most multimedia specifications. The MG1264 Codec's host bus hardware contains endianness conversion that eliminates any performance penalty for reading bitstreams.that never swap data.

The header file for the *qhal_em* module is:

```
typedef enum {
    QHALEM_ACCESSTYPE_CMD,
    QHALEM_ACCESSTYPE_STREAM
} QHALEM_ACCESSTYPE;

typedef enum {
    QHALEM_MODE_FBFRAME,
    QHALEM_MODE_FBFIELD,
    QHALEM_MODE_LINEAR
} QHALEM_MODE;

typedef enum {
    QHALEM_PRIORITY_NORMAL=0,
    QHALEM_PRIORITY_LOWER=1,
    QHALEM_PRIORITY_HIGHER=2,
    QHALEM_PRIORITY_HIGHEST=3
} QHALEM_PRIORITY;

typedef enum {
    QHALEM_BURSTSIZE_8WORDS=0,
    QHALEM_BURSTSIZE_16WORDS=1,
    QHALEM_BURSTSIZE_32WORDS=2,
    QHALEM_BURSTSIZE_64WORDS=3
} QHALEM_BURSTSIZE;

/* No one should modify a handle or what is inside */
typedef int qhalem_handle_t;

qhalem_handle_t qhalem_open(QHALEM_ACCESSTYPE type,QHALEM_MODE
txmode);

int qhalem_setconfig(qhalem_handle_t em_h, char threshold,
QHALEM_BURSTSIZE burst,   QHALEM_PRIORITY priority);
```

```
int qhalem_read_bytes(qhalem_handle_t em_h, unsigned char blockID,
unsigned long addr, char *buffer, int nBytes);

int qhalem_read_words(qhalem_handle_t em_h, unsigned char blockID,
unsigned long addr, long *buffer, int nWords);

int qhalem_write_bytes(qhalem_handle_t em_h, unsigned char
blockID, unsigned long addr, char *buffer, int nBytes);
int qhalem_write_words(qhalem_handle_t em_h, unsigned char
blockID, unsigned long addr, long *buffer, int nWords);

int qhalem_close(qhalem_handle_t em_h);
```

10.1.2 QHAL_MBOX

The *qhal_mbox* driver is used to perform inter-processor communication between the System Host CPU and the Media Processor. It is a set of high-level functions that manipulate special mailbox CSR registers. There are two mailboxes in the system (called 0 and 1). Each mailbox has a data register and an event source register. Mailbox 0 is for Mobilygen internal use, and mailbox 1 is for application use.

The mailboxes registers are used to generate COMMAND_READY interrupts and EVENT_DONE interrupts from the System Host CPU to the Media Processor. COMMAND_READY interrupts are generated by *qhalmbox_write* operations (the actual written data is ignored), and EVENT_DONE interrupts are generated using *qhalmbox_read* operations. The meaning of COMMAND_READY and EVENT_DONE are explained in “Event Transfer Protocol” on page 134.

An application can determine which, if any, event occurred using the *qhal_mbox_get_event* function. This function returns if none, either, or both of the COMMAND_DONE or EVENT_READY interrupts have occurred. An application can either poll this function, or implement an interrupt handler that wakes up a blocked thread that then calls this function.

The *qhal_mbox_get_event* function returns a bit field that contains an indication of which event occurred. The bit fields are called QHAL_MBOX_EVENT_READ, and QHAL_MBOX_EVENT_READY. The Read event corresponds to COMMAND_DONE, and the Ready event corresponds to EVENT_READY.

The full *qhal_mbox.h* header is shown as:

```
typedef enum {
    QHAL_MBOX0,
    QHAL_MBOX1
} QHALMBOX_DEV;

#define QHALMBOX_EVENT_NONE     0
#define QHALMBOX_EVENT_READY    1
#define QHALMBOX_EVENT_READ     2
#define QHALMBOX_EVENT_ALL      3
typedef int QHALMBOX_EVENT;

qhalmbox_handle_t qhalmbox_open(QHALMBOX_DEV mbox);
```

```
int qhalmbox_get_event(qhalmbox_handle_t mbox_h,QHALMBOX_EVENT
*event);
int qhalmbox_read(qhalmbox_handle_t mbox_h, unsigned long *datap);
int qhalmbox_write(qhalmbox_handle_t mbox_h, unsigned long data);
int qhalmbox_close(qhalmbox_handle_t mbox_h);
```

10.1.3 QHAL_BS

The *qhal_bs* driver is used to send compressed data to the MG1264 Codec's input data port. Other than the traditional open and close functions, it features a single function; *qhalbs_write_bytes*(). This function sends byte stream data to the MG1264 Codec with appropriate endianness conversion. Refer to "H.264/ACC Decoder Interface Object" on page 157 for additional information.

```
qhalbs_handle_t qhalbs_open();
int qhalbs_setconfig(qhalbs_handle_t bs_h,int threshold);
int qhalbs_write(qhalbs_handle_t bs_h, char *buffer, int length);
int qhalbs_close(qhalbs_handle_t bs_h);
```

10.2 Media Processor Firmware Programming Model

This section describes the programming model used by the Media Processor firmware.

10.2.1 Control Objects

The firmware presents multiple "Objects" to the System Host CPU. Each of the objects has a well-defined state machine, a set of commands that it accepts and acts upon, a set of configuration parameters whose values can be set by the System Host CPU, a set of asynchronous event notifications that it sends to the System Host CPU, and status that can be read by the System Host CPU.

The Media Processor firmware presents the following objects (called control objects), each of a different type:

- System Control
- H.264/AAC AV Encoder
- H.264/AAC AV Decoder

Each control object is assigned a unique ID, and each command and status message is tagged with this ID.

10.2.2 Commands, Events, and Inter-Processor Communications

The primary methods of communication between the System Host CPU and the Media Processor firmware are commands and events. Commands are sent from the System Host CPU to the firmware, and events are sent from the firmware to the System Host CPU.

A "Command" is a request by the System Host CPU for the Media Processor firmware to either change state, or to configure an operational parameter. Commands are executed immediately upon request, in the order in which they are received. If the command is a state-change request, then the state change operation will be complete when the command completes execution.

An "Event" is a notification sent by the Media Processor firmware to the System Host CPU that a specific event has occurred. The event optionally carries a set of parameters that give more information about the event at the time that it occurred. New events are internally queued by the Media Processor firmware while the System Host CPU is processing the current event. The queue depth is configurable and can be set large enough so that no event is lost (several hundred events).

The System Host CPU writes commands over the MG1264 Codec Host Interface to area in the MG1264 Codec's external DRAM called the "Command Block." Similarly, events are stored in the MG1264 Codec's external DRAM and are read by the System Host CPU using the MG1264 Codec Host Interface. The event area should be treated as read-only by the System Host CPU.

The transfer protocol of both commands and events is fully handshaked, and uses interrupts to ensure that no data is lost. The details of this protocol are provided in "Sending a Command to the Firmware" on page 132 and "Reading Events from the Media Processor Firmware" on page 133.

It is recommended that the host code follow the Mobilygen reference design structure described in "Sample Host Code Architecture" on page 227 to manage sending commands and reading events. This structure is proven and handles the important corner case of receiving an event while waiting for a command to be processed.

10.2.3 Global Pointer Block

There are a number of important shared data structures stored in the MG1264 Codec's DRAM that must be accessed by the System Host CPU. The addresses of these data structures are found in the Global Pointer Block structure. The address of the global pointer block is determined when the firmware image is downloaded to the Media Processor.

Each of the structure members is a big-endian, 32-bit field. The global data block structure is:

```
typedef struct
{
    COMMAND                     *cmdBlock;
    EVENT                       *evBlock;
    void                        *systemControlStatus;
    void                        *avDecoderStatus;
    void                        *avEncoderStatus;
} GLOBAL_POINTER_BLOCK;
```

The command block is a shared memory buffer used for sending commands from the System Host CPU to the firmware. The *cmdBlock* field contains the address of the command block in the MG1264 Codec's DRAM.

The event block is a shared memory buffer used to send asynchronous event information from the firmware to the System Host CPU. Its operation is described in "Reading Events from the Media Processor Firmware" on page 133. Note that events are queued internally by the Media Processor firmware. Therefore, the System Host CPU must fetch the address of the current event for EVERY event. The *evBlock* field contains the address of the current event.

The three status blocks are used by the firmware to post status information for the System Host CPU to poll. There is one status block for each of the three control objects in the system. The status block pointers contain the addresses for these blocks.

.

10.2.4 Sending a Command to the Firmware

Command Block

The System Host CPU uses the Command Block to send a command to the Media Processor firmware The address of the command block is stored in the global pointer block. Each command contains the target control object ID, the command opcode, up to six 32-bit arguments, a return code, and up to seven 32-bit return values.

Each field is a big-endian, 32-bit field. The structure of the command block is shown as:

```
typedef struct
{
    CONTROLOBJECT_ID    controlObjectId;
    unsigned int        opcode;
    unsigned int        arguments[6];
    unsigned int        returnCode;
    unsigned int        returnValues[7];
} COMMAND;
```

Command Transfer Protocol

Sending a command from the System Host CPU to the Media Processor firmware is a fully handshaked transaction that ensures that no data is lost. The handshaking is done through two interrupts: the COMMAND_READY interrupt and the COMMAND_DONE interrupt. The COMMAND_READY interrupt is generated by the System Host CPU to signal the firmware that a new command has been written to the command block. The COMMAND_DONE interrupt is generated by the Media Processor firmware to signal to the System Host CPU that the command execution has completed. No new commands can be generated by the System Host CPU until the COMMAND_DONE interrupt has been received. The System Host CPU generates the COMMAND_READY interrupt through writes from the mailbox register in the MG1264 Codec Host Interface.

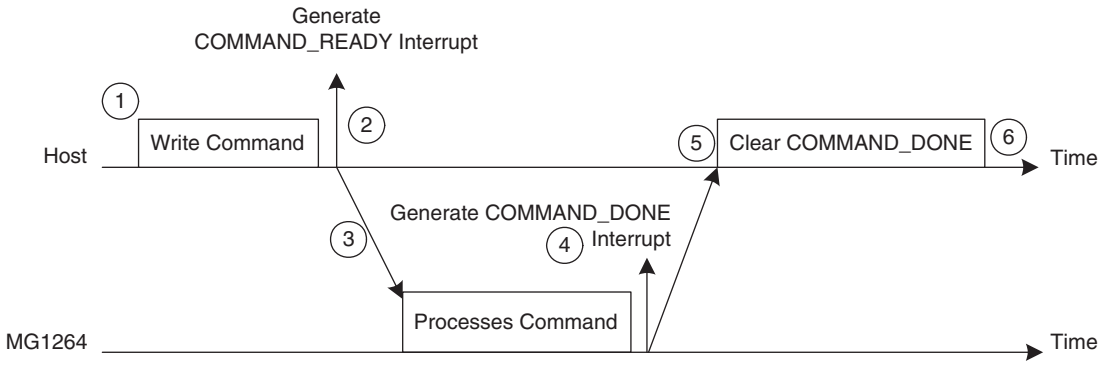


Figure 10-2 Command Transfer Timing

The command transfer protocol is:

1: The System Host CPU writes the command block including opcode, control object ID, and arguments. Only the necessary number of arguments need by written. This is done using the *qhalem_write_words* API call. It is important to use the *qhalem_write_words* call as this corrects for endian-ness.

2: The System Host CPU writes to the mailbox register to assert the COMMAND_READY interrupt and clear the COMMAND_DONE interrupt. This is done through a call to the function *qhalmbox_write*().

3: The Media Processor firmware responds to the interrupt and processes the command.

4: The Media Processor firmware reads from the mailbox register to assert the COMMAND_DONE interrupt and clear the COMMAND_READY interrupt.

5: The System Host CPU waits for and receives the COMMAND_DONE interrupt. The COMMAND_DONE and EVENT_READY interrupts are multiplexed on the same interrupt pin. The System Host CPU must read the interrupt source register to determine which interrupt is the source. This is done through the API *qhalmbox_get_event*() call. This API call also clears the mailbox interrupt bit.

6: The System Host CPU reads the command return code and the return values from the command block.

A return code of zero indicates the command was rejected. A return code of one means success. Any other positive return code indicates success with additional information encoded in the value. The return values can be anything and are command-specific.

10.2.5 Reading Events from the Media Processor Firmware

Events are sent by the Media Processor firmware to the System Host CPU using the same handshaking mechanism that is used to send commands, but in reverse. Events operate on a publish/subscribe paradigm so that the System Host CPU only sees events to which it has subscribed. Some of the events are periodic and relatively high in frequency (once per frame/field/picture, etc.), and are intended only for debug purposes. By default, no events are subscribed.

Event Block

Event Blocks are used by the firmware to store a single event for the System Host CPU. Event blocks are internally queued by the Media Processor firmware and then sent one-by-one to the System Host CPU for processing. The System Host CPU can find the address of the current event (the one to be processed) by reading the event block pointer in the global data pointer block. **It is critical to understand that this address will change**, **and the address must be re-read for each event**.

Each event block contains the event ID, the source control object ID, a 32-bit timestamp measured in microseconds, and a variable length payload up to a maximum of thirteen words. The event ID is a globally unique number that identifies the event type. Each field is 32-bits, big endian. The structure of the event block is shown as:

```
typedef struct
{
    CONTROLOBJECT_ID    controlObjectId;
    EVENT_ID            eventId;
    unsigned int        timestamp;
    unsigned int        payload[13];
} EVENT;
```

Event Transfer Protocol

The transfer protocol for sending events from the Media Processor firmware to the System Host CPU is identical to the command transfer protocol except the role of the processors is reversed. Sending an event is a fully-handshaked transaction that ensures that no data is lost. The handshaking is done through two interrupts: the EVENT_READY interrupt and the EVENT_DONE interrupt.

The EVENT_READY interrupt is generated by the Media Processor firmware to signal to the System Host CPU that a new event has been written to the event block. The EVENT_DONE interrupt is generated by the System Host CPU to signal the firmware that the event handling has completed. No new events can be generated by the firmware until the EVENT_DONE interrupt is received. The System Host CPU generates the EVENT_DONE interrupt through reads from the mailbox register in the MG1264 Codec Host Interface.

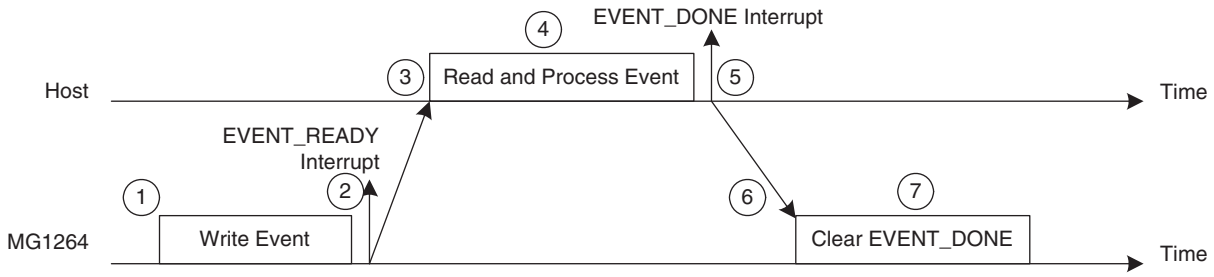


Figure 10-3 Event Transfer Timing

The complete Event Transfer protocol is:

1. The Media Processor firmware writes the event ID, control ID, and payload to the event block, and then writes to the mailbox register to assert the EVENT_READY interrupt and clear the EVENT_DONE interrupt.
2: The System Host CPU responds to the interrupt and reads the current event block address from the global pointer block. The System Host CPU must read the interrupt source register to determine if the interrupt is the EVENT_READY interrupt.
3: The System Host CPU processes the event.
4: The System Host CPU reads from the mailbox register to assert the EVENT_DONE interrupt and clear the EVENT_READY interrupt. This is done using the *qhalmbox_read*() API call.
5: The Media Processor firmware waits for and receives the EVENT_DONE interrupt.
6: The Media Processor firmware clears the EVENT_DONE interrupt.

The internal queueing mechanism can be represented as shown in Figure 10-4.

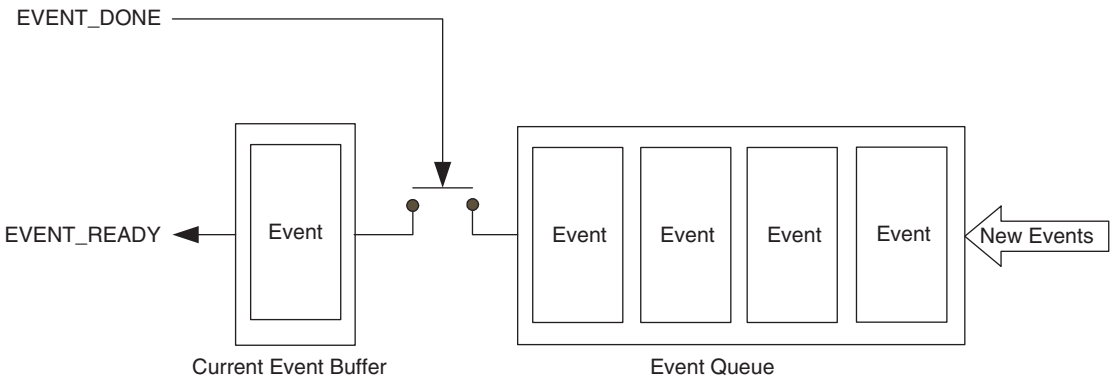


Figure 10-4 Event Queuing

10.2.6 Subscribing and Unsubscribing to Events

By default, all events are unsubscribed, meaning that the System Host CPU will receive no events. Each event that the System Host CPU is interested in receiving must be explicitly subscribed using the SUBSCRIBE_EVENT command. Similarly, events can be unsubscribed using the UNSUBSCRIBE_EVENT command. The argument list for both commands is a NULL terminated list of event IDs that should be subscribed/unsubscribed.

SUBSCRIBE_EVENT

Command Name	Q_CMD_OPCODE_SUBSCRIBE_EVENT
Arguments	Variable list of 32-bit words. Each word contains a valid event ID. The list of IDs should be terminated by a NULL (0) 32-bit word
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The Subscribe Event can be issued at any time, although it is expected that the host application will subscribe to a set of events at startup.

For example:

```
COMMAND cmd;
cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_SUBSCRIBE_EVENT;
cmd.arguments[0] = Q_AVE_EV_BITSTREAM_BLOCK_READY;
cmd.arguments[1] = Q_AVE_EV_VIDEO_FRAME_ENCODED;
cmd.arguments[2] = 0;
```

UNSUBSCRIBE_EVENT

Command Name	Q_CMD_OPCODE_UNSUBSCRIBE_EVENT
Arguments	Variable list of 32-bit words. Each word contains a valid event ID. The list of IDs should be terminated by a NULL (0) 32-bit word.
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	As previously stated, the typical operational procedure is to subscribe to events at startup, and does not either unsubscribe or further subscribe during operation. However, these features are supported for debug purposes or for the implementation of features not anticipated at this time.

For example:

```
COMMAND cmd;
cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_UNSUBSCRIBE_EVENT;
cmd.arguments[0] = Q_AVE_EV_BITSTREAM_BLOCK_READY;
cmd.arguments[1] = Q_AVE_EV_VIDEO_FRAME_ENCODED;
cmd.arguments[2] = 0;
```

10.2.7 Configuration Parameters

Each control object presents a set of configuration parameters for the System Host CPU to set. These parameters control how the object behaves in each state, and also how it transitions states.

There are two types of configuration parameters: single-buffered and double-buffered. Single-buffered parameters either take effect immediately or upon the next state transition, as indicated by the parameter. Double-buffered parameters take effect only when the host issues a matching ACTIVATE command. For example, most of the video encoder parameters are double-buffered so that the host can change a group of parameters at one time while recording.

A configuration parameter has a unique ID and an associated 32-bit value. The 32-bit value can include multiple bit fields. Single buffered configuration parameters are set using the CONFIGURE command (see “Configure Command” on page 137), which has the same opcode for all control objects. Double buffered parameters are set by different commands that are explained in each of the control object's API section.

Configure Command

Command Name	Q_CMD_OPCODE_CONFIGURE
Arguments	Variable list of 32-bit words. Each pair consists of a configuration parameter name and a parameter value.
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	All single-buffered configuration parameters are set using the CONFIGURE command. Each parameter has a one 32-bit value associated with it that is stored by the firmware. The arguments to the CONFIGURE command are parameter or value pairs. If fewer than three pairs are specified, then a parameter value of 0 terminates the list.

For example:

```
COMMAND cmd;
cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
cmd.opcode = Q_CMD_OPCODE_CONFIGURE;
cmd.arguments[0] = Q_AVD_CFG_BITSTREAM_TYPE;
cmd.arguments[1] = Q_AVD_CFP_BITSTREAM_TYPE_QBOX;
cmd.arguments[2] = 0;
```

10.2.8 Status Block

Each control object has a status block located in the MG1264 Codec's DRAM that is pointed to by the global pointer block. The intent of the status block is to store information that does not change over time, or whose changes do not need to be synchronized with the System Host CPU. The System Host CPU can read the contents of the status block at any time simply by accessing the Media Processor firmware memory using the *qhalem_read_words* API. The specific layout of each status block is described in each control object's section.

10.3 Bitstream Formats

The Media Processor is capable of generating and decoding any bitstream formats, but the firmware currently only supports QBox, Elementary, and MP4.

10.3.1 QBox Bitstream Format

The QBox format consists of a simple header preceding audio and video access units. It is designed for applications where the System Host CPU is doing bitstream multiplexing or demultiplexing, and can be considered an interchange format. When encoding, the Media Processor firmware sends access units (either compressed audio or video frames) following a standard header (called the QBox Header). This header has the size of the access unit and information about the contents. It is expected that the System Host CPU will only use the header for informational purposes and will not store entire QBoxes. When decoding, the System Host CPU must then generate these headers on the fly, and send the header and payload to the Media Processor for decoding.

The first video QBox contains the AVC sequence/picture parameter set NAL unit. Subsequent QBox headers contain either I-frames or P-frames. QBoxes that contain I-frames contain both a picture parameter set NAL unit followed by the video frame NAL unit. QBoxes that contain only P-frames contain only the frame NAL unit.

If the selected audio codec is AAC then the first audio QBOX contains configuration information according to the AudioSpecificConfig() structure as specified in section 1.6.2.1 of ISO/IEC 14496-3:2001 (MPEG4 Systems). The audio object type is 2 for AAC-LC. When decoding a stream, the configuration QBOX must be sent first after a transition from IDLE to PLAY.

Other audio codecs do not have a configuration QBOX as the relevant header information is stored in the audio elementary stream.

As a C structure, the QBox header structure is:

```
typedef struct {
     uint32 box_size;
     uint32 box_type;
     uint32 box_flags;
     uint16 sample_stream_type;
     uint16 sample_stream_id;
     uint32 sample_flags;
     uint32 sample_cts; // optional
     uint8 sample_data[];
} QBox;
```

box_size: Size of the box including the header.

box_type: Always four characters “qbox”.

box_flags: The upper eight bits are the header version. The lower 24 bits are flags. Bit 0 is set if there is sample data in the box. Bit 1 is set if this is the last access unit in the stream. Bit 2 is set if the QBox is followed by padding bytes to make the QBox size, plus the padding bytes a multiple of 4 bytes.

sample_stream_type: Set to 1 if it is an AAC audio frame or configuration data, or set to 2 if it is an H.264 frame or configuration data.

sample_stream_ID: Unused at this time.

sample_flags: Bit 0 is set if the data contains configuration information for the decoder. Bit 1 is set if the CTS field is present and valid. Bit 2 is set if the video frame is a synchronization point (meaning I frame for H.264), and bit 3 is set if the frame is disposable (meaning a B frame in H.264). Bit 4 is set if the audio or video sample is the result of a MUTE command sent to the AV encoder. Bits 30-31 represent the number of leading padding bytes in the QBox (0-3) that are skipped by the MG1264 Codec demultiplexer.

cts: Sample composition time in 90 kHz ticks.

Reading QBOX Bitstreams

When reading the bitstream data from the MG1264 Codec, special care is required if the host processor is little endian. As mentioned in the QHAL_EM driver description, endianness conversion is done for *qhalem_read_words*(), but *qhalem_read_bytes*() forces big-endianness for bitstream transfers. Therefore, the host must either read the QBOX header first using *qhalem_read_words*() and then the bitstream using *qhalem_read_bytes*(), or use a single *qhalem_read_bytes*() call to read both the header and bitstream, and then perform endianness conversion on the header afterwards.

QBOX Payload

For a QBox that contains AVC data as defined in ISO14996-3 as AudioSpecificConfig., there will be an integer number of Network Abstraction Layer (NAL) units contained within the box. The box may contain zero or one slice data NAL unit, and an arbitrary number of other NAL units (such as SEI messages, end of stream, end of sequence etc.). The format of the data consists of a 32-bit value containing the size of the NAL unit (including the four bytes used to encode the size) followed by the NAL unit data.

For QBoxes that contain AAC data, there will be zero or one raw data blocks per box. The first audio-related QBox will contain the stream configuration information as defined in ISO14996-3 as AudioSpecificConfig..

10.3.2 Elementary Video

The Elementary Video stream accepted and generated by the Media Processor firmware is specified in ISO/IEC 14496-10 Annex B. This stream consists of a sequence of NAL units with each NAL unit proceeded by a startcode. The bitstream data corresponding to one event is similar to the data that is contained in a QBOX. That is, an integer number of NAL units with each NAL unit preceded by a 0x00000001 startcode. Note that when the decoder is in elementary video mode, it cannot accept or generate compressed audio data at the same time.

10.3.3 MP4

TBD

10.4 System Control Interface Object

10.4.1 Overview

The System Control interface object is responsible for overall system control such as power management, audio, video input/output timing, as well as the video and OSD display.

Video Display

The video display features three display planes that are stacked (top to bottom) as OSD, video frame 1 and video frame 0. That is, OSD is overlaid on top of video frame 1 which is overlaid on top of video frame 0.

The host has control over which planes are enabled. In the case of the video planes, the host also has control over whether the encoder or decoder output is routed to the plane. Each plane then has independent scaling and placement on the display. These capabilities allow for picture-in-picture operation (PIP).

On-Screen Display

The OSD system offers a full screen display with eight bits per pixel using a full 32-bit color indexed by the pixel's value. The set of 256 colors that can be used is referred to as the palette and is stored as red, green, blue, and alpha. The OSD system also offers the host the ability to download up to 128 "pixel maps" which are rectangular images. The pixel maps can be downloaded in raw form (meaning only the pixel data is downloaded) or as BMPs where the palette and pixel data are downloaded together. The downloaded palette can be used to set the system palette.

Due to performance considerations, there are some restrictions in the API.

1. Width of the Bitmap and OSD Screen Size must be multiple of four.
2: Start position for the OSD destination screen has to be multiple of four.

10.4.2 Object ID

The system control object has the object ID of 0x1.

10.4.3 State Machine

The system control object has no state machine. It is considered to be always in the ENABLED state.

10.4.4 Commands

ECHO

Command Name	Q_SYS_CMD_ECHO
Arguments	Any 32-bit value.
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The ECHO command is used primarily for debug and bring-up purposes. When the ECHO command is received, a corresponding ECHO event, Q_SYS_EV_ECHO, is created with the first payload entry of the event being the same as the first argument of the command.

For example:

```
COMMAND cmd;
cmd.controlObjectId = SYSTEMCONTROL_CTRLOBJ_ID;
cmd.opcode = Q_SYS_CMD_ECHO;
cmd.arguments[0] = 1; // any arbitrary 32 bit value
```

POWERDOWN

Command	Q_SYS_CMD_POWERDOWN
Arguments	0 = To exit 1 = To enter sleep
Return Code	Cannot be checked, see description
Return Values	None
Valid States	All
Description	The POWERDOWN command is used to transition the MG1264 Codec to and from a sleep mode where very little power is consumed. If the argument value is 1, then the codec enters the POWERDOWN state, if its 0 then it wakes up. **Note that the command sends a COMMAND_DONE interrupt as all other commands do, but it is critical to note that the System Host CPU code cannot check the return code when entering sleep, because the memory controller has been placed into an AUTO-REFRESH state**. The command cannot fail, and it is assumed that if a COMMAND_DONE interrupt is received, that the command was accepted.

10.4.5 OSD Commands

RES_DOWNLOAD

Command	Q_SYS_CMD_RES_DOWNLOAD
Arguments	0 = Resource type 1 = File size of the resource file to be downloaded.
Return Code	0 = Failure 1 = Success
Return Values	0 = Handle of the bitmap 1 = Address for downloading the bitmap file.
Valid States	All
Description	The Resource Download Handle Request command is used to request a storage place for the resource file, h264Iframe, and bitmap data or bitmap palette. The first argument of the command is the Resource Type, which can be one of the following: 1. Q_SYS_RCTYPE_BITMAP: A BMP file 2. Q_SYS_RCTYPE_USER_DEFINED: User defined data. You can use user defined data for downloading any type of data, including a raw pixel map. The general model for using this command is to send the command to ask for space in MG1264 Codec memory, and then download the resource itself to the address provided by the codec in the return value.

RES_RELEASE

Command	Q_SYS_CMD_RES_RELEASE
Arguments	Handle of the resource to be freed.
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The Bitmap Handle Release command is used to free the storage place for the resource handle specified. Once the memory is free, it can then be reused for downloading other resource files.

OSD_PALETTE (Set Single Entry)

Command	Q_SYS_CMD_OSD_PALETTE
Arguments	0 = Sub-command set to 0 for set operation 1 = Palette index (0-255) 2 = Red 3 = Green 4 = Blue 5 = Alpha
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD_PALETTE command, and its subcommand of 0 is used to set a single palette entry. The first argument must be zero to indicate a read operation. The next argument is the palette index to read, and the subsequent are the 8-bit red, green, blue, and alpha fields of the palette entry.

OSD_PALETTE (Get Single Entry)

Command	Q_SYS_CMD_OSD_PALETTE
Arguments	0 = Sub-command set to -1 for read operation 1 = Palette index (0-255)
Return Code	0 = Failure 1 = Success
Return Values	0 = Red 1 = Green 2 = Blue 3 = Alpha
Valid States	All
Description	The OSD_PALETTE command, and its subcommand of -1 is used to read a single palette entry. The first argument must be -1 to indicate a write operation. The next argument is the palette index to update.

OSD_PALETTE (Set Multiple Entries)

Command	Q_SYS_CMD_OSD_PALETTE
Arguments	0 = Palette address in codec memory 1 = Start index to update 2 = End index to update
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD_PALETTE command is used to set the multiple palette entries based on already downloaded memory, typically a palette contained in a BMP file. The first argument is the address of the palette, the second argument is the start index, and the third argument is the end index. If the entire palette is to be set from a BMP, the correct offset from the BMP's address in memory must be selected and the start and end indexes must be 0 and 256.

OSD_SCRN_ALPHA

Command	Q_SYS_CMD_OSD_SCRN_ALPHA
Arguments	0 = Enable or disable 1 = Screen global alpha value
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD_SCRN_ALPHA command is used to use a single global alpha for the entire OSD plane instead of a per-pixel alpha used in the palette.

BMPDATA_BLIT

Command	Q_SYS_CMD_OSD_BMPDATA_BLIT
Arguments	0 = Bitmap data address (must be 4-byte aligned) 1 = xAddress of the OSD memory to which the bitmap is blitting to 2 = yAddress of the OSD memory to which the bitmap is blitting to 3 = Width of the bitmap data 4 = Height of the bitmap data
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD Bitmap Data Blit command is used to transfer/blit a raw pixel map of a specified width and height to an (x,y) location on the OSD display. Note that if the source is a BMP file, the address of the raw pixel data inside the BMP must be specified, and not the base address of the BMP itself.

OSD_BMPDATA_FILL

Command	Q_SYS_CMD_OSD_BMPDATA_FILL
Arguments	0 = Fill data value 1 = xAddress of the OSD screen to which bitmap is blitting to. 2 = yAddress of the OSD screen to which bitmap is blitting to. 3 = Width of the fill rectangle 4 = Height of the fill rectangle
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD Bitmap Data Blit Fill command is used to fill a rectangle of size (width x height) in the OSD screen at location (xaddr, yaddr) with the value specified.

BMP_SHOW

Command	Q_SYS_CMD_OSD_BMP_SHOW
Arguments	0 = Mode (0 to disable, 1 to enable) 1 = xAddress of the video display window 2 = yAddress of the video display window
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	The OSD Show command is used to show or hide the OSD display on the screen. The command takes an (x,y) address, which is the coordinate relative to the top-left corner of the display, to display the screen.

10.4.6 Double-Buffered Configuration Commands

The system control object manages a set of double buffered configuration parameters that are set using a dedicated configuration command. The set of double buffered parameters are then activated in the MG1264 Codec using an activate command.

SET_OUTPUT_PARAM

Command	Q_AVE_CMD_SET_OUTPUT_PARAM
Arguments	0 = Parameter 0 1 = Value 0 2 = Parameter 1 or 0 3 = Value 1 4 = Parameter 2 or 0 5 = Value 2
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This parameter sets a double buffered video output parameter. Up to three parameters and their associated value can be set by a single command. Once a parameter is set, it has to be forcibly activated by sending the Q_AVE_CMD_ACTIVATE_OUTPUT_CFG command. When this command is sent, all pending parameters are activated.

ACTIVATE_OUTPUT_CFG

Command	Q__AVE_CMD_ACTIVATE_OUTPUT_CFG
Arguments	None
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command activates all pending parameters set by the SET_OUTPUT_PARAM command since the last time either ACTIVATE_OUTPUT_CFG was called.

10.4.7 Single-Buffered Configuration Parameters

AUDIO_NUM_CHANNELS

Parameter	Q_SYS_CFG_AUDIO_NUM_CHANNELS
Value	1 or 2
States	IDLE
Effective	On the next AV decoder or AV encoder state transition out of IDLE
Description	This parameter is to configure the number of input and output channels (stereo or mono).

AUDIO_SAMPLE_RATE

Parameter	Q_SYS_CFG_AUDIO_SAMPLE_RATE
Value	24000, 32000, 48000
States	IDLE
Effective	On the next AV decoder or AV encoder state transition out of IDLE
Description	This parameter configures the sampling rate of the system.

AUDIO_SAMPLE_SIZE

Parameter	Q_SYS_CFG_AUDIO_SAMPLE_SIZE
Value	16, 20, 24
States	IDLE
Effective	On the next AV decoder or AV encoder state transition out of IDLE
Description	This parameter configures the sampling size.

AUDIO_OUT_MASTER_CLOCK

Parameter	Q_SYS_CFG_AUDIO_OUT_MASTER_CLOCK
Value	1 = Q_SYS_CFP_AUDIO_OUT_MASTER_CLOCK_256FS 2 = Q_SYS_CFP_AUDIO_OUT_MASTER_CLOCK_512FS
States	IDLE
Effective	On the next AV decoder or AV encoder state transition out of IDLE
Description	This parameter configures the frequency of the audio output Master clock to either 256 times the sampling frequency or 512 times the sampling frequency.

AUDIO_OUT_SERIAL_MODE

Parameter	Q_SYS_CFG_AUDIO_OUT_SERIAL_MODE
Value	1 = Q_SYS_CFP_AUDIO_OUT_SERIAL_MODE_I2S 2 = Q_SYS_CFP_AUDIO_OUT_SERIAL_MODE_LEFT
States	IDLE
Effective	On the next AV decoder or AV encoder state transition out of IDLE
Description	This parameter configures the formatting of the audio output data to be either I^2S or left-justified.

10.4.8 Double-Buffered Output Parameters

These double buffered parameters are activated by the ACTIVATE_OUTPUT_CFG command. Until the activate command is sent, these parameters have no effect.

VID_0_ENABLE

Parameter	Q_SYS_CFG_VID_0_ENABLE
Value	0 = Disable 1= Enable
States	Any
Effective	Activation
Description	This parameter enables the display of video plane 0.

VID_1_ENABLE

Parameter	Q_SYS_CFG_VID_1_ENABLE
Value	0 = Disable 1= Enable
States	Any
Effective	Activation
Description	This parameter enables the display of video plane 1.

VID_0_SOURCE

Parameter	Q_SYS_CFG_VID_0_SOURCE
Value	Q_SYS_CFG_OUT_SOURCE_DECODER, Q_SYS_CFG_OUT_SOURCE_ENCODER, Q_SYS_CFG_OUT_SOURCE_NONE
States	Any
Effective	Activation
Description	This parameter controls which video source, either the encoder or the decoder is displayed on video plane 0.

VID_1_SOURCE

Parameter	Q_SYS_CFG_VID_1_SOURCE
Value	Q_SYS_CFG_OUT_SOURCE_DECODER, Q_SYS_CFG_OUT_SOURCE_ENCODER, Q_SYS_CFG_OUT_SOURCE_NONE
States	Any
Effective	Activation
Description	This parameter controls which video source, either the encoder or the decoder is displayed on video plane 1.

VID_0_SCALING_ENABLE

Parameter	Q_SYS_CMP_OUTPUT_VID_0_SCALING_ENABLE
Value	0 = Disable the output scaler on video plane 0 1 = Enable the output scaler on video plane 0
States	Any
Effective	Activation
Description	This variable is used to enable/disable the video output scaler on video plane 0. When the scaler is enabled, it automatically resizes the decoded video to fit the plane's display rectangle.

VID_1_SCALING_ENABLE

Parameter	Q_SYS_CMP_OUTPUT_VID_1_SCALING_ENABLE
Value	0 = Disable the output scaler on video plane 1 1 = Enable the output scaler on video plane 1
States	Any
Effective	Activation
Description	This variable is used to enable/disable the video output scaler on video plane 1. When the scaler is enabled, it automatically resizes the decoded video to fit the plane's display rectangle.

VID_0_DISPLAY_WIDTH

Parameter	Q_SYS_CMP_OUTPUT_VID_0_DISPLAY_WIDTH
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the width of the display rectangle on video plane 0. When scaling is enabled, the scaler scales the video to match this width.

VID_0_DISPLAY_HEIGHT

Parameter	Q_SYS_CMP_OUTPUT_VID_0_DISPLAY_HEIGHT
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the height of the display rectangle on video plane 0. When scaling is enabled, the scaler scales the video to match this height.

VID_0_DISPLAY_OFFSET_X

Parameter	Q_SYS_CMP_OUTPUT_VID_0_DISPLAY_OFFSET_X
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the X position of video plane 0 relative to the start of active video.

VID_0_DISPLAY_OFFSET_Y

Parameter	Q_SYS_CMP_OUTPUT_VID_0_DISPLAY_OFFSET_Y
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the Y position of video plane 0 relative to the start of active video.

VID_1_DISPLAY_WIDTH

Parameter	Q_SYS_CMP_OUTPUT_VID_1_DISPLAY_WIDTH
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the width of the display rectangle on video plane 1. When scaling is enabled, the scaler scales the video to match this width.

VID_1_DISPLAY_HEIGHT

Parameter	Q_SYS_CMP_OUTPUT_VID_1_DISPLAY_HEIGHT
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the height of the display rectangle on video plane 1. When scaling is enabled, the scaler scales the video to match this height.

VID_1_DISPLAY_OFFSET_X

Parameter	Q_SYS_CMP_OUTPUT_VID_1_DISPLAY_OFFSET_X
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the X position of video plane 1 relative to the start of active video.

VID_1_DISPLAY_OFFSET_Y

Parameter	Q_SYS_CMP_OUTPUT_VID_1_DISPLAY_OFFSET_Y
Value	Positive non-zero integer
States	Any
Effective	Activation
Description	This variable configures the Y position of video plane 1 relative to the start of active video.

VID_0_ZOOM_SOURCE_SIZE

Parameter	Q_SYS_CMP_OUTPUT _VID_0_ZOOM_SOURCE_SIZE
Value	Source size as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	Video zoom is used to perform an arbitrary horizontal and vertical crop of the source and scale it to fit the display rectangle. The size, x offset and y offset are all specified as 16-bit fractions (such that 65536/2 is ½). This parameter is used to set the fractional size of the crop (note that zoom retains the same aspect ratio of the source so only scaling parameter is needed).

VID_0_ZOOM_SOURCE_OFFSET_X

Parameter	Q_SYS_CMP_OUTPUT_VID_0_ZOOM_OFFSET_X
Value	Source offset as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	This parameter is used to set the start x-offset as a fraction of the entire source.

VID_0_ZOOM_SOURCE_OFFSET_Y

Parameter	Q_SYS_CMP_OUTPUT_VID_0_ZOOM_OFFSET_Y
Value	Source offset as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	This parameter is used to set the start y-offset as a fraction of the entire source.

VID_1_ZOOM_SOURCE_SIZE

Parameter	Q_SYS_CMP_OUTPUT _VID_1_ZOOM_SOURCE_SIZE
Value	Source size as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	Video zoom is used to perform an arbitrary horizontal and vertical crop of the source and scale it to fit the display rectangle. The size, x offset and y offset are all specified as 16-bit fractions (such that 65536/2 is ½). This parameter is used to set the fractional size of the crop (note that zoom retains the same aspect ratio of the source so only scaling parameter is needed).

VID_1_ZOOM_SOURCE_OFFSET_X

Parameter	Q_SYS_CMP_OUTPUT_VID_1_ZOOM_OFFSET_X
Value	Source offset as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	This parameter is used to set the start x-offset as a fraction of the entire source.

VID_1_ZOOM_SOURCE_OFFSET_Y

Parameter	Q_SYS_CMP_OUTPUT_VID_1_ZOOM_OFFSET_Y
Value	Source offset as a 16-bit unsigned fraction
States	Any
Effective	Activation
Description	This parameter is used to set the start y-offset as a fraction of the entire source.

AUD_SOURCE

Parameter	Q_SYS_CMP_AUD_SOURCE
Value	Q_SYS_CMP_OUT_AUD_SOURCE_MAIN Q_SYS_CMP_OUT_AUD_SOURCE_PIP Q_SYS_CMP_OUT_AUD_SOURCE_ENCODER Q_SYS_CMP_OUT_AUD_SOURCE_DECODER
States	Any
Effective	Activation
Description	This parameter is used to select the active audio source. The options are the stream being displayed on the main window (video plane 0), the stream being displayed on the PIP window (video plane 1), or forced to follow the encoder or decoder.

10.4.9 Events

Q_SYS_EV_HEARTBEAT

Payload	None
Description	The heartbeat event is created once per second to indicate that the firmware is alive. The event can be used for bring-up and/or for debug purposes.

Q_SYS_EV_ECHO

Payload	0 = Value of the first argument to the corresponding ECHO command
Description	This event is created in response to the Q_SYS_CMD_ECHO command. The event has a single payload word that contains the value of the first argument to the ECHO command.

Q_SYS_VIDEO_OUTPUT_UNDERFLOW

Payload	None
Description	This event is created whenever the video display is ready for a new frame to be displayed but its input queue is empty. During decode, this is typically caused by a video decoder underflow.

Q_SYS_AUDIO_OUTPUT_UNDERFLOW

Payload	None
Description	This event is created whenever the audio output unit is ready for a new frame to be played but its input queue is empty. During decode, this is typically caused by an audio decoder underflow.

10.5 Status Block

The system control object maintains a status block that is typically used for bring-up and debug purposes. The structure of the block is:

```
typedef struct
{
    int             heartbeat;
    unsigned long   droppedEvents;
    unsigned long   evReadWritePtrs;
    int             pendingEvent;
} SYSTEM_CONTROL_STATUS;
```

10.5.1 heartbeat

The heartbeat field of the status block is periodically incremented by the command processor in the Media Processor firmware. The rate of increase is much faster than the rate of the heartbeat event.

10.5.2 droppedEvents

The droppedEvents field is incremented any time an event could not be posted to the internal event queue because the queue was full. Any dropped event is a serious condition and is considered a fatal error.

10.5.3 evReadWritePointers

This field stores the read and write pointers (indexes) into the internal event queue. The read pointer is the pointer used to send events to the System Host CPU, and the write pointer is the next location to be written with a new event. The read pointer is in the upper 16 bits and the write pointer is in the lower 16 bits. When the pointers are equal, the queue is empty, otherwise the full condition has the write pointer lagging behind the read pointer by one.

10.5.4 pendingEvent

This field indicates that the firmware has sent an event to the System Host CPU through the EVENT_READY interrupt and the System Host CPU has not yet acknowledged it. This field is typically used for bring-up and debugging of System Host CPU code where events could be unacknowledged, thus stopping event generation by the firmware.

10.6 H.264/ACC Decoder Interface Object

10.6.1 Overview

The H.264/AAC Decoder Interface object is responsible for controlling the H.264 Video Decoder, the AAC Decoder, and the demultiplexer as a combined entity. However, the object is sufficiently flexible to decode only video or audio streams, in both multiplexed and elementary formats.

The decoder and the video output unit work together to provide a set of trick play features that are comparable to those found in DVD players. This includes a full set of forward and backward smooth, slow motion, and scan modes. Additionally, the video output unit contains a scaler that can be used for PAL/NTSC/VGA conversion and arbitrary zoom.

10.6.2 Logical View of the AV Decoder

An idealized view of the decoder datapath is shown in Figure 10-5.

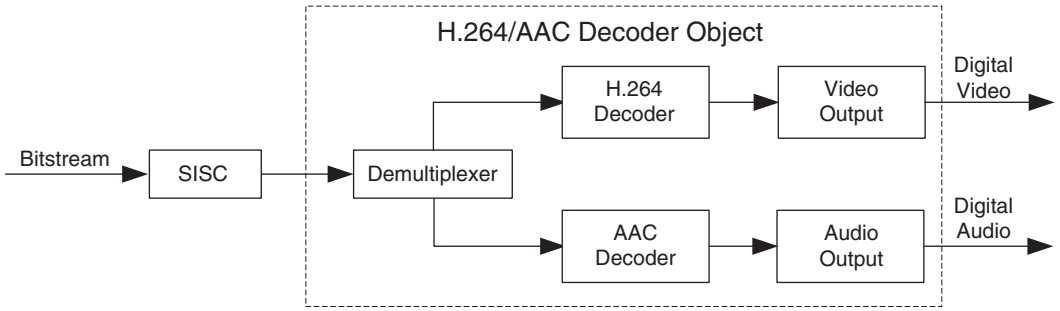


Figure 10-5 Idealized Decoder Datapath

This object takes compressed bitstreams as its input, and has a video output and audio output port. It is responsible for creating decoded 4:2:0 images at its video output port, and decoded PCM samples at its audio output port. The object contains five logical processing blocks:

- Demultiplexer
- AAC Decoder
- H.264 Decoder
- Video Output
- Audio Output

10.6.3 AV Decoder Features

Audio/Video Synchronization

Playback of audio or video streams is synchronized by the video and audio display units. The synchronization mechanism used is referred to as “Audio Master”. Audio Master means that the audio is played in a continuous fashion, while video frames are dropped or repeated as needed in order to achieve synchronization. The synchronization algorithm attempts to maintain synchronization timing of less than 1.5 video frame times (45 ms. in NTSC; 60 ms. in PAL).

There are situations where the system will run as “Video Master”. This includes playing streams with no audio, and doing trick play where the audio is decoded, but muted. The output units are

also programmed to smoothly switch from the Video Master mode during trick play to Audio Master mode in normal linear play.

The firmware has a programmable offset that can be used to skew audio or video timing. This offset is typically required when the video and audio datapaths have different delays. For example, a system may contain a video scaler where the incoming video is captured to memory and then scaled before sending to the MG1264 Codec, whereas the audio is sent out directly. In this situation, you have to program the offset to one frame time to allow for synchronized presentation, even with the extra frame delay in the video pipeline.

Programmable Pre-buffer

In situations where the data to be decoded is being received from a real-time source, it is often necessary for the decoder to pre-buffer a certain amount of data to ensure that it does not underflow at a later time due to variable bitrates produced by the encoder. The AV decoder can be programmed to have a specified amount of startup delay that can be matched from the encoded size.

Hardware/Software Flow Control

Both audio and video data is sent to the single bitstream port in the MG1264 Codec Host Interface. The demultiplexer reads bitstream data from this port and writes the video data to the video bit buffer and audio data to the audio bit buffer. The MG1264 Codec Host Interface features full hardware flow control either through a DMA request de-assertion for DMA operations, by asserting WAIT, or by delaying the ready bit during polling. This means that no data is lost if the MG1264 Codec cannot accept more data. Flow control is triggered any time either the audio or video buffers are completely full and new data is sent to the demultiplexer.

In some system designs, enabling the hardware flow control is not desirable because it locks the bus and prevents access to other devices on the same bus. In order to prevent this problem, the firmware provides commands that return the emptiness of both the video and audio buffers, which allows the System Host CPU to never send more data than is allowed in the buffer. The emptiness of the buffer is expressed both in bytes and in access units (frames). The System Host CPU must be careful not to send too many data bytes or too many access units that could trigger the hardware flow control.

Automatic Video Standard Conversion

The firmware supports the conversion of a bitstream from any of the supported video standards (PAL/NTSC/VGA) to the currently selected video standard. This conversion includes both spatial (vertical and horizontal scaling) and temporal scaling. The firmware uses a special algorithm for the frame rate conversion and does not rely on audio or video synchronization to do the frame rate conversion. This special algorithm results in a smoother presentation with fewer obvious dropped or repeated frames. Video standard conversion is automatic if a stream is detected that has been encoded differently from the current standard.

Arbitrary Video Zoom

The video output unit contains a scaler that can arbitrarily upscale an image to any resolution (the scaler can also downscale an image to fixed ratios such as 480/576 for PAL to NTSC standard conversion). The generalized upscaler is used to implement an arbitrary zoom feature where any part of the image (with the same aspect ratio as the display) can be cropped, and then zoomed to fit the full-display window.

Arbitrary zoom works for any ratios above 1.0 when the video is not having its standard converted. There is a limitation with zoom in PAL to NTSC where the video output unit is already downscaling the video with a ratio of 480/576. Since the generalized upscaler only works for ratios above 1.0, the smallest scaling ratio that is supported in PAL to NTSC is 576/480 = 1.2.

> **Note:** As of release 3.0 of the SDK, arbitrary zoom has been moved to the System Control object.

Trick Play

The firmware implements a complete set of trick play features that allow the System Host CPU to implement a natural user interface that offers the same user experience in both the forward and reverse directions. Specifically, forward and reverse singlestep, forward and reverse slow-motion, and forward and reverse smooth-scan (up to 4x) are offered. Additionally, the firmware can smoothly transition from any of these trick modes back to linear forward or reverse playback.

The System Host CPU is also free to implement higher speed trick play scans by sending only I-frames from specific GOPs. This technique allows for almost any speed of forward or reverse scan, at the expense of smoothness as a maximum of one frame per GOP is being decoded and displayed. The API supports a command that forces the firmware to decode and display only I-frames for a specified amount of frame times.

Trick play techniques are discussed in “Trick Play Techniques” on page 177.

10.6.4 Sending Encoded Bitstreams to the Decoder

Bitstream data is sent to the MG1264 Codec Host Interface bitstream device that, in turn, enters a FIFO called the System Input Stream Controller (SISC). From the input FIFO, the audio or video bitstream is demultiplexed into bitstream data and control data for both audio and video. The bitstream data is stored in a large FIFO and the control data is stored in a queue. The control data consists of one data structure per audio or video frame, and includes information such as timestamp, image size, and pointers to the associated bitstream data.

The hardware flow-control WAIT signal (also known as DMA_REQ) is generated by the input FIFO and is asserted anytime the FIFO becomes full. The input FIFO becomes full when any of the downstream queues or FIFOs become full. That is, if any of the video access unit queues, audio access unit queues, or the bitstream FIFOs become full, then WAIT will be asserted until the corresponding decoder removes data from the queue. The video decoder reads data at 29.97 Hz for NTSC and 25 Hz for PAL; the audio decode reads data every 1024 output samples (approximately 40 Hz at the 48 kHz sampling rate). Note that these rates can increase, decrease, or even stop due to trick play such as slow-motion, scan, or pause.

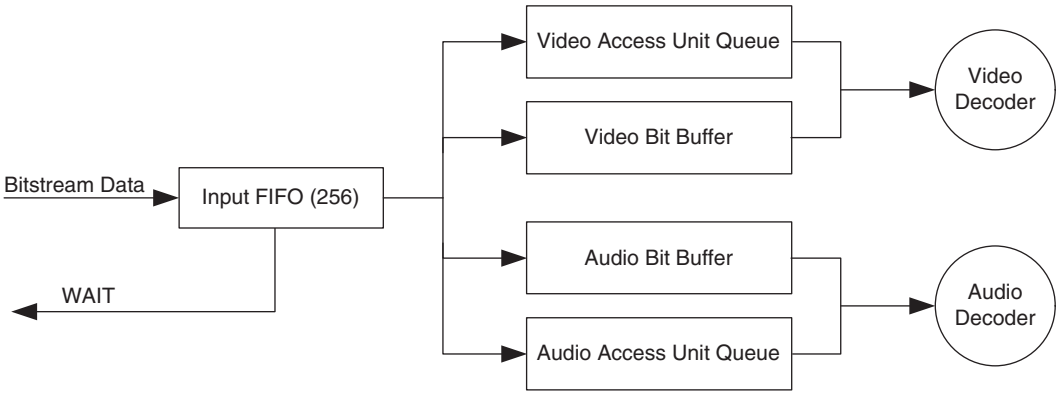


Figure 10-6 Decoder Buffer Structure

There are two types of bitstream transfer algorithms that can be selected by the System Host CPU. They are referred to as either a “Push” or a “Pull” model, and the model that is used is selected by the configuration parameter BITSTREAM_SOURCE.

In the push model, the System Host CPU does not care if the hardware flow control signal WAIT is asserted either because the bus is not shared, or if the bus can continue to be shared even if the transfer pauses. It is important to understand that during regular playback, either the audio or video buffer will be full almost all the time because the incoming data rate will be higher than the bitrate at which the bitstream was encoded. Which of the audio or video buffers becomes full depends upon the relative bitrates of the audio or video streams, as well as the sizes of the audio and video bit buffers.

In the pull model, the System Host CPU makes use of signaling from the firmware to ensure that the hardware flow control mechanism is never triggered for extended periods of time due to internal buffer fullness.

Push Transfer Model

If the System Host CPU can use the push transfer model, then transferring the bitstream is quite simple. The System Host CPU can open the QHAL_BS device and send as much or as little data to the MG1264 Codec as it wishes, as it does not care if the hardware flow control mechanism is triggered. Typical transfer logic (for forward playback and trick play) is similar to this:

```
bytesToSend = size of input file;
char localBuffer[BUFFER_SIZE];
while (bytesToSend != 0)
{
     bytesRead = read(inputfd, localBuffer, BUFFER_SIZE];
     qhalbs_write_bytes(handle, localBuffer, bytesRead);
     bytesToSend -= bytesRead;
}
```

Pull Transfer Model

In the pull transfer model, the System Host CPU sends data in such a way that the audio or video buffers never become full, and the hardware flow control signal is never asserted. This is also referred to as “Non-Blocking Operation”. This section shows sample code that can be used for non-blocking streaming.

The data streaming algorithm is fairly simple but does require the System Host CPU to parse the bitstream to identify audio and video data. For purposes of this algorithm, assume the bitstream consists of consecutive QBox structures. The key to the algorithm is that there are commands that query the firmware for video and audio buffer emptiness, both in terms of bytes and control structures. These commands are VIDEO_BUFFER_EMPTINESS and AUDIO_BUFFER_EMPTINESS as described in “Commands” on page 141.

Before sending data to the MG1264 Codec, the host should query the amount of space in both the audio and video buffers and then ensure that it does not send more data than there is space available before it checks for space again. Note that available space is expressed in two measurements. The first measurement is the amount of data in the compressed bitstream buffer. The second measurement is the number of spaces in the "access unit" queue.

For video streams an access unit is a NAL unit, for audio streams an access unit is a frame. Note that for audio streams a QBOX contains a single access unit but for video streams a QBOX can contain multiple NAL units that match a single presentation time. These NAL units include SEI timing messages and slice data. Therefore it is up to the host to count the number of NAL units in each QBOX before sending it so that it can compare the number of NAL units against the number of free entries in the video queue.

The algorithm (for forward playback and trick play only) is:

```
while!(end of file)
{
     // sleep 10ms here to allow the host to read some data

     // read the available space in each queue/FIFO
     videoQueueEmptiness = readVideoQueueEmptiness();
     videoFIFOEmptiness = readVideoBitstreamFIFOEmptiness();
     audioQueueEmptiness = readVideoQueueEmptiness();
     audioFIFOEmptiness = readAudioBitstreamFIFOEmptiness();

     while (1)
     {
       qboxSize = ParseNextQboxSize();
       qboxType = ParseNextQboxType();
       if (qboxType == VIDEO_QBOX)
       {
         // count the number of NAL units in the qbox
         nalus = GetNALUInQbox();
         if (videoFIFOEmptiness - qboxSize < 0)
         {
            break;
         }
         if (videoQueueEmptiness - nalus < 0)
         {
            break;
```

```
            }
            videoQueueEmptiness -= nalus;
            videoFIFOEmptiness -= qboxSize;
          }
          else if (qboxType == AUDIO_QBOX)
          {
            if (audioFIFOEmptiness - qboxSize < 0)
            {
               break;
            }
            if (audioQueueEmptiness == 0)
            {
               break;
            }
            videoQueueEmptiness--;
            videoFIFOEmptiness -= qboxSize;
          }

          // Send Qbox bytes to the codec using qhalbs_write

          // Move to next QBOX by adding qboxSize to the current
read pointer
}
```

10.6.5 Object ID

The H.264/AAC decoder object ID is 0x2.

10.6.6 State Machine

The AV decoder state machine consists of two parts linked by an IDLE state. The first part is the forward-play state machine and the second part is the reverse-play state machine. The only way to transition between the forward and reverse parts of the state machine is by transitioning to the IDLE state through the STOP command.

States

The decoder object has the following states:

Q_AVD_ST_IDLE: This is the startup state for the decoder, and the target state for the STOP command. No decoding is done in this state and all internal buffers are flushed. Transitions out of this state cause the decoder to restart decoding at the next I-frame. The last decoded frame is output by the video output hardware. The System Host CPU should put the system into an IDLE state for all bitstream discontinuities (such as changing from one file to another), or for switching between forward and reverse playback.

Q_AVD_ST_FLUSH: This state is an intermediate state between a playback state and IDLE. Because sending data to the MG1264 Codec involves hardware flow control, the data pipeline in the MG1264 Codec often needs to be flushed before stopping the bitstream transfer process on the System Host CPU. Once the System Host CPU has sent the FLUSH command it is free to use the STOP command to transition to IDLE.

Q_AVD_ST_FWDPLAY: This state performs continuous audio or video decoding and presentation. Additionally, frame rate and spatial conversion is performed as required if the input stream does not match the current video standard for the AV decoder.

Q_AVD_ST_FWDPAUSE: This state stops the video and audio decoder, and freezes the presentation at the last video and audio frames. No internal buffers are flushed so that a RESUME from the PAUSE state is completely seamless. The AV decoder can enter this state explicitly through the PAUSE command, or it can be entered automatically as part of a SINGLESTEP command once video decode and display are completed.

Q_AVD_ST_FWDSLOW: This state performs audio or video decoding, but at a rate that is slower than real time. Audio is decoded internally, but is muted due to discontinuities. Video frames are presented and deinterlaced (if necessary). Video and audio buffering remains synchronized, allowing for a seamless transition from Q_AVD_ST_FWDSLOW to Q_AVD_ST_FWDPLAY.

Q_AVD_ST_FWDPAUSE_WAIT: This is a temporary state that the decoder occupies from the time a SINGLESTEP command is issued to when the decoder has completed decoding and presenting the next frame. Once the decoding and presentation of this frame is complete, the decoder object automatically transitions to the Q_AVD_ST_FWDPAUSE state.

Q_AVD_ST_FWDIPLAY: This state performs video decoding of I-frames only. This state is used during fast-forward with the System Host CPU sending discontinuous parts of the bitstream. No audio decoding is done in this state, which prevents a seamless transition to the Q_AVD_ST_FWDPLAY state. Instead, the System Host CPU should transition to the other states via the Q_AVD_ST_IDLE state which resets the internal buffers.

Q_AVD_ST_FWDSCAN: This state decodes and displays of every Nth video frame to achieve a smooth fast-forward effect. Audio is decoded internally, but is muted due to discontinuities. Video and audio buffering remains synchronized allowing for a seamless transition from Q_AVD_ST_FWDSLOW to Q_AVD_ST_FWDPLAY.

Q_AVD_ST_BWDPLAY: This state performs continuous video decoding and presentation of frames in reverse order. No audio is decoded or presented in this state.

Q_AVD_ST_BWDPAUSE: This state stops the video decoder and freezes the presentation at the last video frame. No internal buffers are flushed so a RESUME from PAUSE is completely seamless. The AV decoder can enter this state explicitly through the PAUSE command, or automatically as part of a SINGLESTEP command once video decode and display are completed.

Q_AVD_ST_BWDSLOW: This state performs video decoding and presentation, but at a rate that is slower than real time. Video frames are presented and de-interlaced (if necessary).

Q_AVD_ST_BWDPAUSE_WAIT: This is a temporary state that the decoder occupies from the time a SINGLESTEP command is issued to when the decoder has completed decoding and presenting the previous frame. Once the decode and presentation of this frame is complete, the decoder object automatically transitions to the Q_AVD_ST_BWDPAUSE state.

Q_AVD_ST_BWDIPLAY: This state performs video decoding of I-frames only. It is used when performing fast-reverse with the System Host CPU sending discontinuous parts of the bitstream. The System Host CPU should transition to the other states via the Q_AVD_ST_IDLE state which resets the internal buffers.

Q_AVD_ST_BWDSCAN: This state performs video decoding and display of every Nth frame in order to achieve a smooth fast-reverse effect. The host must transition out of this state with a STOP command followed by a frame accurate PLAY.

State Transition Matrices

These matrices show the commands that can transition from one state to another. Note that several transitions are impossible and indicated by a (—) in the cell. Both forward and reverse matrices are shown. No direct state transitions are allowed from a FORWARD state to a REVERSE state, or vice versa. The starting state is shown in the left column, and the destination state is shown along the top row.

Table 10-1 Forward State

State	IDLE	FLUSH	PLAY	SLOW	IPLAY	PAUSE_WAIT	SCAN	PAUSE
IDLE	STOP	—	PLAY	—	—	PLAY	—	—
FLUSH	STOP	—	—	—	—	—	—	—
PLAY	STOP	FLUSH	—	SLOW	IFRAME_PLAY	STEP	SCAN	PAUSE
SLOW	STOP	FLUSH	RESUME	SLOW	—	STEP	—	PAUSE
IPLAY	STOP	FLUSH	—	—	—	—	—	PAUSE
PAUSE_WAIT	STOP	FLUSH	RESUME	SLOW	—	—	—	*Automatic*
SCAN	STOP	FLUSH	—	—	—	—	SCAN	—
PAUSE	STOP	FLUSH	RESUME	SLOW	—	STEP	—	—

Table 10-2 Backward State

State	IDLE	FLUSH	PLAY	SLOW	IPLAY	PAUSE_WAIT	SCAN	PAUSE
IDLE	STOP	—	PLAY	—	—	PLAY	—	—
FLUSH	STOP	—	—	—	—	—	—	—
PLAY	STOP	FLUSH	—	SLOW	IFRAME_PLAY	STEP	SCAN	PAUSE
SLOW	STOP	FLUSH	RESUME	SLOW	—	STEP	—	PAUSE
IPLAY	STOP	FLUSH	—	—	—	—	—	PAUSE
PAUSE_WAIT	STOP	FLUSH	RESUME	SLOW	—	—	—	*Automatic*
SCAN	STOP	FLUSH	—	—	—	—	SCAN	—
PAUSE	STOP	FLUSH	RESUME	SLOW	—	STEP	—	—

10.6.7 Commands

STOP

Command Name	Q_AVD_CMD_STOP
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command forcibly changes the state of the system to the IDLE state.

PLAY

Command Name	Q_AVD_CMD_PLAY
Arguments	0 = Play direction 1 = Start presentation time 2 = 0 for normal play, 1 to display first frame and pause
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	IDLE
Description	This command transitions the AV decoder to the FWDPLAY or BWDPLAY state, depending upon the value of the play direction argument. If the direction is 0, then the state is FWDPLAY; if the direction is 1, then the state is BWDPLAY. The second argument indicates a start presentation time. If this value is zero, then presentation starts at the first I-frame that is found in the stream. A non-zero value results in presentation starting at or later in the forward direction, or at or before in the reverse direction. This field is used to implement frame accurate trick play transitions that require a STOP command, such as switching between forward and reverse play, as well as from I-frame scan to normal playback. The third argument is a flag that indicates to the decoder that it should enter the PAUSE state immediately after displaying the first frame. This feature is required to implement a frame accurate single-step from the opposite direction. See “Trick Play Techniques” on page 177 for more information. As described in the state transition tables on page 165, the only states that can be entered from IDLE are the FWDPLAY and BWDPLAY states. Once in those states, the play direction is set and further transitions to SLOW, PAUSE, STEP, etc. can be done.

FLUSH

Command	Q_AVD_CMD_FLUSH
Arguments	None
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	IDLE
Description	The FLUSH command is used just prior to the STOP command. The purpose of the command is to clear out internal buffers that cause any bitstream sending to block.

I-FRAME_PLAY

Command	Q_AVD_CMD_IFRAME_PLAY
Arguments	Number of frame times to display each I-frame
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	PLAY
Description	The I-FRAME_PLAY command is used to transition the firmware to a state where only I-frames are decoded. All other frames are dropped. Each I-frame is decoded and then displayed for a number of frame times as specified by the first argument of the command. Because only I-frames are decoded, the same command is used for both forward and reverse playback. In order to transition to this state from IDLE, the System Host CPU must first send the PLAY command, and then immediately send the IFRAME_PLAY command before sending data.

PAUSE

Command Name	Q_AVD_CMD_PAUSE
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDPLAY, BWDPLAY, FWDSLOW, BWDSLOW, FWDSTEP, BWDSTEP, FWDSCAN, BWDSCAN
Description	The PAUSE command is used to transition the state into either FORWARD or REVERSE PAUSE. It is also entered automatically once a single-step operation has been completed.

IFRAME_PAUSE

Command Name	Q_AVD_CMD_IFRAME_PAUSE
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDPLAY, BWDPLAY, FWDSLOW, BWDSLOW, FWDSTEP, BWDSTEP, FWDSCAN, BWDSCAN
Description	The IFRAME_PAUSE command differs from the PAUSE command in that this command requests the AV decoder to enter the PAUSE state (either forward or backward) when the next I-frame is being displayed. The state of the AV decoder is not changed once this command is executed by the firmware. Instead, the AV decoder generates the event PAUSE_COMPLETE once the I-frame has been displayed and the PAUSE state has been entered.

SLOW

Command Name	Q_AVD_CMD_SLOW
Arguments	[0] Speed
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDPLAY, BWDPLAY, FWDSLOW, BWDSLOW, FWDSTEP, BWDSTEP, FWDSCAN, BWDSCAN, FWDPAUSE, BWDPAUSE
Description	The SLOW command is used to transition the state into either FORWARD or REVERSE SLOW MOTION. It is also used to change the slow motion speed once the SLOW MOTION state has been entered. The value of argument 0 is the inverse of the play speed such that a value of 3 is a 1/3 rate, 5 is 1/5, etc.

STEP

Command Name	Q_AVD_CMD_STEP
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDPLAY, BWDPLAY, FWDSLOW, BWDSLOW, FWDSCAN, BWDSCAN, FWDPAUSE, BWDPAUSE
Description	The STEP command is used to instruct the AV decoder to decode and display the next video frame and then automatically transition to either the FWDPAUSE or BWDPAUSE state (depending upon the current playback direction). The event PAUSE_COMPLETE is generated once this state transition has been performed.

RESUME

Command Name	Q_AVD_CMD_RESUME
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDSLOW, BWDSLOW, FWDSCAN, BWDSCAN, FWDPAUSE, BWDPAUSE
Description	The RESUME command is used to transition the AV decoder back to the FWDPLAY or BWDPLAY states in a smooth fashion while maintaining AV synchronization.

SMOOTH_SCAN

Command Name	Q_AVD_CMD_SMOOTH_SCAN
Arguments	0 = Speed
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	FWDSCAN, BWDSCAN, FWDPLAY, BWDPLAY
Description	The SMOOTH_SCAN command is used to perform smooth forward or reverse scans according to the speed specified in argument 0. Allowed speeds are 2 and 4.

SET_AUDIO_STREAM

Command Name	Q_AVD_CMD_SET_AUDIO_STREAM
Arguments	0 = Audio stream
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Any
Description	The SET_AUDIO_STREAM command is used to change the audio decode between allowed formats. It is implemented as a command rather than a configuration parameter since it takes effect immediately. The audio stream parameter can either be: 1 = PCM audio 2 = AAC audio

VIDEO_BUFFER_EMPTINESS

Command Name	Q_AVD_CMD_VIDEO_BUFFER_EMPTINESS
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	0 = Video buffer emptiness in bytes 1 = Video buffer emptiness in access units
Valid States	Any
Description	The VIDEO_BUFFER_EMPTINESS command is used by the System Host CPU to query the firmware about the emptiness of the video buffer. The firmware returns the emptiness in both bytes and access units (frames). The System Host CPU can use these values to ensure that it does not overflow the internal buffers during playback (thus triggering hardware flow control). Refer to “Sending Encoded Bitstreams to the Decoder” on page 159 for additional information.

AUDIO_BUFFER_EMPTINESS

Command Name	Q_AVD_CMD_AUDIO_BUFFER_EMPTINESS
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	0 = Audio buffer emptiness in bytes 1 = Audio buffer emptiness in access units
Valid States	Any
Description	The AUDIO_BUFFER_EMPTINESS command is used by the host to query the firmware for the emptiness of the audio buffer. The firmware returns the emptiness in both bytes and access units (frames). The host can use these values to ensure that it does not overflow the internal buffers (thus triggering hardware flow control) during playback. Refer to “Sending Encoded Bitstreams to the Decoder” on page 159 for additional information.

10.6.8 Configuration Parameters

These parameters can only be set when the decoder interface object is in the IDLE state and take effect on the next transition out of the IDLE state. The values assigned to the configuration parameters are persistent and are not reset by any state transition. They can only be changed by subsequent configuration commands.

BITSTREAM_TYPE

Parameter	Q_AVD_CFG_BITSTREAM_TYPE
Values	1 = Q_AVD_CFP_BITSTREAM_TYPE_ELEM_VIDEO 2 = Q_AVD_CFP_BITSTREAM_TYPE_QBOX
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This parameter is used to configure the decoder demultiplexing unit before bitstreams are sent to the decoder. This parameter must be setup when the system is in an IDLE state.

BITSTREAM_SOURCE

Parameter	Q_AVD_CFG_BITSTREAM_SOURCE
Values	1 = Q_AVD_CFP_BITSTREAM_SOURCE_SISC_PUSH 2 = Q_AVD_CFP_BITSTREAM_SOURCE_SISC_PULL
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This parameter is used to select the bitstream transfer method. This parameter must be set in IDLE state.

AV_SYNCH_ENABLE

Parameter	Q_AVD_CFG_AV_SYNCH_ENABLE
Values	0 or 1
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This parameter is used to enable or disable audio/video synchronization.

VIDEO_STC_OFFSET

Parameter	Q_AVD_CFG_VIDEO_STC_OFFSET
Values	Signed value representing 90 kHz ticks
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This parameter allows the System Host CPU to program a fixed offset between the video and audio streams in order to compensate for variable delays in the presentation datapath. For example, a system might capture and scale the video output, creating a one video frame delay relative to the audio. In this case, a negative offset of one frame (-3003 in NTSC) should be programmed.

VIDEO_OUTPUT_STANDARD

Parameter	Q_AVD_CFG_VIDEO_OUTPUT_STANDARD
Values	1 = Q_AVD_CFG_VIDEO_OUTPUT_STANDARD_NTSC 2 = Q_AVD_CFG_VIDEO_OUTPUT_STANDARD_PAL
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This parameter sets the video standard for the video output unit. Note that the video standard for the input unit can be different.

VIDEO_DECODE_FRAMERATE

Parameter	Q_AVD_CFG_DECODE_FRAMERATE
Values	32-bit value consisting of two 16-bit fields. Bits [31:16] are the integer frame rate and bits [15:0] are the fractional part.
States	IDLE
Effective	On the next AV decoder state transition out of IDLE.
Description	This variable is used to control the video decoder's frame rate. In normal full-frame rate video with audio, the frame rate is not used as the system is synchronized by the audio timing (Audio Master). However, the frame rate is needed whenever the system is running in Video Master mode, such as trick play. Additionally, it is used by the PAL <-> NTSC conversion code to do a smoother frame rate conversion than can be achieved solely by using audio or video synchronization. The frame rate is set using a 16-bit integer and a 16-bit fractional component. The two 16-bit values are sent as a single 32-bit configuration parameter. The upper 16 bits are the integer component and the lower 16 bits are the fractional. Consider the following examples:

Frame Rate in Hz	Value
30.0	0x1E0000
29.97	0x1DF851 (equivalent to 30000/1001)
25.0	0x190000
12.5	0xC8000

INIT_PREBUFFER

Parameter	Q_AVD_CFG_INIT_BUFFERING
Values	32 bit value of an integer number of frames
States	Idle
Effective	On next AV decoder state transition out of IDLE
Description	This parameter is used to set the initial prebuffering time of the AV decoder in frame times.

10.6.9 Decoder Configuration

The decoder also has a set of double-buffered configuration parameters that are set and then explicitly activated with an activation command. The command used to configure a parameter is Q_AVD_CMD_SET_VIDEO_ENC_PARAM (opcode 19). The command works exactly the same as the global CONFIGURE command except that it uses a different opcode. That is, it takes a list of zero-terminated configuration parameter/value pairs. The encoder configuration is activated using the Q_AVD_CMD_ACTIVATE_VIDEO_ENC_CFG command (opcode 20).

TICKS_PER_FRAME_DEFAULT

Parameter	Q_AVE_CFP_VIDEO_DEC_TICKS_PER_FRAME_DEFAULT
Values	Any non-zero value
States	IDLE
Effective	Q_AVD_ACTIVATE_VIDEO_DEC_CFG Command
Description	This parameter is used to set the native frame rate of the a video stream in the case it is an elementary video AVC stream without any SEI timing messages or VUI.

10.6.10 Events

Q_AVD_EV_VIDEO_DECODER_ERROR

Event	Q_AVD_EV_VIDEO_DECODER_ERROR
Payload	None
Description	This event is generated once for every video decoder error detected by the firmware.

Q_AVD_EV_AUDIO_DECODER_ERROR

Event	Q_AVD_EV_AUDIO_DECODER_ERROR
Payload	None
Description	This event is generated once for every audio decoder error detected by the firmware.

Q_AVD_EV_VIDEO_FRAME_DECODED

Event	Q_AVD_EV_VIDEO_FRAME_DECODED
Payload	None
Description	This event is generated once for every video frame decoded.

Q_AVD_EV_AUDIO_FRAME_DECODED

Event	Q_AVD_EV_AUDIO__FRAME_DECODED
Payload	None
Description	This event is generated once for every audio frame decoded.

Q_AVD_EV_VIDEO_PRESENTATION_COMPLETE

Event	Q_AVD_EV_VIDEO_PRESENTATION_COMPLETE
Payload	None
Description	This event is generated once the last frame in the video stream has been decoded and displayed.

Q_AVD_EV_AUDIO_PRESENTATION_COMPLETE

Event	Q_AVD_EV_AUDIO_PRESENTATION_COMPLETE
Payload	None
Description	This event is generated once the last frame in the audio stream has been decoded and sent from the output unit.

Q_AVD_EV_PAUSE_COMPLETE

Event	Q_AVD_EV_PAUSE_COMPLETE
Payload	None
Description	This event is generated when the MG1264 Codec transitions to the PAUSE state. The first way is through the PAUSE command. The second is through the System Host CPU issuing a SINGLESTEP command followed by the firmware completing the automatic state transition to PAUSE (forward or backward). The third way this event can be generated is through the IFRAME_PAUSE command, which delays the AV decoder transitioning to PAUSE until an I-frame is being displayed. The fourth way is PLAY with the pause trigger set. In all cases, this event is generated when the AV decoder completes the transition to PAUSE (forward or backward).

Q_AVD_EV_START_VIDEO_PRESENTATION

Event	Q_AVD_EV_START_VIDEO_PRESENTATION
Payload	None
Description	This event is generated once the first video from of a stream has been displayed. Until this event has been received, it can be assumed that the video display contains the last frame of the previous stream, or black if no streams have been played.

10.6.11 Status Block

The AV decoder object maintains a status block that can be polled by the System Host CPU at any time. The contents of the block are not synchronized with any event, and there is no indication from the firmware that an update has, or will occur.

```
typedef struct {
   uint32 videoFramesDecoded;
   uint32 audioFramesDecoded;
   uint32 videoDecoderErrors;
   uint32 audioDecoderErrors;
   uint16 videoBufferEmptiness;
   uint32 videoBufferAccessUnits;
   uint16 audioBufferEmptiness;
   uint32 audioBufferAccessUnits;
   uint32 videoPresentationTime;
   uint32 audioPresentationTime;
   uint32 avsyncVideoDrops;
   uint32 avsyncVideoRepeats;
} AVDecoderStatusBlock;
```

The fields in the status block are valid during audio or video decoding and presentation, and are reset when the AV decoder exits the IDLE state. Therefore, they remain valid after the STOP command has been issued, and represent the state of the AV decoder just prior to the STOP command being processed.

videoFramesDecoded

This field contains the number of video frames decoded since the last PLAY command.

audioFramesDecoded

This field contains the number of audio frames decoded since the last PLAY command.

videoDecoderErrors

This field contains the number of video decoding errors since the last PLAY command.

audioDecoderErrors

This field contains the number of audio decoding errors since the last PLAY command.

videoBufferEmptiness

This field contains the emptiness (total size-fullness) of the video bit buffer.

videoBufferAccessUnits

This field contains the number of available video buffer access units.

audioBufferEmptiness

This field contains the emptiness (total size-fullness) of the audio bit buffer.

audioBufferAccessUnits

This field contains the number of available audio buffer access units.

videoPresentationTime

This field contains the time of the most recently presented video access unit expressed in 90 kHz ticks.

audioPresentationTime

This field contains the time of the most recently presented audio access unit expressed in 90 kHz ticks.

avsyncVideoDrops

This field contains the number of video frames that were dropped (not displayed) due to audio or video synchronization requirements.

avsyncVideoRepeats

This field contains the number of video frames that were repeated due to audio or video synchronization requirements.

10.6.12 Trick Play Techniques

Implementing a complete set of trick play features requires careful system design of the System Host CPU code. The techniques used to implement these features can be divided into four categories:

1. "Forward Smooth Trick Play"
2: "I-Frame Trick Play"
3: "Reverse Trick Play"
4: "Switching Between Forward and Reverse Trick Play"

Forward Smooth Trick Play

Implementing forward trick play is the simplest of the four categories since it is most similar to linear playback where the audio or video data is sent to the MG1264 Codec in decode order. The only exception is doing I-frame only scans with jumps and that is dealt with in section "I-Frame Trick Play" on page 178.

Forward trick play modes are pause, singlestep, slow-motion, and scan. In all of these cases, the bitstream data is sent to the MG1264 Codec as if the MG1264 Codec is playing the data at regular speed. However, in trick play, the decoder either drops or repeats frames at various defined intervals in order to achieve the trick play effect. Pause, singlestep, and slow-motion place no additional burden on the System Host CPU since the data is being processed by the MG1264 Codec at a rate slower than real-time. The hardware flow control mechanism ensures that data is sent to the System Host CPU at the required rates, and the System Host CPU can continue to use the same data streaming algorithms that are used for linear playback.

Forward smooth scan is the most difficult of the trick modes since the decoder must drop frames in order to achieve a speed-up. However, since the video bitstream consists entirely of reference pictures (either I-frames or P-frames), the decoder must decode each picture of the GOP. The net effect is that the MG1264 Codec is limited to providing a 4x smooth scan. Also, note that the System Host CPU must be able to deliver the data to the MG1264 Codec at a 4x rate, meaning a 4 Mbit/sec stream is sent at 16 Mbit/sec.

All smooth forward trick play returns to the FWDPLAY state through the RESUME command. Audio or video synchronization is maintained across the trick play boundary without frame drops or repeats. The System Host CPU can go directly to an IDLE state by issuing a STOP command.

Note that the trick play states of SINGLESTEP, FWDSCAN, and FWDSLOW cannot be reached directly from the IDLE state. However, you can do slow and scan from IDLE by issuing a PLAY command followed by the SLOW or SCAN command BEFORE sending any bitstream data. You can perform a SINGLESTEP from IDLE by issuing the PLAY command with the pause trigger set (argument 2).

I-Frame Trick Play

An important limitation of smooth forward and reverse scan is that the System Host CPU must send data to the decoder at a rate equal to the scan rate multiplied by the video bitrate. These data rates from the System Host CPU may not be achievable for moderate-to-high video bitrates, making a 4x smooth scan impossible.

An alternative trick play technique, which is often used in DVD players, is to show I-frames only at the start of a GOP and to jump GOPs. Almost any rate of forward scan can be achieved by changing the jump distance between frames, however, these high rates come at the expense of smoothness.

A slight variation on this technique is to show a small number of frames at the start of the GOP in addition to the I-frame. These extra frames can provide the user with additional context beyond a still frame, and can still achieve high rates of scan.

The decoder state machine does not allow the RESUME command to be used in I-frame trick play to return to linear playback. This is because it is assumed that the System Host CPU is sending discontinuous bitstream data. Therefore, the only way out of I-frame trick play is through the STOP command. Once the STOP command is issued, the internal buffers of the decoder are flushed and playback can begin with the PLAY command.

However, it is important that the System Host CPU does not simply restart playback at the last I-frame sent to the decoder. Because the System Host CPU is sending only I-frames, a tremendous number of frames (and by extension, playback time) will be in the video bit buffer when the STOP command is issued. If data streaming resumed from the same point, the effect to the user would be a very large jump forward in time.

Instead, the System Host CPU should query the decoder for the current presentation time (by reading the *presentationTime* field in the AC decoder status block), and restart playback from the nearest GOP boundary matching that time.

Reverse Trick Play

Reverse trick play presents a challenge for the System Host CPU since it must send GOPs to the decoder in reverse order. Note that the data inside the GOP is sent in the traditional forward direction, it is only the order of the GOPs that must be reversed.

Reversing the order of the GOPs must be done using some type of random access information that the System Host CPU maintains. Typically, this is the random access information found in MP4 files, but can take the form of any metadata that the System Host CPU wishes to store.

No additional signaling is required by the System Host CPU when sending the GOPs in reverse. The System Host CPU must simply send the data in reverse GOP order.

Switching Between Forward and Reverse Trick Play

As can be seen from the State Transition Matrices, the only way to transition between forward and reverse playback is through the IDLE state, which means issuing a STOP command. This restriction makes it somewhat more difficult to implement common user operations such as forward singlestep, followed immediately by reverse singlestep. It is up to the System Host CPU to transition the decoder from IDLE to a trick play state in such a way that the user sees a seamless display of frames with no jumps or extraneous frames being displayed.

Transitioning between forward and reverse trick play requires the System Host CPU to do three general operations. The first step is to issue the STOP command to force the IDLE state. The second operation is to query the current presentation time from the decoder. Note that this presentation time can refer to any type of frame, either I-frame or P-frame. The third step is for the System Host CPU to start trick play in the other direction at the previous frame in the case of a forward to reverse switch, or to the next frame in the case of a reverse to forward switch.

Example: Forward slow-motion to reverse slow-motion proceeded by forward play:

1. Host receives user event signaling forward slow-motion
2: Host sends SLOW command
3: Host receives user event signaling reverse slow
4: Host sends the STOP command
5: Host reads the current video presentation time by reading the *videoPresentationTime* field in the AV decoder status block.
6: Host issues PLAY command indicating reverse direction, the current presentation time and with no pause trigger
7: Host issues SLOW command
8: Host identifies the byte position of the GOP which contains the current presentation time
9: Host sends the data starting at the GOP found in step 8

Example: Forward single-step to reverse single-step proceeded by forward play:

1. Host receives user event signaling forward single-step
2: Host sends the SINGLESTEP command
3: Host waits for and receives the PAUSE_COMPLETE event
4: Host receives user event signaling reverse single-step
5: Host sends the STOP command
6: Host reads the current video presentation time by reading the *videoPresentationTime* field in the AV decoder status block
7: Host issues a PLAY command indicating the reverse direction, the current presentation time and with the pause trigger set
8: Host identifies the byte position of the GOP that contains the current presentation time
9: Host sends the data starting at the GOP identified in step 8.
10: Host waits for and receives the PAUSE_COMPLETE event.

Bitstream Indexing

The MG1264 does not perform any indexing for the host code other than signaling in the QBOX header if the frame is an I-frame. It is up to the host to “index” (which typically means store a mapping from GOP number to byte position) the stream as it is recorded, and to send the bitstream in the correct order when being played back. Note that the popular MP4 file formats contains this mapping as part of the file's meta data, automatically making trick play implementation an issue of traversing this mapping and extracting the video data.

10.7 H.264/AAC Encoder Interface Object

10.7.1 Overview

The H.264/AAC encoder interface object is responsible for controlling both the H.264 and the AAC encoders as a combined entity. However, the object is sufficiently flexible to encode video-only or audio-only streams, in both multiplexed and elementary formats.

10.7.2 Logical View of the AV Encoder

An idealized view of the encoder datapath in coprocessor mode is shown in Figure 10-7.

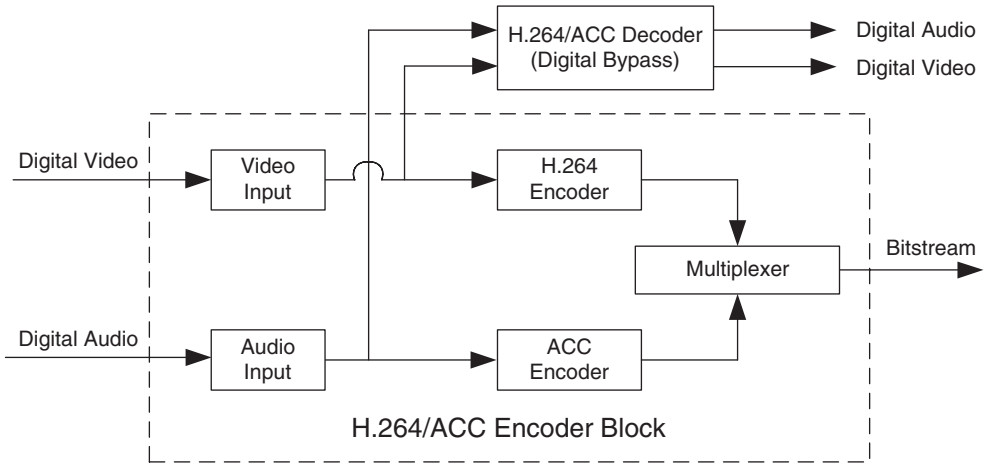


Figure 10-7 Idealized Encoder Datapath

The H.264/AAC encoder object takes in raw audio and video streams and produces a compressed bitstream. The object contains three logical functions

- H.264 Encoding
- AAC Encoding
- Multiplexing.

10.7.3 AV Encoder Features

Real-Time Encoding with Spatial and Temporal Scaling

The MG1264 Codec can perform real-time encode AVC raw video at resolutions of up to 800x600 at 30 frames per second, and scale to a minimum of 144 x 96. It can also encode AAC mono or stereo audio at sampling rates of up to 48 kHz at 16-bits per sample.

In addition, the video input block supports both spatial and temporal scaling. The horizontal or vertical resolutions can be halved independently to support resolutions such as 320x480, 352x480, 720x240, 720x576, 320x240, 352x240, and 352x288. Additionally, the video frame rate can be decimated to arbitrary frame rates, including less than one frame per second.

The minimum picture size that can be encoded is 96 x 96. The resolution can be obtained by either setting the capture rectangle to that resolution, or by scaling a larger capture rectangle to that resolution. See the crop and scaling commands for more information. However, note that

you must use one slice per macroblock row for any horizontal resolution below 128, meaning that pictures that are 112 or 96 pixels wide must use one slice per row. See "VENC_SLICES_PER_FRAME" on page 207. for more information.

Multiple Encoder Operational Profiles

The AVC encoder contains a number of algorithmic "tools" that are used to achieve either higher video quality or lower video bitrates. These tools come pre-configured in three sets of operational profiles. These profiles correspond to low, medium, and high bitrates. Low bitrates are considered to be <= 1.5 Mbps, medium are 1.5 to 3.5 Mbps, and high is 3.5 Mbps or greater.

Once an operational profile is set, the System Host CPU is free to select any video bitrate. The rate control algorithms in the MG1264 Codec will then use the selected toolset to match these bitrate requirements.

Controlling the Video Bitrate

The encoder allows the System Host CPU to specify an average video bitrate and runs three concurrent algorithms that are used to control the actual bitrate over time. These algorithms are short-term bitrate control, long-term bitrate control, and peak quality control. These algorithms work together to ensure that internal buffers are not overflowed, that the target file size is achieved, and bits are not wasted unnecessarily.

Field or Frame Video Encoding

The video input to the MG1264 Codec can be either progressively-scanned or interlace-scanned. In the case of progressive-scanned video, the encoder will produce a video sequence consisting entirely of frame pictures. However, if the video source is interlaced, the encoder will adaptively select between frame or field pictures depending upon the amount of motion in each frame. Adaptively choosing the picture coding type produces an important coding gain. This type of operation is called "Picture Adaptive Field/Frame".

Adaptive Frame Rate

The video rate control firmware module implements an adaptive frame rate algorithm that can be enabled for difficult content. As content gets more difficult, the rate control will typically raise the picture QP (quantization parameter) to meet its bit budget. However, the host can set a maximum QP then, when reached by the rate control, will result in the frame rate being lowered instead to hit the target bitrate. The host can also set a minimum frame rate where if the bit budget still cannot be met at this frame rate, the QP will be raised above the 'maximum'.

Hue/Saturation And Contrast Control

The MG1264 Codec's video input block has the ability to control the hue/saturation and contrast of the incoming video. These operations consist of a scaling and rotation of the chroma pixels in the chroma plane for hue and saturation control, and an arbitrary 256 entry lookup table for performing contrast enhancement.

Hue and Saturation modification are performed by performing by multiplying the 2-element chrominance vector by a 2x2 matrix to produce a new 2-element chrominance vector:

$$\begin{bmatrix} Cb' - 128 \\ Cr' - 128 \end{bmatrix} = \begin{bmatrix} S.\cos\theta & -S.\sin\theta \\ S.\sin\theta & S.\cos\theta \end{bmatrix} \begin{bmatrix} (Cb - 128) \\ (Cr - 128) \end{bmatrix}$$

Where S is the saturation (chroma gain, 1.0 = unity gain) and θ is the hue rotation angle.

In the VPU, the matrix values are generalized to:

$$\begin{bmatrix} Cb' - 128 \\ Cr' - 128 \end{bmatrix} = \begin{bmatrix} Ka & Kb \\ Kc & Kd \end{bmatrix} \begin{bmatrix} (Cb - 128) \\ (Cr - 128) \end{bmatrix}$$

Where Ka, Kb, Kc, and Kd are represented by 2.8 two's complement fixed point numbers. The firmware API allows the setting of the Ka, Kb, Kc, and Kd coefficients.

Text Overlay

The encoder has the ability of superimposing two 24 character strings onto the video prior to encoding. Each string can optionally have a frame counter that automatically increments. The counting range is configurable, allowing for arbitrary frame rates or NTSC drop-frame timecode to be implemented. A 16x16 one-bit-per-pixel font table is supported. A "1" bit in the character indicates that white (or black) is written to the video pixel, a "0" bit leaves the underlying video pixel unchanged.

Motion Alarms

The encoder has the ability to generate alarms depending on the amount of motion in the incoming video. The user can set "regions of interest" that consist of a set of blocks. The alarm is triggered when two thresholds are exceeded. The first threshold is the amount of motion in a block for the block to be considered in motion. The second threshold is the fractional amount of blocks in the region that have to be considered in motion. In this case, motion per block is calculated as the sum of absolute differences with its collocated macroblock in the previous frame divided by the number of pixels in the block.

10.7.4 Overview of the Video Encoding Process

The video encoding consists of a three-stage pipeline. Good understanding of each stage is required to get optimum performance from the encoder, both for quality and for latency.

The first stage is Video Capture. Video Capture refers to the process of taking video from the video input port, processing it and then storing it in memory for subsequent encoding. The codec architecture requires that an entire frame be captured before encoding begins, therefore creating a minimum one frame delay.

The second stage is Video Encoding. Video Encoding refers to the process of compressing the captured video frame into one or more slice NAL units, as well as generating the relevant SEI messages. All of these NAL units are sent to the multiplexer. However, the encoder can operate in a low latency mode in which the encoder and multiplexer send each slice NAL unit to the next pipe stage as soon as possible. If the encoder is configured to use multiple slices per picture this can reduce latency (note that when coding field pictures there are always at least two slices per picture).

At the maximum clock rate, the encoder is configured to use approximately 93% of each frame-time (assuming 29.97 fps for NTSC or 25 fps for PAL) so that each frame takes about 30ms to encode. The total latency from capture to encode is therefore one frame time + 30 ms. However, assuming that three slices are used per frame, the latency is one frame time + only 10 ms. The host then overlaps the fetching of the first slice NAL unit with the encoding of the second NAL unit.

The last stage is Multiplexing compressed video and audio into their buffers and coordinating bitstream transfer with the host processor as described in the next section. Multiplexing is done either at the frame level, or the slice level depending on the low latency configuration.

Video Capture

The video capture process it itself pipelined into four stages. The stages are synchronization, crop, scale, and store. When encoding interlaced sequences it is important to take into account the distinction between the temporal ordering and spatial (vertical) ordering of fields. The concept of top and bottom fields only has meaning when referring to vertical ordering. As indicated in Figure 10-8 and Figure 10-9, top or bottom fields may proceed from different temporal sampling times. It is customary to designate lines as “even” if they belong to even-numbered fields, and “odd” if they belong to odd-numbered fields. This document follows the ITU convention of starting line numbering at one.

The relationship between top/bottom fields when the top field is the older field in time is illustrated in Figure 10-8.

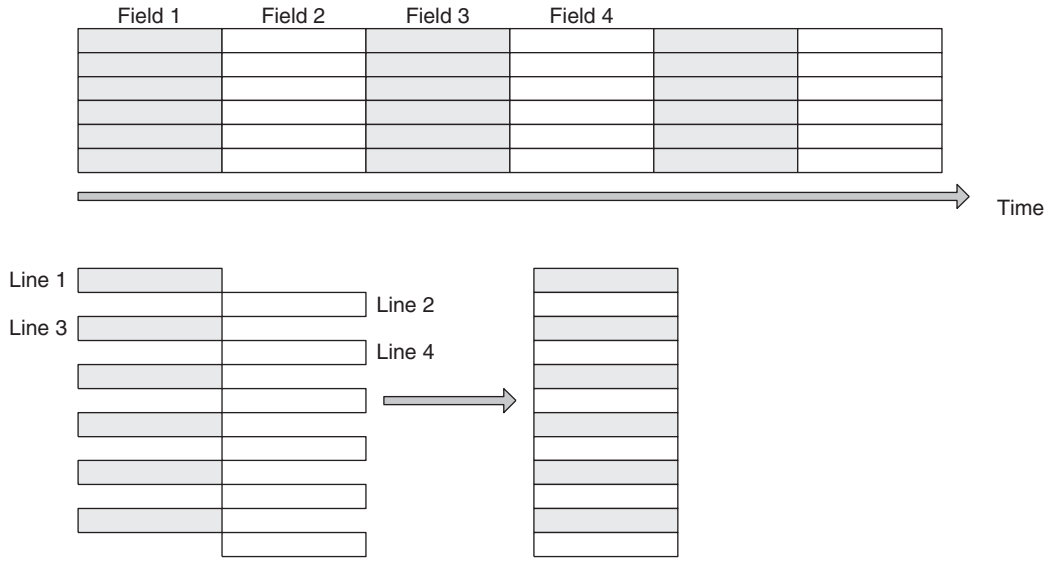


Figure 10-8 Top Field First

When the bottom field is the older field in time, the relationship is as shown in Figure 10-9:

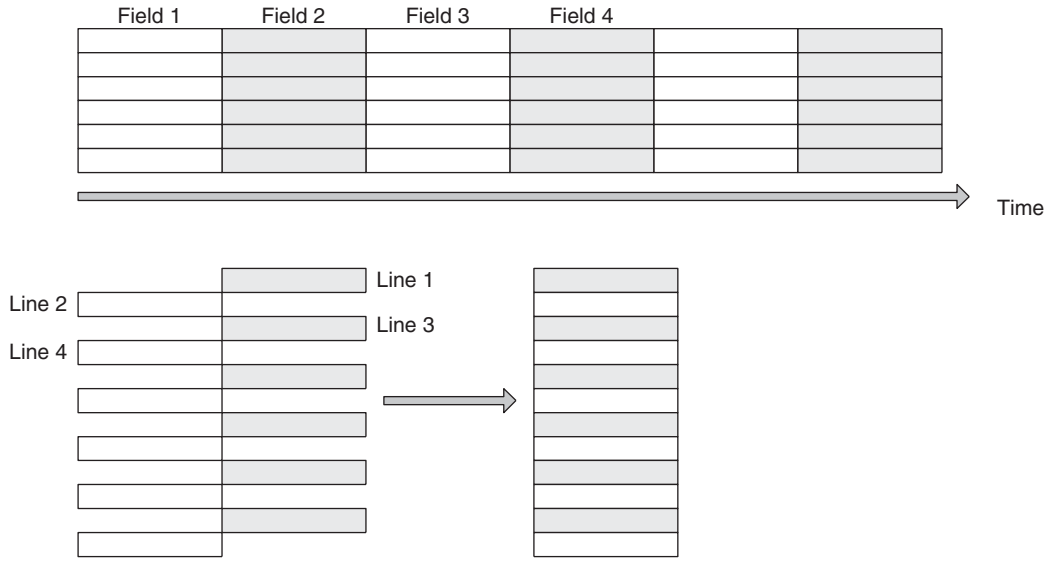


Figure 10-9 Bottom Field First

Synchronization

The first stage in the video capture process is synchronization. This involves identifying the first pixel in the active region of the incoming video. Two key configuration variables affect this operation. The first variable is progressive or interlace interface. Progressive interface refers to the transmission of video data to the MG1264 Codec line by line (as opposed to interlaced

where every other line is sent as a top and bottom field). In this mode, the F bit in the ITU-656 embedded sync code is ignored. This mode is the one typically used by CMOS sensors. Note that the maximum clock rate for video input is 40 Mhz.

The second configuration variable is internal or external sync signals. In internal sync, ITU-656 codes (typically referred to as end of active video/start of active video, or EAV/SAV codes) are used to identify horizontal, vertical blanking, as well as top and bottom fields. The ITU-656 specification defines the relative occurrence of vertical synchronization (V) and field identification (F) signals for standard television systems (Figure 9 and Figure 10). Notice from Figure 10-10 and Figure 10-11 that in the 525-line system (NTSC) the first line (in time) is a bottom field (even line 4); while in the 625-line system (PAL) the first line (in time) is a top field (odd line 1). However, the aforementioned relationship can be changed in external synchronization mode by programming the line on which active video starts for each field.

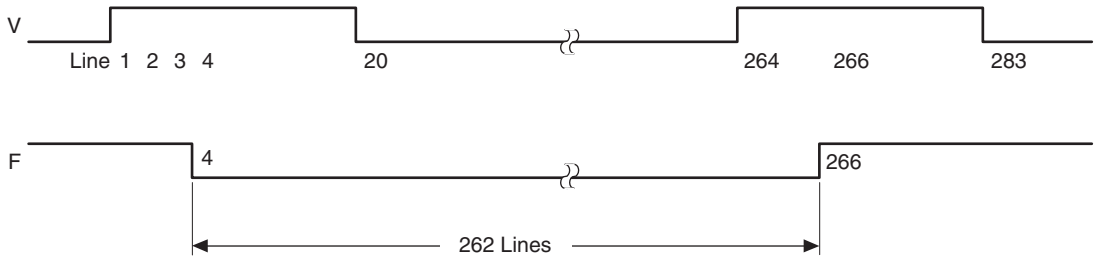


Figure 10-10 Synchronization 525-line System

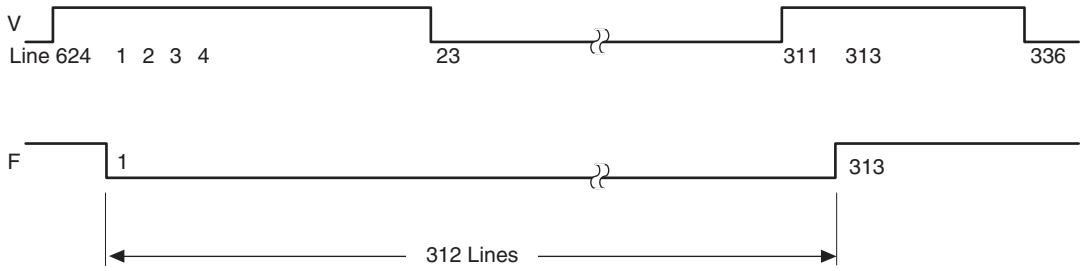


Figure 10-11 Synchronization 625-line System

A programmable vertical offset identifies the start of active video (the horizontal start of active video is defined by the SAV code). The beginning of active video is the first sample after SAV. There must be at least eight clock cycles between EAV and SAV codes.

When operating in external synchronization mode, the transition of the ITU-656 V signal from 0 to 1 indicates the start of vertical blanking. The value of F during this transition is the field number of the video field that has just ended (see Figure 10-10 and Figure 10-11). The first active lines in this mode are 20 and 283 when operating in the 525-line system; or 23 and 336 when operating in the 625-line system.

When operating in free-run synchronization mode, the first line of each field is programmable, but care should be taken to program these parameters so that the total number of lines in each field is consistent with the external synchronization signals. Furthermore, in free-run mode,

automatic lock mechanism is disabled. For example, there is no attempt made to automatically synchronize to any external signals coming into the MG1264 Codec. In this mode, the starting active video line number can be programmed for each odd or even field independently.

In external sync, there are separate horizontal, vertical, field signals that frame the pixel data, as well as programmable vertical, and horizontal offsets relative to hsync and vsync to identify the start of active video. In the case of interlaced video there are vertical offsets for both the top and bottom fields.

Lastly, a software state-machine ensures that video capture always starts with a top-field in the case of interlaced video. As is discussed later, top/bottom field pairs are captured together as a single frame to allow for a picture-adaptive field/frame coding algorithm to be employed. The sampling clock phase can be inverted in all synchronization modes. This feature is intended for non-standard systems that have a 180 degree phase difference between sampling clock and data.

Once the start of active video has been identified, pixels are sent to the crop stage of the pipeline.

Chroma Adjustments

It is possible to delay the Luma component by one sample time with respect to chroma. This may be necessary in systems that pair the collocated chroma with the even luma samples instead of the pairing indicated in ITU-601. Furthermore, it is possible to swap the order of Cb and Cr components for systems that do not implement the standard CbYCrY signal ordering.

Cropping

The crop operation is specified through the coordinates of the capture rectangle. The capture rectangle is the area of the video frame that is sent to the scaler. The rectangle is defined by a width, height and an (x,y) coordinate relative to the start of active video. Typically the capture rectangle is set to the be full resolution of the input signal (720x480 @ 0x0) but reducing the size of the rectangle and/or moving the origin of the rectangle can crop the video frame to a reduced resolution.

The capture rectangle is controlled through the set of Q_AVE_CMP_VIDEO_IN_CROP_* configuration parameters. The cropped video is then sent to the scaler stage of the pipeline.

Scaling

Video scaling is controlled through the specification of a scaling rectangle. The codec scales the captured video (as defined by its capture rectangle) to fill the scaling rectangle (Note that the MG1264 Codec hardware only supports downscaling of the capture rectangle). Additionally, 4:2:2 to 4:2:0 color space conversion is performed.

The scaling rectangle is set using the Q_AVE_CMP_VIDEO_IN_DECIMATION_H and Q_AVE_CMP_VIDEO_IN_DECIMATION_V configuration parameters.

There are some hardware constraints relating to scaling. When using vertical filtering and scaling there should be at least three lines of blanking in order to allow internal buffers to initialize properly. The final scaled line width should not exceed 800 pixels.

Storing

Video storage is a straight-forward process of storing the resultant 4:2:0 video frame to memory so that it can be sent to the video encoder.

Video Encoding

There are many variables that control the video encoding process, all of which can be found in the AVEncoder control object's API. However, there are a few key concepts that are identified here.

Entire video frames (which can consist of a top/bottom field pair) are sent to the video encoder. The encoder then decides to code the frame using frame coding or field coding. If the host indicates that the video is progressively scanned (note that this is different from a progressive interface) then the encoder will always use frame coding. However, if the video is identified as interlace scan (again, different from interlace interface) then the encoder will code the image either as a frame or field pair, depending on the amount of motion in the image (note that pure field coding can also be specified). Note that while top/bottom field pairs are always grouped, the encoder can mark the bitstream as being temporally bottom-field first using SEI picture structure messages.

A scene detection algorithm is run on the video frames to determine the picture coding type. If a scene change is detected, the frame is coded as either a P-frame with all intra blocks, or as an I-frame (controlled using the Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_ENABLE and Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_I_SLICE parameters). Otherwise, the picture coding type is controlled by the frame's position with the GOP (group of pictures) where the first frame is coded as an I-frame (note that for field coding only the top field is coded intra, the bottom field is coded inter).

The picture-level rate control algorithm then determines a base quantizer to use for the frame. The quantizer is selected to ensure bitrate, buffer fullness, and optimized for quality. Once the base quantizer is chosen the encoder starts to compress the frame. A macroblock-level rate control algorithm refines the choice of quantizer to more fully optimize for quality. New slice NAL units are started as specified by the host configuration.

Multiplexing

Multiplexing is the relatively simple operation of sending bitstream to the host as described in the subsequent section.

10.7.5 Receiving Encoded Bitstreams from the Encoder

Bitstream Transfer

The encoder produces a bitstream that is transferred between the firmware and the System Host CPU through commands, events, and memory transfers using the external memory interface in the MG1264 Codec Host Interface. Bitstream data is sent to the MG1264 Codec Host Interface in discrete “bitstream blocks”. Each bitstream block contains one access unit or QBox. The firmware maintains a set of bitstream blocks that are managed as a circular queue.

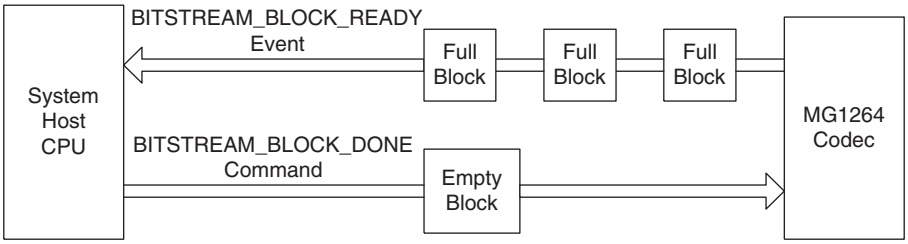


Figure 10-12 Circular Buffer Management of Bitstream Blocks

The availability of a new bitstream block is signaled by the BITSTREAM_BLOCK_READY event. In order for the System Host CPU to reduce the event rate, up to 6 bitstream blocks can be sent per event. The number of blocks that are sent per event is set using the AV encoder configuration parameter NUMBLOCKSPEREVENT.

When the encoder fills an event with the required number of bitstream blocks, the firmware signals to the System Host CPU that the new blocks are available through the BITSTREAM_BLOCK_READY event. The event payload contains the number of blocks, the start address of each block, and the size. The event also contains information about the type of bitstream; either AVC elementary video, AVC elementary audio, MP4, or QBox. In the case of QBox data, each bitstream block event can contain a mix of audio and video data. Note that once the System Host CPU has sent the FLUSH command, each bitstream block is sent with its own event (equivalent to setting NUMBLOCKSPEREVENT to 1) to ensure a proper bitstream flush.

When the System Host CPU receives the BITSTREAM_BLOCK_READY event, it must read the bitstream data from the MG1264 Codec memory and transfer it to the System Host CPU’s local memory. This is done using the QHAL function *qhalem_read_bytes*. **Do not use the function *qhalem_read_words* on bitstream data because that function corrects for endianess.**

Once the System Host CPU is through reading the bitstream data, it must send a command to the firmware to release the memory back to the encoder. This command is the BITSTREAM_BLOCK_DONE command, and has as arguments, the same information in the event (start address and size of the access unit). The firmware interprets the block address and determines if the command is referring to a video or audio block.

As a further optimization for QBox streams, the System Host CPU is only required to issue a BITSTREAM_BLOCK_DONE command for the last block of each type in the event. For example, if there are six blocks in the event consisting of three video blocks and three audio blocks, the System Host CPU can issue only one BITSTREAM_BLOCK_DONE for the last video block, and one BITSTREAM_BLOCK_DONE for the last audio block. This operation

requires the System Host CPU to parse the contents of each QBox to determine if the contents are audio or video, although presumably this is already being done in order to multiplex the bitstream data.

The event, Q_AVE_EV_BITSTREAM_BLOCK_READY, is represented by the following structure:

```
typedef struct
{
       CONTROLOBJECT_IDcontrolObjectId;
       EVENT_IDeventId;
     unsigned inttypeAndNumBlocks;
     unsigned intaddress0;
     unsigned intsize0;
     unsigned intaddress1;
     unsigned intsize1;
     unsigned intaddress2;
     unsigned intsize2;
     unsigned intaddress3;
     unsigned intsize3;
     unsigned intaddress4;
     unsigned intsize4;
     unsigned intaddress5;
     unsigned intsize5;
} STRUCT_Q_AVE_EV_BITSTREEAM_BLOCK_READY;
```

The field *typeAndNumBlocks* consists of two 16-bit fields. The upper 16 bits contain the bitstream type, and the lower 16 bits contain the number of blocks in the event. Bitstream types are the same as the parameter value set in the BITSTREAM_TYPE configuration parameter.

The command Q_AVE_CMD_BITSTREAM_BLOCK_DONE is created by copying the fields *frameAddress* and *frameSize* from the event structure. For example, given a pointer to the event block *event*:

```
COMMAND cmd;
cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
cmd.opcode = Q_AVE_CMD_BITSTREAM_BLOCK_DONE;
cmd.arguments[0] = event->frameAddress;
cmd.arguments[1] = event->frameSize;
```

The firmware can optionally pad each elementary stream sample (AVC video frame or AAC raw data block) to 4-byte alignment. This alignment is done using a private SEI NAL unit in the AVC and padding bits in the AAC. Creating a stream with 4-byte alignment can simplify System Host CPU multiplexing on systems that cannot do misaligned transfers on their 16-bit bus.

Bitstream Timing Information

Each video and audio frame is assigned a timestamp using an internal 90 kHz clock starting at time 0. This timestamp is always present in the QBOX header, and can be optionally stored in SEI picture timing messages in the elementary video stream. Additionally, the frame rate is stored in the H.264 VUI. The timestamps are separated by the sample duration, which is the reciprocal of the frame rate expressed in 90 kHz ticks. Because the MG1264 Codec supports variable frame rate, the delta between samples is not necessarily an integer number of frame or field times.

For audio frames, AAC codes 1024 samples per channel. For 24 kHz, this is (1024/24000) * 90000 = 3840 ticks per frame, for 48 kHz it is (1024/48000) * 90000 = 1920, and for 32 kHz it is (1024/32000) * 9000 = 2880.

NTSC video frames use timestamps using a frame time of 30000/1001 which is approximately 29.97. In terms of 90 kHz ticks, this is a frame time of 3003 ticks. PAL video frames use 3600 ticks per frame according to their 25 Hz frame rate.

10.7.6 Controlling the Video Bitrate

The MG1264 Codec has two versions of rate control that are optimized for different applications. These applications are storage and streaming. Only one algorithm can run at a time.

Rate Control for Storage Applications

Storage applications are distinguished by two-features. First, storage is finite, and managing the instantaneous bitrate is typically secondary to managing the total file size so as to guarantee a certain record time. Second, the sustained transfer rate of the storage medium is always higher than, and typically much higher than the average bitrate.

In that context, the rate control algorithm's primary goal is to maintain the file size so that it is as within a specified delta of nominal as is possible (as calculated by total time * average bitrate). For example, the host may want the total file size to be nominal ± 2 MBytes. The peak bitrate can be significantly higher than the average bitrate due to the high transfer rate of the storage channel.

The storage rate control algorithm has a decoder startup delay equal to the file size delta divided by the average bitrate. For example, if the average bitrate is 750 kbps and the delta is 1.5 MBytes, then the startup delay is the time required to buffer 1.5 MBytes. In a streaming application where the channel rate is close to the average bitrate then this corresponds to a two-second latency (1.5 MBytes/750 kbits). However, in storage applications where the sustained transfer rate is typically much higher than the average bitrate, the initial startup delay is much shorter. For example, for a transfer rate of 7.5 Mbps the initial delay is 200 ms. instead of 2 seconds.

Rate Control for Streaming Applications

Streaming applications differ from storage applications in both the speed of the data transfer (where the transfer rate is close to the average bitrate), and the continuous nature of the bitstream. In storage applications, the file is recorded in a "session" and the rate control manages the total size of the stream. In streaming applications, client decoders can start receiving data at any arbitrary I-frame and therefore the buffer state must be managed continuously.

Streaming rate control is implemented using a leaky-bucket model parameterized by the target average video bitrate, the actual video bitrate (the fill rate of the bucket), the speed of the transmit channel (the drain rate of the bucket), and the initial decoder delay (the fullness of the bucket before it starts to drain). Additionally, there is a low-delay mode that, when set, allows the rate control to momentarily underflow (run empty) the bucket with the trade off that subsequent frames must be dropped to allow the buffer to fill again. If low-delay is not set, then the rate control does not allow the buffer to underflow. The rate control manages the video bitrate in such a way that the bucket underflows either never, or as little as possible in a low-

delay mode, and never overflows while maintaining an average bitrate that is close to the target bitrate.

If the transmit channel rate is set higher than the target bitrate, then the long term average bitrate will tend to be larger than the target due to available slack created by the higher transfer rate. In order to generate a stream that has the long term average bitrate that matches the target, the size constraint should be used concurrently with the streaming rate control. The two rate control algorithms can run concurrently and use a voting scheme to ensure that the constraints set by each algorithm are met.

10.7.7 Using the Text Overlay

The encoder has the ability to apply a text overlay to the incoming video, along with an incrementing frame counter (this process is referred to as "burning" in the text to the video). There can only be two strings, each a maximum of 24 characters in length, and each of the strings can have an incrementing frame counter (limited to 0-99 maximum). The MG1264 Codec has no knowledge of the string itself and it is up to the host code to set the string properly.

Setting static strings is very simple and uses a typical set and enable design. However, a common feature is the implementation of a real-time clock with frame counter, which is slightly more complex. A key ability of the text overlay is to generate a "rollover" event each time the frame counter resets back. For example, if the frame counter is configured to count from 0-29, a rollover event will be generated each time it counts from 29 back to 0. Additionally, a string can be set by the host to be displayed by the codec only upon the next rollover.

The combination of these two features allows for an event-driven time display to be done. Specifically, the host is responsible for generating the time string (not including frames) and sending the string to the host each time there is a rollover event. Typically the host queries the current real-time and adds one second (assuming the frame count rolls over each second) and generates a string on the fly.

The API also allows for more sophisticated timings to be generated such as NTSC drop-frame timecode since the start frame number is configurable and not fixed to 0. The host can detect the drop-frame condition (typically each minute that is not divisible by 10) and set the start frame to 2 instead of 0.

10.7.8 Object ID

The H.264/AAC encoder object ID is 0x3.

10.7.9 State Machine

States

The H.264/AAC encoder object has the following states:

- **Q_AVE_ST_IDLE:** This is the startup state for the encoder. When in this state, the encoder is reset such that the first frame it generates will be an I-frame.
- **Q_AVE_ST_ENCODING:** This state performs continuous audio or video encoding with bitstream output to the System Host CPU.
- **Q_AVE_ST_PAUSE:** This state does not reset any of the encoder buffers, but prevents the encoder from creating new bitstream data. When the system returns to the ENCODING state, the first frame will be an I-frame.

- **Q_AVE_ST_FLUSHING:** This state is an intermediate state between Q_AVE_ST_ENCODING and Q_AVE_ST_IDLE. Unlike the decoder, the encoder cannot transition directly to IDLE from a non-IDLE state because the encoded data needs to be flushed. When this state is entered through the FLUSH command, the encoder stops creating new bitstream data. The encoder remains in this state until the System Host CPU acknowledges the receipt of the last bitstream block, after which the encoder automatically transitions to IDLE and sends the Q_AVE_EV_FLUSH_COMPLETE event.

State Transition Matrix

This matrix shows the commands that can be used to transition from one state to another. Note that several transitions are impossible and indicated by a (—) in the cell. The starting state is shown in the left column, and the destination state is shown along the top row.

State	IDLE	ENCODING	PAUSE	FLUSHING
IDLE	—	RECORD	—	—
ENCODING	—	—	PAUSE	FLUSH
PAUSE	—	RESUME	—	FLUSH
FLUSHING	(1)	—	—	—

1. This transition happens automatically when the bitstream has been flushed from the internal memory buffers to the System Host CPU.

10.7.10 Commands

FLUSH

Command Name	Q_AVE_CMD_FLUSH
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Q_AVE_ST_ENCODING and Q_AVE_ST_PAUSE
Description	This command changes the encoder's state to Q_AVE_ST_FLUSHING and stops the encoder from generating new bitstream data. Once transitioned to Q_AVE_ST_IDLE, the Q_AVE_EV_FLUSH_COMPLETE event is generated.

RECORD

Command Name	Q_AVE_CMD_RECORD
Arguments	0 = Enable preview
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Q_AVE_ST_IDLE
Description	This command changes the encoder's state to Q_AVE_ST_ENCODING and starts generating encoded data. If the enable preview argument is set to 1 then the input video will be echoed out the video port.

PAUSE

Command Name	Q_AVE_CMD_PAUSE
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Q_AVE_ST_ENCODING
Description	This command changes the encoder's state to Q_AVE_ST_PAUSE.

RESUME

Command Name	Q_AVE_CMD_RESUME
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Q_AVE_ST_PAUSE
Description	This command changes the encoder's state back to Q_AVE_ST_ENCODING and starts generating encoded audio or video data.

FORCE_NEW_GOP

Command Name	Q_AVE_CMD_FORCE_NEW_GOP
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command instructs the video encoder to start a new GOP immediately on the next frame.

BURNIN_INSERT_STR

Command Name	Q_AVE_CMD_BURNIN_INSERT_STR
Arguments	0 = (bits 31-24) String index of 0 or 1 0 = (bits 23-16) Update mode (0, 1, or 2) 0 = (bits 15-8) Offset into string 0 = (bits 7-0) Start counter 1 = Characters 0-3 for update 2 = Characters 4-7 for update 3 = Characters 8-11 for update 4 = Characters 12-15 for update 5 = Characters 16-19 for update
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command is used to update the string that is to be burned into the video. The command can be used to update the entire string, or a subset by changing the offset. The update mode selects how the string is to be displayed. If the update mode is 0 then the string is updated, but is not forced to be displayed. This mode is useful for updating strings of length greater than 20 as the command can only take 20 characters at a time. If the update mode is 1 then the string is displayed immediately. If the update mode is 2 then the string is updated immediately upon the next rollover by the frame counter. The start counter value is used to set the low-end of the frame count. Typically this is 0 but can be any non-zero value up to 99.

BURNIN_STR_SET

Command Name	Q_AVE_CMD_BURNIN_STR_SET
Arguments	0 = String index (0 or 1) 1 = 1 to enable, 0 to disable 2 = End counter 3 = X position for the string (multiple of 16) 4 = Y position for the string (multiple of 16)
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command is used to enable and set the location of the strings to be displayed. Additionally, the high value of the frame counter is set with this command.

BURNIN_FNUM_SET

Command Name	Q_AVE_CMD_BURNIN_FNUM_SET
Arguments	0 = String index (0 or 1) 1 = 1 to enable, 0 to disable 2 = String position
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command is used to enable the placement of the frame counter in a string and to set its position within the string. For example, if the position is 4 then the 2 character frame counter will be at the 4th character in the string.

BURNIN_FONT_SET

Command Name	Q_AVE_CMD_BURNIN_FONT_SET
Arguments	0 = Address of downloaded font table
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command is used to set an alternative font set that has already been downloaded to MG1264 Codec memory. Please consult Mobilygen for use of this feature.

MD_GLOBAL_RESET

Command Name	Q_AVE_CMD_MD_GLOBAL_RESET
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command resets the motion detection alarm system by clearing all regions of interest and thresholds.

MD_GLOBAL_REGION_SET

Command Name	Q_AVE_CMD_MD_REGION_SET
Arguments	0 = Region index 1 = Enable (1 for enable, 0 for disable) 2 = Motion threshold (0 - 255) 3 = Sensitivity threshold (percent 0 - 10000)
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command enables or disables a specific region of interest and sets the motion and sensitivity thresholds for that region.

MD_GLOBAL_REGION_ADD

Command Name	Q_AVE_CMD_MD_REGION_ADD
Arguments	0 = Region index 1 = X position (multiple of 16) 2 = Y position (multiple of 16) 3 = Width (multiple of 16) 4 = Height (multiple of 16)
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command adds a rectangular area to a specified region of interest. Note that a specific region of interest can be made of an arbitrary number of connected or disconnected blocks. The add region command takes a rectangle for convenience and multiple numberS of these commands can be used on a single region.

MD_GLOBAL_REGION_SUB

Command Name	Q_AVE_CMD_MD_REGION_SUB
Arguments	0 = Region index 1 = X position (multiple of 16) 2 = Y position (multiple of 16) 3 = Width (multiple of 16) 4 = Height (multiple of 16)
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command removes a rectangular area to a specified region of interest.

SET_GAMMA_LUT

Command Name	Q_AVE_CMD_SET_GAMMA_LUT
Arguments	0 = Table index (0-255) 1 = Number of entries to update (1-8) 2 = Y position (multiple of 16) 3 = Entry 0 (bits 0-7), entry 1 (bits 8-15), entry 2 (bits 16-23), entry 3 (bits 24-31) 4 = Entry 4 (bits 0-7), entry 5 (bits 8-15), entry 6 (bits 16-23), entry 7 (bits 24-31)
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command sets up to eight gamma look-up table entries.

SET_VIDEO_ENC_PARAM

Command Name	Q_AVE_CMD_SET_VIDEO_ENC_PARAM
Arguments	0 = Parameter 0 1 = Value 0 2 = Parameter 1 or 0 3 = Value 1 4 = Parameter 2 or 0 5 = Value 2
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This parameter sets a double buffered video encoder parameter. Up to three parameters, and their associated value can be set by a single command. Once a parameter is set, it has to be forcibly activated by sending the Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG command. When this command is sent, all pending parameters are activated.

ACTIVATE_VIDEO_ENC_CFG

Command Name	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command activates all pending parameters set by the SET_VIDEO_ENC_PARAM command since the last time either the RECORD or ACTIVATE_VIDEO_ENC_CFG commands were called.

SET_VIDEO_IN_PARAM

Command Name	Q_AVE_CMD_SET_VIDEO_IN_PARAM
Arguments	0 = Parameter 0 1 = Value 0 2 = Parameter 1 or 0 3 = Value 1 4 = Parameter 2 or 0 5 = Value 2
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This parameter sets a double-buffered video input parameter. Up to three parameters and their associated values can be set by a single command. Once a parameter is set, it has to be forcibly activated by sending the Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG command. When this command is sent, all pending parameters are activated.

ACTIVATE_VIDEO_IN_CFG

Command Name	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command activates all pending parameters set by the SET_VIDEO_IN_PARAM command since the last time either the RECORD or ACTIVATE_VIDEO_ENC_CFG commands were called.

SET_VIDEO_RC_PARAM

Command Name	Q_AVE_CMD_SET_VIDEO_RC_PARAM
Arguments	0 = Parameter 0 1 = Value 0 2 = Parameter 1 or 0 3 = Value 1 4= Parameter 2 or 0 5 = Value 2
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This parameter sets a double-buffered video rate control parameter. Up to three parameters and their associated values can be set by a single command. Once a parameter is set, it has to be forcibly activated by sending the Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG command. When this command is sent, all pending parameters are activated.

ACTIVATE_VIDEO_RC_CFG

Command Name	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG
Arguments	None
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command activates all pending parameters set by the SET_VIDEO_RC_PARAM command since the last time either the RECORD or ACTIVATE_VIDEO_ENC_CFG commands were called.

10.8 Single Buffered Configuration Parameters

These parameters can only be set when the encoder interface object is in an IDLE state and they take effect on the next transition out of the IDLE state. The values assigned to the configuration parameters are persistent and are not reset by any state transition. They can only be changed by subsequent configuration commands. All of these parameters are set using the Q_CMD_OPCODE_CONFIGURE command.

BITSTREAM_TYPE

Parameter	Q_AVE_CFG_BITSTREAM_TYPE
Value	1 = Q_AVE_CFP_BITSTREAM_TYPE_ELEM_VIDEO 2 = Q_AVE_CFP_BITSTREAM_TYPE_QBOX
States	IDLE
Effective	On the next AV encoder state transition out of IDLE.
Description	This parameter is used to configure the encoder multiplexing unit before bitstreams are sent to the System Host CPU. This parameter must be setup when the system is in the IDLE state.

NUMBLOCKSPEREVENT

Parameter	Q_AVE_CFG_NUMBLOCKSPEREVENT
Value	1 - 6
States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter is used to configure the number of bitstream blocks that are sent by the encoder per event.

INPUT_SELECT

Parameter	Q_AVE_CFG_ENC_INPUT_SELECT
Value	1 = Q_AVE_CFP_ENC_INPUT_SELECT_AV 2 = Q_AVE_CFP_ENC_INPUT_SELECT_VIDEO_ONLY 3 = Q_AVE_CFP_ENC_INPUT_SELECT_AUDIO_ONLY
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter controls the initialization of the video and audio encoder pipelines. This parameter should be used if there is no external hardware driving an audio or video interface (such as no audio clock). For example, a video only product would select VIDEO_ONLY.

AV_SELECT

Parameter	Q_AVE_CFG_ENC_AV_SELECT
Value	1 = Q_AVE_CFP_ENC_INPUT_SELECT_AV 2 = Q_AVE_CFP_ENC_INPUT_SELECT_VIDEO_ONLY 3 = Q_AVE_CFP_ENC_INPUT_SELECT_AUDIO_ONLY
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter selects either video-only encoding, audio-only encoding, or audio and video encoding. However, this parameter assumes that there is valid data present at the video and audio interfaces (such as a valid clock).

PREV_AV_SELECT

Parameter	Q_AVE_CFG_ENC_PREV_AV_SELECT
Value	1 = Q_AVE_CFP_ENC_PREV_AV_SELECT_AV 2 = Q_AVE_CFP_ENC_PREV_AV_SELECT_VIDEO_ONLY 3 = Q_AVE_CFP_ENC_PREV_AV_SELECT_AUDIO_ONLY
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter selects either video-only, audio-only encoding, or audio and video preview while encoding. Note that to disable preview completely (such as in an encode only product) a parameter to the RECORD command must be specified.

VENC_BITRATE

Parameter	Q_AVE_CFG_VENC_BITRATE
Value	Positive integer in bits per second
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter selects the target bitrate of the encoded video stream. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

AENC_ BITRATE

Parameter	Q_AVE_CFG_AENC_BITRATE
Value	Positive integer in bits per second
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter selects the long-term bitrate of the encoded audio stream.

VENC_FIELD_CODING

Parameter	Q_AVE_CFG_VENC_FIELD_CODING
Value	1 = Q_AVE_CFP_VENC_FIELD_CODING_FIELD 2 = Q_AVE_CFP_VENC_FIELD_CODING_FRAME 3 = Q_AVE_CFP_VENC_FIELD_CODING_ADAPTIVE
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	When the video source is interlaced (as indicated by the configuration variable VIN_PROG_SOURCE), this variable controls the picture coding type. The System Host CPU can select between all frame pictures, all field pictures, or adaptively select between field or frame pictures based upon the amount of motion observed in the two fields. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

VENC_GOP_SIZE

Parameter	Q_AVE_CFG_VENC_GOP_SIZE
Value	32-bit unsigned integer
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter sets the GOP size of the encoded video stream. The default value is 15 which means the GOP consists of one I-frame and 14 P-frames. A value of 1 indicates an all I-frame stream, and a value of 0 indicates a stream that consists of a single I-frame followed by P-frames. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

VENC_OPERATIONAL_MODE

Parameter	Q_AVE_CFG_VENC_OPERATIONAL_MODE
Value	0 = Q_AVE_CFG_VENC_OPERATIONAL_MODE_LOW_BITRATE 1 = Q_AVE_CFG_VENC_OPERATIONAL_MODE_MED_BITRATE 2 = Q_AVE_CFG_VENC_OPERATIONAL_MODE_HIGH_BITRATE
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter sets the general operational mode for the video encoder. It selects a collection of video encoding tools that are suitable to a particular bitrate range. The low bitrate toolset should be selected for bitrates <1.5 Mbps, the medium bitrate is suitable for the range 1.5 to 3.5 Mbps, and the high bitrate is suitable for rates greater than 3.5 Mbps. The System Host CPU must still explicitly select the target bitrate and set the rate control parameters. Setting this configuration parameter has the effect of resetting many other parameters. The System Host CPU should, therefore, be careful to set the operational mode first, and then set the remaining parameters.

AI_CHANNELS

Parameter	Q_AVE_CFG_AI_CHANNELS
Value	1 = Q_AVE_CFP_AI_CHANNELS_STEREO 2 = Q_AVE_CFP_AI_CHANNELS_STEREO_SWAP 3 = Q_AVE_CFP_AI_CHANNELS_STEREO_MONO_LEFT 4 = Q_AVE_CFP_AI_CHANNELS_STEREO_MONO_RIGHT
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter is used to direct a particular audio input channel configuration to the audio encoder. Note that this value should be consistent with the system control configuration parameter, AUDIO_NUM_CHANNELS, such that if the number of channels is 1, a mono configuration should be chosen. If the number of channels is 2, then either a mono or a stereo configuration can be chosen.

VIDEO_STC_OFFSET

Parameter	Q_AVE_CFG_VIDEO_STC_OFFSET
Value	Signed value representing 90 kHz ticks
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter allows the System Host CPU to program a fixed offset between the video and audio streams in order to compensate for variable delays in the input datapath. For example, a system might capture the video output and scale it creating a one video frame delay relative to the audio. In this case, a negative offset of one frame (-3003 in NTSC) should be programmed.

VIDEO_MUTE

Parameter	Q_AVE_CFG_VIDEO_MUTE
Value	0= Mute off 1= Mute on
Valid States	IDLE
Effective	Immediate if recording, otherwise on next transition out of IDLE
Description	This parameter is used to ‘mute’ the video input which results in an immediate fade to black, or black to full video. The AV encoder continues to run with both audio and video being encoded, although the encoded video frames will be black.

AUDIO_MUTE

Parameter	Q_AVE_CFG_AUDIO_MUTE
Value	0= Mute off 1= Mute on
Valid States	IDLE
Effective	Immediate if recording, otherwise on next transition out of IDLE
Description	This parameter is used to ‘mute’ the audio input which is results in an almost immediate fade to digital silence (the input signal is attenuated over 3 ms to ensure that there are no audio discontinuities), or from silence to full audio. The AV encoder continues to run with both audio and video encoded, although the encoded audio frames will be silent.

VENC_SLICES_PER_FRAME

Parameter	Q_AVE_CFG_VENC_SLICES_PER_FRAME
Value	1 - 6
Valid States	IDLE
Effective	On next transition out of IDLE
Description	This parameter is used to set the number of slices per encoded frame. Note that interlaced frames contain two slices by default. This parameter can be used to reduce encoder latency.

OUTSAMPLE_ALIGN

Parameter	Q_AVE_CFG_OUTSAMPLE_ALIGN
Value	0 = For no-padding to 4-byte alignment 1 = To align samples to 4-bytes
Valid States	IDLE
Effective	On next transition out of IDLE
Description	This parameter is used to force the AAC and AVC encoders to align their sample data to 4-byte boundaries. This alignment is done using a private SEI message for the AVC and using padding bits in the AAC.

10.9 Double-Buffered Video Encoder Parameters

The video encoder has a set of double-buffered parameters that are activated by the Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG command. The parameters are double-buffered because they are used during recording, and multiple parameters may need to be set at one time.

DEBLOCK _ENABLE

Parameter	Q_AVE_CMP_VIDEO_ENC_DEBLOCK_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter enables the in-loop de-blocking filter in the AVC encoder, and the bitstream.

DEBLOCK _OFFSET_ALPHA

Parameter	Q_AVE_CMP_VIDEO_ENC_DEBLOCK_OFFSET_ALPHA
Value	-5 to 5
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter sets the alpha coefficient of the de-blocking filter.

DEBLOCK _OFFSET_BETA

Parameter	Q_AVE_CMP_VIDEO_ENC_DEBLOCK_OFFSET_BETA
Value	-5 to 5
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter sets the beta coefficient of the de-blocking filter.

VUI_TIMING_ENABLE

Parameter	Q_AVE_CMP_VIDEO_ENC_VUI_TIMING_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter controls the presence of timing information in the AVC VUI structure.

SEI_PICT_TIMING_ENABLE

Parameter	Q_AVE_CMP_VIDEO_ENC_SEI_PICT_TIMING_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter controls the presence of SEI picture timing messages in the AVC bitstream.

SEI_ENC_CFG

Parameter	Q_AVE_CMP_VIDEO_ENC_SEI_CFG
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter controls the presence of private SEI messages that store the AVC encoder's configuration. This feature is typically used in bitstream analysis and debug.

SEI_RC_FRAME_STATS

Parameter	Q_AVE_CMP_VIDEO_ENC_SEI_RC_FRAME_STATS
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter controls the presence of private SEI messages that store the AVC encoder's frame-based statistics. This feature is typically used in bitstream analysis and debug.

SEI_RC_CALC_STATS

Parameter	Q_AVE_CMP_VIDEO_ENC_SEI_RC_CALC_STATS
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter controls the presence of private SEI messages that store the AVC encoder's calculated and derived frame-based statistics. This feature is typically used in bitstream analysis and debug.

SCENE_CHANGE_ENABLE

Parameter	Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter enables the scene detection algorithm. If the algorithm is enabled (by setting the parameter to 1), the encoder will either code a P-slice with all intra blocks, force an I-slice but not restart a GOP, or fully restart a new GOP with an I-slice and IDR picture. Which action is taken depends upon other configuration parameters.

SCENE_CHANGE_I_SLICE

Parameter	Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_I_SLICE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	If this parameter is set, the scene detection algorithm will force an I-slice at scene detection. If this parameter is not set, but the scene detection algorithm is enabled, then all intra blocks will be coded in a P-slice.

SCENE_CHANGE_NEW_GOP

Parameter	Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_NEW_GOP
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	If this parameter is set, then a new GOP (with IDR picture) is started when a scene change is detected.

SCENE_CHANGE_PERIOD

Parameter	Q_AVE_CMP_VIDEO_ENC_SCENE_CHANGE_PERIOD
Value	Positive value in ticks
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_ENC_CFG or IDLE to non-IDLE
Description	This parameter sets the minimum time between scene changes in ticks. For example, setting it to 30030 prevents a scene change for 10 frames from a previous scene change.

10.10 Double-Buffered Video Input Parameters

The video input block has a set of double-buffered parameters which are activated by the Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG command. The parameters are double-buffered because they ar used during recording and multiple parameters may need to be set at one time.

VIDEO_INPUT_STANDARD

Parameter	Q_AVE_CMP_VIDEO_INPUT_STANDARD
Value	1 = Q_AVE_CFP_VIDEO_INPUT_STANDARD_NTSC 2 = Q_AVE_CFP_VIDEO_INPUT_STANDARD_PAL
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter selects the video input standard to be either NTSC or PAL. The host must also set the correct capture rectangle using the VIDEO_IN_CROP_* configuration parameters and the number of ticks per frame (3003 or 3600) using the VIDEO_IN_TICKS_PER_FRAME configuration parameter.

VIDEO_IN_CROP_WIDTH

Parameter	Q_AVE_CMP_VIDEO_IN_CROP_WIDTH
Value	16 to 800 (multiples of 16)
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter sets the width of the crop rectangle relative to the start of active video. Typical values are 320, 352, 640 and 720.

VIDEO_IN_CROP_HEIGHT

Parameter	Q_AVE_CMP_VIDEO_IN_CROP_HEIGHT
Value	16 to 800 (multiples of 16)
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter sets the height of the crop rectangle relative to the start of active video. Typical values are 480 and 576.

VIDEO_IN_CROP_OFFSET_X

Parameter	Q_AVE_CMP_VIDEO_IN_CROP_OFFSET_X
Value	16 to 800 (multiples of 16)
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter sets the horizontal offset of the crop rectangle relative to the start of active video. The typical value is zero.

VIDEO_IN_CROP_OFFSET_Y

Parameter	Q_AVE_CMP_VIDEO_IN_CROP_OFFSET_Y
Value	16 to 800 (multiples of 16)
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter sets the vertical offset of the crop rectangle relative to the start of active video. The typical value is zero.

PROG_SOURCE

Parameter	Q_AVE_CMP_VIDEO_IN_PROG_SOURCE
Value	0 = Not progressive scanned 1 = Progressive Scanned
Valid States	Idle
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_IN_CFG or IDLE to non-IDLE
Description	This parameter indicates if the source video is progressive scanned. Note that a source can be progressively scanned even if it is an interlace interface as the top and bottom fields can be sampled at the same time. If the source is interlaced, then the VENC_FIELD_CODING parameter controls the picture coding type. **Note that the values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter**.

VIN_DECIMATION_H

Parameter	Q_AVE_CMP_VIDEO_IN_DECIMATION_H
Value	1 -16 or a value smaller than the actual horizontal image size
Valid States	Idle
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter sets the horizontal decimation ratio for the input stream. If the decimation ratio is in the range 1-16 then it is interpreted as a decimation ratio and frames will be horizontally scaled by that ratio to a multiple of 16 pixels. For example, on a 720 wide image setting the parameter to two will scale the image to 352 pixels. If the parameter is set to a value greater than sixteen, then it is interpreted as target pixel width and the image will be scaled to that width. For example, if the source is 720 wide and the value is 320, then the video will be scaled to 320 pixels wide.

VIN_DECIMATION_V

Parameter	Q_AVE_CMP_VIDEO_IN_DECIMATION_V
Value	1 -16 or a value smaller than the actual vertical image size
Valid States	Idle
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter sets the vertical decimation ratio for the input stream. If the decimation ratio is in the range 1-16, then it is interpreted as a decimation ratio and frames will be vertically scaled by that ratio to a multiple of 16 pixels. For example, on a 480 high image setting the parameter to two will scale the image to 240 pixels. If the parameter is set to a value greater than 16, then it is interpreted as target pixel height and the image will be scaled to that height. For example, if the source is 480 high and the value is 240 then the video will be scaled to 240 pixels high. The behavior of the scaler is affected by the setting of the INT_TO_PROG_SCALE parameter. If this parameter is set, then on interlace material where the vertical scaling ratio is two or greater the bottom field is dropped first (to achieve a 2x decimation) and further scaling is done on the top field only.

TICKS_PER_FRAME

Parameter	Q_AVE_CFG_VIDEO_IN_TICKS_PER_FRAME
Value	Any non-zero value
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to set the native frame rate of the video input hardware by setting the frame time in 90 kHz ticks. For NTSC, it should be set to 3003; for PAL it should be set to 3600.

TICKS_PER_OUTPUT_FRAME

Parameter	Q_AVE_CMP_VIDEO_IN_TICKS_PER_OUTPUT_FRAME
Value	Any non-zero value
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to set the frame rate of the bitstream. The native frame rate of the physical video input, set by VIN_TICKS_PER_FRAME (for example 3003 ticks in NTSC) is converted to the desired frame rate by dropping or repeating video frames. For example, if the ticks per output frame is set to 6006, then the frame rate is ½. If it is 9009, then its 1/3. Fractional frame rates are also supported by setting the frame length appropriately. For example, if the parameter is set to 4504, then the frame rate is (approximately) 20 fps. **Note that the values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.**

INT_TO_PROG_SCALE

Parameter	Q_AVE_CMP_VIDEO_IN_INT_TO_PROG_SCALE
Value	0 or 1
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to trigger vertical decimation of an interlace stream by dropping the bottom field instead of scaling. The effect will only be noticed if the vertical decimation (VIN_DECIMATION_V) is two or greater. Note that the end result is a progressive frame and frame coding should be selected. **Note that the values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.**

PIXEL_AR_X

Parameter	Q_AVE_CMP_VIDEO_IN_PIXEL_AR_X
Value	1 or greater
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to set the X value of the native video input pixel aspect ratio. This value is identical to those placed in the VUI (see table E-1 of ISO/IEC 14496-10 E). If the input video is not scaled and the pixel aspect ratio is identical to one of the standard (0-13) aspect ratios, then the VUI stores the index. If the aspect ratio is different, then the extended aspect ratio is used. Also note that the pixel aspect ratio stored in the VUI will be changed from the native aspect ratio if the input video is scaled AND the parameter VIDEO_IN_PIXEL_AR_FIXED is not set.

PIXEL_AR_Y

Parameter	Q_AVE_CMP_VIDEO_IN_PIXEL_AR_Y
Value	1 or greater
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to set the Y value of the native video input pixel aspect ratio. This value is identical to those placed in the VUI (see table E-1 of ISO/IEC 14496-10 E). If the input video is not scaled and the pixel aspect ratio is identical to one of the standard (0-13) aspect ratios, then the VUI stores the index. If the aspect ratio is different, then the extended aspect ratio is used. Also note that the pixel aspect ratio stored in the VUI will be changed from the native aspect ratio if the input video is scaled AND the parameter VIDEO_IN_PIXEL_AR_FIXED is not set.

PIXEL_AR_AR_FIXED

Parameter	Q_AVE_CMP_VIDEO_IN_PIXEL_AR_FIXED
Value	0 or 1
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to force the VUI to store the aspect ratio set by VIDEO_IN_PIXEL_AR_X and VIDEO_IN_PIXEL_AR_Y even if the input video is scaled. The VUI is forced if it is set to 1, otherwise the native aspect ratio is changed by the scalar.

FIELD_ORDER

Parameter	Q_AVE_CMP_VIDEO_IN_FIELD_ORDER
Value	1 = Q_AVE_CFP_VIDEO_INPUT_TOP_FIELD_FIRST 2 = Q_AVE_CFP_VIDEO_INPUT_BOTTOM_FIELD_FIRST
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter is used to indicate if the physical video input hardware sends fields that are temporally in a top to bottom order or a bottom to top order. Note that the temporal order may be different from the order that the hardware sends the data if the input hardware has a frame store.

HUE_SAT_CB_KA

Parameter	Q_AVE_CMP_VIDEO_IN_HUE_SAT_CB_KA
Value	10 bit fixed point in 2.8 two's complement
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter sets the Ka matrix element in the [2x2] hue/saturation matrix.

HUE_SAT_CB_KB

Parameter	Q_AVE_CMP_VIDEO_IN_HUE_SAT_CB_KB
Value	10 bit fixed point in 2.8 two's complement
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter sets the Kb matrix element in the [2x2] hue/saturation matrix.

HUE_SAT_CR_KC

Parameter	Q_AVE_CMP_VIDEO_IN_HUE_SAT_CB_KC
Value	10 bit fixed point in 2.8 two's complement
Valid States	Idle
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter sets the Kc matrix element in the [2x2] hue/saturation matrix.

HUE_SAT_CR_KD

Parameter	Q_AVE_CMP_VIDEO_IN_HUE_SAT_CB_KD
Value	10 bit fixed point in 2.8 two's complement
Valid States	Any
Effective	Command Q_AVE_ACTIVATE_VIDEO_IN_CFG
Description	This parameter sets the Kd matrix element in the [2x2] hue/saturation matrix.

10.11 Double-Buffered Video Rate Control Parameters

The video rate control has a set of double-buffered parameters which are activated by the Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG command. The parameters are double-buffered as they can be used during record and multiple parameters that may need to be set at one time.

SIZE_ENABLE

Parameter	Q_AVE_CFG_VIDEO_RC_SIZE_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter enables the rate control algorithm that manages the bitstream for total size.

SIZE_BIT_TOLERANCE

Parameter	Q_AVE_CFG_VIDEO_RC_SIZE_BIT_TOLERANCE
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the tolerance by which the rate control manages total file size. The file size will always be equal to the nominal file size (time multiplied by average bitrate) +/- a delta equal to this parameter. Note that this parameter is measured in bits, not bytes.

BUFFER_ENABLE

Parameter	Q_AVE_CFG_VIDEO_RC_BUFFER_ENABLE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter enables the rate control algorithm that manages the bitstream in streaming applications.

BUFFER_SIZE_BITS

Parameter	Q_AVE_CFG_VIDEO_RC_BUFFER_SIZE BITS
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the size of the HRD decoder buffer.

BUFFER_TRANSFER_RATE_BITS

Parameter	Q_AVE_CFG_VIDEO_RC_BUFFER_TRANSFER_RATE_BITS
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the transfer rate of the video bitstream or similarly, the fill rate of the HRD decoder buffer.

BUFFER_INITIAL_DELAY

Parameter	Q_AVE_CFG_VIDEO_RC_BUFFER_INITIAL_DELAY
Value	Positive value in 90 kHz ticks
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the initial delay of the decoder in 90 kHz ticks. For example, 200 ms is 18000.

BUFFER_LOW_DELAY_MODE

Parameter	Q_AVE_CFG_VIDEO_RC_BUFFER_LOW_DELAY_MODE
Value	0 or 1
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter enables low-delay mode in the streaming rate control. If set, the rate control is allowed to underflow the buffer at the expense of dropping frames to catch up.

ADAPTIVE_FRAMERATE_ENABLE

Parameter	Q_AVE_CFG_VIDEO_RC_ADAPTIVE_FRAMERATE_ENABLE
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter enables the adaptive frame rate algorithm. The algorithm must have the minimum and maximum QP set, as well as the frame rate scaling parameters set to be properly used.

AFR_MAX_QP

Parameter	Q_AVE_CFG_VIDEO_RC_AFR_MAX_QP
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the QP that, when reached by the rate control algorithm when raising the QP, will result in the frame rate being lowered instead.

AFR_MIN_QP

Parameter	Q_AVE_CFG_VIDEO_RC_AFR_MIN_QP
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter sets the QP that, when reached by the rate control algorithm when lowering the QP, will result in the frame rate being raised instead. Typically this QP is set to 2 or 3 below the AFR_MAX_QP.

AFR_SCALING_DENOMINATOR

Parameter	Q_AVE_CFG_VIDEO_RC_AFR_SCALING_DENOMINATOR
Value	Positive value
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter is used by the adaptive rate control to scale the frame rate. Assuming that complete frames will be dropped, this value should be set to the current maximum frame rate (rounding 29.97 to 30). If the current standard is NTSC, then typically this means it will be set to 30. However, if the frame rate has been reduced through the TICKS_PER_OUTPUT_FRAME parameter, then the reduced frame rate should be used.

AFR_SCALING_MIN_NUMERATOR

Parameter	Q_AVE_CFG_VIDEO_RC_AFR_SCALING_MIN_NUMERATOR
Value	Positive value in bits
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter is used to set the minimal frame rate where the adaptive frame rate algorithm starts to raise the QP above the ‘maximum’ specified. Typical values for NTSC would be 5, 10, 15, etc.

QP_RANGE_MAX

Parameter	Q_AVE_CFG_VIDEO_RC_QP_RANGE_MAX
Value	13-55
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter is used to set the maximum QP value selected by the rate control. Note that the actual QP per macroblock may go outside of this range based on the visual importance of that macroblock.

QP_RANGE_MIN

Parameter	Q_AVE_CFG_VIDEO_RC_QP_RANGE_MIN
Value	13-55
Valid States	Any
Effective	Q_AVE_CMD_ACTIVATE_VIDEO_RC_CFG or IDLE to non-IDLE
Description	This parameter is used to set the minimum QP value selected by the rate control. Note that the actual QP per macroblock may go outside of this range based on the visual importance of that macroblock.

10.12 Events

Q_AVE_EV_BITSTREAM_BLOCK_READY

Event	Q_AVE_EV_BITSTREAM_BLOCK_READY
Payload	0 = typeAndNumBlocks 1 = address0 2 = size0 3 = address1 4 = size1 5 = address2 6 = size2 7 = address3 8 = size3 9 = address4 10 = size4 11 = address5 12 = size5
Description	This event is generated once for every video and audio frame that is encoded. It is up to the System Host CPU to read the data in the block, store it, and then free it using the BITSTREAM_BLOCK_DONE command.

Q_AVE_EV_BITSTREAM_FLUSHED

Event	Q_AVE_EV_BITSTREAM_FLUSHED
Payload	None
Description	This event is generated once the last bitstream block in the internal memory buffers has been posted as an event in the event queue. It does not indicate that the System Host CPU has read the bitstream blocks, merely that the AV encoder object has transitioned to the IDLE state.

Q_AVE_EV_VIDEO_FRAME_ENCODED

Event	Q_AVE_EV_VIDEO_FRAME_ENCODED
Payload	None
Description	This event is generated once for every video frame that is encoded.

Q_AVE_EV_AUDIO_FRAME_ENCODED

Event	Q_AVE_EV_AUDIO_FRAME_ENCODED
Payload	None
Description	This event is generated once for every audio frame that is encoded.

Q_AVE_EV_VIDEO_FRAME_DROP

Event	Q_AVE_EV_VIDEO_FRAME_DROP
Payload	None
Description	This event is generated once for every video frame that is dropped by the video input unit due to drift between the audio and video clocks.

Q_AVE_EV_VIDEO_FRAME_REPEAT

Event	Q_AVE_EV_VIDEO_FRAME_REPEAT
Payload	None
Description	This event is generated once for every video frame that is repeated by the video input unit due to drift between the audio and video clocks.

Q_AVE_EV_BURNIN_ROVER

Event	Q_AVE_EV_BURNIN_ROVER
Payload	0 = String index 0 or 1
Description	This event is generated whenever the automatic frame counter "rolls over" from its maximum value to its minimum value.

Q_AVE_EV_VIDEO_MD_ALERT

Event	Q_AVE_EV_BURNIN_ROVER
Payload	0 = Region index 1 = Transition 2 = Average motion 3 = Number of macroblocks in motion
Description	This event is generated whenever a specific region of interest triggers a motion detection alarm, or when the region of interest no longer is in a state where it would trigger an alarm. When motion is first detected an alarm with a transition of 1 will be sent with the specified region index. When motion is no longer detected the same event will be sent with a transition of 0.

10.12.1 Average Motion Field

The average motion field contains the average motion in the region multiplied by 1000 (such that 32500 is 32.5) and the number of macroblocks in motion.

10.13 Status Block

The AV encoder objects maintains a status block that can be polled by the System Host CPU at any time. The contents of the block are not synchronized with any event, and there is no indication from the firmware that an update has, or will occur.

```
typedef struct {
    unsigned int    videoFramesEncoded;
    unsigned int    videoBufferEmptiness;
    unsigned int    videoBufferAccessUnits;
    unsigned int    reserved0;
    unsigned int    reserved1;
    unsigned int    audioFramesEncoded;
    unsigned int    audioBufferEmptiness;
    unsigned int    audioBufferAccessUnits;
} AVENCODER_STATUS;
```

The fields in the status block are valid during audio or video encoding, and are set when the AV encoder exits the IDLE state. Therefore, they remain valid after the FLUSH command has been issued, and represent the state of the AV encoder just prior to the FLUSH command being processed.

videoFramesEncoded

This field stores the number of video frames encoded since the last RECORD command.

videoBufferEmptiness

This field stores the current emptiness of the compressed video buffer.

videoBufferAccessUnits

This field stores the current number of access units in the compressed video buffer. The number of access units is incremented by one for each video-related BITSTREAM_BLOCK_READY event, and is decremented by one for every video-related BITSTREAM_BLOCK_DONE command.

audioFramesEncoded

This field stores the number of audio frames encoded since the last RECORD command.

audioBufferEmptiness

This field stores the current emptiness of the compressed audio buffer.

audioBufferAccessUnits

This field stores the current number of access units in the compressed audio buffer. The number of access units is incremented by one for each audio-related BITSTREAM_BLOCK_READY event, and is decremented by one for every audio-related BITSTREAM_BLOCK_DONE command.

Chapter 11. Sample Host Code Architecture

There are many design choices that can be made when architecting the host code needed to interface to the MG1264 Codec's firmware API. This chapter discusses the system characteristics that guide these choices.

How many, and how frequent are the events that will be subscribed? The event transfer protocol is fully handshaked and therefore prevents the sending of new events until the current event is processed. If the number of events that need to be handled is relatively small, then the host can wait until each event is completely handled before sending the EVENT_DONE acknowledgement. The theory is that the event processing time is approximately the same time as the interval between events, which results in no event queuing inside the MG1264 Codec memory. However, if there are many events to be examined, also at low latency then the host may have to acknowledge events as it receives them and not wait until they are processed.

Is bitstream storage fast or slow? When recording, the host must read the bitstream from the MG1264 Codec's memory and then store it to a file system. If this process is slow then BITSTREAM_BLOCK_READY processing is very slow.

How responsive does the UI need to be? In order to have a responsive system, the thread that sends commands to the firmware cannot be blocked for long periods of time.

A typical system will have relatively few events and a relatively slow file-system. The system will have relatively few events as only the BITSTREAM_BLOCK_DONE event needs to be subscribed during RECORD, and no events need to be subscribed (except for the VIDEO_DECODER_ERROR and AUDIO_DECODER_ERROR events, but they are rare) during decode. This means that event processing can take quite some time and still not require queuing. However, even though the number of events are low, the processing time for the BITSTREAM_BLOCK_READY event can be long due to the slow file system. This long processing time can result in blocking the UI if the system design is not done carefully.

A suitable architecture can be designed that uses a small number of threads, an interrupt handler and a sendCommand function.

sendCommand function: The sendCommand function is protected by a mutex that forces each command to be sent to the firmware, and the COMMAND_DONE acknowledgement to be received before processing a new command. When invoked, it sends a NEW_COMMAND message to the CommandEventHandler thread.

EventHandler thread: This thread manages the command and event transfer protocol and ensures that no protocol violations are performed. All events are read from this thread, and all commands are sent from this thread. The thread operates in an infinite loop and waits for either a NEW_INTERRUPT message or a NEW_COMMAND message.

BitstreamRecord thread: The BitstreamRecord thread is responsible for writing data to the Flash storage device, and also reading bitstream data from the MG1264 Codec. The input to the thread is a queue of transfer requests. A transfer request instructs the BitstreamRecord thread to read data of a specific size and address from the MG1264 Codec and store it to a file. The queue is written to by the CommandEventHandler thread when it receives a BITSTREAM_BLOCK_READY event during record.

BitstreamPlayback thread: The BitstreamPlayback threads is responsible for reading data from the Flash storage device and sending it to the MG1264 Codec. The input to the thread is a queue of transfer requests. A transfer request instructs the BitstreamPlayback thread to read data from a specific position and size from a file and send it to the MG1264 Codec. The queue is written by the UI thread. Note that for simple playback of an entire file, you do not need a queue of transfer requests, however, when it comes to streaming data which involves seeking in the bitstream (scan I-frame or reverse playback) the queue is helpful for optimizing performance.

UI Thread: The UI thread takes user input and translates it into calls to sendCommand and calls to the bitstream thread.

The complete architecture, along with sample code is described in the following sections. The thread API that is used is POSIX threads (called pthreads). Note that the code is simplified through extensive use of global static variables. A cleaner implementation would make use of object-oriented techniques.

11.1 Common Types and Definitions

These types are used throughout the reference code.

```
// host should always use memory partition 64 to read/write
memory
#define FWPARTITION 64
enum
{
     FIRST_BLOCK = 1,
     LAST_BLOCK = 2,
};
```

These definitions are related to the BitstreamRecord thread. Each BITSTREAM_BLOCK_READY event is translated into a write request for BitstreamThread to use.

```
typedef struct
{
     int blockType;
     int transfers;
     int address[6];
     int size[6];
} RECORD_REQUEST;
#define RECORD_QUEUE_SIZE 64
#define RECORD_BUFFER_SIZE (32768)
#define RECORD_BUFFER_PAD (4)
```

These definitions are related to the BitstreamPlayback thread.

```
typedef struct
{
     int blockType;
     int bytePosition;
     int size;
} PLAYBACK_REQUEST;
#define PLAYBACK_QUEUE_SIZE 64
#define PLAYBACK_BUFFER_SIZE (32768)
#define PLAYBACK_BUFFER_PAD (4)
```

11.2 Global Variables

These global variables are used by the command and event thread.

```
EVENT   localEvBlock;
pthread_id EvThreadId;
```

These global variables are used by the BitstreamRecord thread.

```
pthread_id bitstreamRecordThreadId;
pthread_mutex_t recordQueueMutex;
pthread_cond_t recordQueueCv;
int        recordQueueFullness;
int        recordQueueWrPtr;
int        recordQueueRdPtr;
char       recordBuffer[RECORD_BUFFER_SIZE +
RECORD_BUFFER_PAD];
```

These global variables are used by the BitstreamPlayback thread.

```
pthread_id bitstreamPlaybackThreadId;
pthread_mutex_t playbackQueueMutex;
pthread_cond_t playbackQueueCv;
int        playbackQueueFullness;
int        playbackQueueWrPtr;
int        playbackQueueRdPtr;
char       playbackBuffer[PLAYBACK_BUFFER_SIZE +
RECORD_BUFFER_PAD];
```

11.3 Initialization

All code is initialized by the call to fwInit. This function initializes all of the global variables and spawns all of the threads except the UI thread. It is assumed that the host application spawns this thread.

```
void fwInit()
{
     // Init command mutex
     pthread_mutex_init(&sendCommandMutex, NULL);
     hem = qhalem_open(QHALEM_ACCESSTYPE_CMD,
QHALEM_MODE_LINEAR);
     hmbox_ev = qhalmbox_open(QHAL_MBOX1);
     hmbox_cmd = qhalmbox_open(QHAL_MBOX1);
     // Lock the sendCommandMutex so that first
     // call blocks in sendCommand
     pthread_mutex_lock(&sendCommandMutex);
     // Init the BitstreamRecord thread variables
     recordQueueWrPtr = 0;
     recordQueueRdPtr = 0;
     recordQueueFullness = 0;
     pthread_mutex_init(&recordQueueMutex, NULL);
     pthread_cond_init(&recordQueueCv, NULL);
     hembs = qhalem_open(QHALEM_ACCESSTYPE_BITSTREAM,
QHALEM_MODE_LINEAR)
     // Init the BitstreamPlayback thread variables
     playbackQueueWrPtr = 0;
```

```
    playbackQueueRdPtr = 0;
    playbackQueueFullness = 0;
    pthread_mutex_init(&playbackQueueMutex, NULL);
    pthread_cond_init(&playbackQueueCv, NULL);
    hbs = qhalbs_open();
    // spawn command event thread
    pthread_create(&cmdEvThreadId, NULL, EvThreadProc, NULL);
    // spawn the bitstream record thread
    pthread_create(&bitstreamRecordThreadId, NULL, bitstream-
RecordThreadProc, NULL);
    // spawn the bitstream playback thread
    pthread_create(&bitstreamPlaybackThreadId, NULL, bit-
streamPlaybackThreadProc, NULL);
}
```

11.4 sendCommand function

This function is executed in the calling thread's context and blocks the calling thread until the command is received by the MG1264 Codec's firmware, and acknowledged with the COMMAND_DONE interrupt. It is protected by a global mutex that serializes the commands, and blocks until the COMMAND_DONE interrupt is received.

```
int sendCommand(COMMAND *cmd)
{
    int rval;
    QHALMBOX_EVENT mbs = QHALMBOX_EVENT_READ;
    // take global mutex
    pthread_mutex_lock(&sendCommandMutex);
    // copy command to codec memory
    qhalem_write_words(hem, FWPARTITION, cmdBlockAddr,
          cmd, sizeof(COMMAND)/4);
    // Signal command ready
    qhalmbox_write(hmbox_cmd, 1);
    // wait for command
    qhalmbox_wait_event(hmbox_cmd, &mbs);
    // copy the return code and values back to the cmd
    qhalmbox_read_words(hem, FWPARTITION, cmdBlockAddr,
          cmd, sizeof(COMMAND)/4);
    // the return code is return by the function
    rval = cmd->returnCode;
    // unlock global mutex
    pthread_mutex_unlock(&sendCommandMutex);
    return(rval);
    }
```

11.5 EventHandler Thread

The EventHandler thread waits for events through the *qhalmbox_wait_event*() call. When an event is received, the event block is fetched. Bitstream events are sent to the bitstream transfer thread while other events are handled in place.

```
int EvThreadProc(void *arg)
{
     QHALMBOX_EVENT mbs = QHALMBOX_EVENT_READ;
     unsigned int evBlockAddr, evAddr;
     EVENT localEvBlock;
     while (1)
     {
         // wait for event ready interrupt
         qhalmbox_wait_event(hmbox_ev, &mbs);
         // read the event pointer
         qhalem_read_words(hem, FWPARTITION, evBlockAddr,
           &evAddr, 1);
         // read the event
         qhalem_read_words(hem, FWPARTITION, evAddr,
           &localEvBlock, sizeof(EVENT)/4);
         // queue bitstream events
         if (localEvBlock.eventId ==
Q_AVE_EV_BITSTREAM_BLOCK_READY)

{

           RECORD_REQUEST rqst;
                   // read # of blocks in this event
           rqst.transfers = localEvBlock.payload[0] & 0xffff;
           for (int i = 0; I < rqst.transfers; i++)
           {
             rqst.address = localEvBlock.payload[2*i+1];
             rqst.size = localEvBlock.payload[2*i+2];
           }
           sendRecordRequest(&rqst);
         }
         // handle other events here as needed
         // send EVENT_DONE
         qhalmbox_read(hmbox, &rval);
       }
     }
}
```

11.6 BitstreamRecord thread

The BitstreamRecord thread is responsible for moving data from the MG1264 Codec to the storage device (Flash card). The input to the thread is a queue of data transfer requests. The data transfer request is similar to the BITSTREAM_BLOCK_READY event.

In this example, the BitstreamRecord thread stores a QBOX stream to a file exactly as it is sent by the MG1264 Codec. No parsing or multiplexing of the stream is done in any way. The interface to the thread is the sendRecordRequest function, which writes a transfer request to the queue. The thread reads a request from the queue, reads the data from the MG1264 Codec and stores it to a file. The file is opened or closed based on flags in the request structure that indicate if the request is the first or last block.

11.6.1 Writing a New Record Request to the Queue

The sendRecordRequest function copies in a transfer request to the queue and signals to the bitstream thread that there is a request to be read.

```
int sendRecordRequest(RECORD_REQUEST *rqst)
{
     // gain access to the queue
     pthread_mutex_lock(&recordQueueMutex);
     // copy the request to the queue
     bcopy(rqst, &(recordQueue[recordQueueWrPtr]),
sizeof(RECORD_REQUEST));
     // move the write pointer
     recordQueueWrPtr = recordQueueWrPtr++ %
RECORD_QUEUE_SIZE;
     // increment the fullness
     recordQueueFullness++;
     // signal the thread
     pthread_cond_signal(recordQueueCv);
     // unlock queue mutex
     pthread_mutex_unlock(&recordQueueMutex);
}
```

11.6.2 Reading a New Record Request from the Queue

The getRecordRequest blocks until there is at least one entry in the queue and then copies out a record request from the head of the queue.

```
int getRecordRequest(RECORD_REQUEST *rqst)
{
     // gain access to the queue
     pthread_mutex_lock(&recordQueueMutex);
     // wait for signal
     while (recordQueueFullness == 0)
     {
       pthread_cond_wait(recordQueueCv, recordQueueMutex);
     }
     // copy the request out of the queue
     bcopy(recordQueue[recordQueueWrPtr], rqst,
sizeof(RECORD_REQUEST));
     // move the write pointer
     recordQueueRdPtr = recordQueueRdPtr++ %
RECORD_QUEUE_SIZE;
```

```
        // decrement the fullness
        recordQueueFullness--;
        // unlock queue mutex
        pthread_mutex_unlock(&recordQueueMutex);
}
```

11.6.3 BitstreamRecord Thread Procedure

The bitstream thread procedure is quite simple. It reads a record request from the queue and transfers data from the MG1264 Codec and stores it in a file. The data is read from the MG1264 Codec into a local buffer and then written out from buffer. Multiple reads might be required per transfer request if the size is larger than the buffer size. Once the transfer request is done, a BITSTREAM_BLOCK_DONE is sent to the MG1264 Codec.

```
int bitstreamRecordThreadProc(void *arg)
{
     int fd;
     RECORD_REQUEST rqst;
     char filename = "test.qbx";
     COMMAND cmd;
     // init the bitstream block done command
     cmd.opcode = Q_AVE_CMD_BITSTREAM_BLOCK_DONE;
     cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
     while (1)
     {
       // block and wait for something to do
       readRecordRequest(&rqst);
       // if this is the first block open the file
       if (rqst->blockType | FIRST_BLOCK)
       {
         fd = open(filename, O_CREAT|O_TRUNC|O_WRONLY);
       }
       // transfer the data
       for (i = 0; i < rqst->transfers; i++)
       {
              // prepare for next transfer
         bytesLeft = rqst->size[i];
         currAddr = rqst->address[i];
         bytesWritten = 0;
         // transfer the data via an internal buffer
         while (bytesLeft != 0)
         {
           // read what is left in the transfer, or the
           // local buffer size, whichever is bigger
           bytesToRead = (bytesLeft > RECORD_BUFFER_SIZE:
                                RECORD_BUFFER_SIZE? bytesLeft);
           // pad the read out to the nearest 32 bits
           paddedBytesToRead = (bytesToRead + 3) & 0xfffffffc;
           // read the data
           qhalem_read_bytes(hembs, FWPARTITION, recordBuffer,
paddedBytesToRead);
           // Adjust bytesLeft for next run
           bytesLeft -= bytesToRead;
```

```
            currAddr += bytesToRead;
          }
          // acknowledge that this block is read
          cmd.arguments[0] = rqst->address[i];
          cmd.arguments[1] = rqst->size[i];
          sendCommand(&cmd);
        }
        // if this is the last block close the file
        if (rqst->blockType | LAST_BLOCK)
        {
          close(fd);
        }
      }
}
```

11.7 BitstreamPlayback thread

The BitstreamPlayback thread is responsible for moving data to the MG1264 Codec from the storage device (Flash card). The input to the thread is a queue of data transfer requests.

In this example, the BitstreamPlayback thread reads a QBOX stream to a file exactly as it is sent by the MG1264 Codec. No parsing or demultiplexing of the stream is done in any way. The interface to the thread is the sendPlaybackRequest function which writes a transfer request to the queue. The thread reads a request from the queue, reads the data from the file and sends it to the MG1264 Codec.

11.7.1 Writing a new playback request to the queue

The sendPlaybackRequest function copies in a transfer request to the queue and signals to the bitstream thread that there is a request to be read. The fields of the playback request have a byte position and size. These are useful when expanding the architecture to include random access, but for linear playback a byte position of -1 indicates that playback should continue from the current position in the stream.

```
int sendPlaybackRequest(PLAYBACK_REQUEST *rqst)
{
     // gain access to the queue
     pthread_mutex_lock(&playbackQueueMutex);
     // copy the request to the queue
     bcopy(rqst, &(playbackQueue[playbackQueueWrPtr]),
sizeof(PLAYBACK_REQUEST));
     // move the write pointer
     playbackQueueWrPtr = playbackQueueWrPtr++ %
PLAYBACK_QUEUE_SIZE;
     // increment the fullness
     playbackQueueFullness++;
     // signal the thread
     pthread_cond_signal(playbackQueueCv);
     // unlock queue mutex
     pthread_mutex_unlock(&playbackQueueMutex);
}
```

11.7.2 Reading a New Playback Request from the Queue

The getPlaybackRequest blocks until there is at least one entry in the queue and then copies out a record request from the head of the queue.

```
int getPlaybackRequest(PLAYBACK_REQUEST *rqst)
{
     // gain access to the queue
     pthread_mutex_lock(&playbackQueueMutex);
     // wait for signal
     while (playbackQueueFullness == 0)
     {
       pthread_cond_wait(playbackQueueCv, playbackQueueMutex);
     }
     // copy the request out of the queue
     bcopy(playbackQueue[playbackQueueWrPtr], rqst,
sizeof(PLAYBACK_REQUEST));
```

```
        // move the write pointer
        playbackQueueRdPtr = playbackQueueRdPtr++ %
PLAYBACK_QUEUE_SIZE;
        // decrement the fullness
        playbackQueueFullness--;
        // unlock queue mutex
        pthread_mutex_unlock(&playbackQueueMutex);
}
```

11.7.3 BitstreamPlayback Thread Procedure

```
int bitstreamPlaybackThreadProc(void *arg)
{
    int fd;
    PLAYBACK_REQUEST rqst;
    char filename = "test.qbx";
    int bytesToRead;
    int paddedBytesToSend;
    int bytesLeft;
    while (1)
    {
      // block and wait for something to do
      readPlaybackRequest(&rqst);
      // if this is the first block open the file
      if (rqst->blockType | FIRST_BLOCK)
      {
        fd = open(filename, O_RDONLY);
      }
      // set bytes to read, -1 means to end of file
      bytesLeft = rqst->size;
      // seek to position
      lseek(fd, rqst->bytePosition, SEEK_SET);
      while (bytesLeft > 0)
      {
        // read what is left in the transfer, or the
        // local buffer size, whichever is bigger
        bytesToRead = (bytesLeft > PLAYBACK_BUFFER_SIZE:
                          PLAYBACK_BUFFER_SIZE? bytesLeft);
        // read the data
        bytesRead = read(fd, buffer, bytesToRead);
        // if end of file get out
        if (bytesRead == 0)
        {
          break;
        }
        // We pad to the nearest 32 bits. Since the buffer
size
        // is already aligned to that, the only case where we
        // need to pad is at the end. It is ok to send extra
data
        // at the end of the stream.
        paddedBytesToSend = (bytesToRead+3) & 0xfffffffc;
        qhalbs_write(hbs, playbackBuffer, paddedBytesToSend);
        // Adjust bytesLeft for next run
        bytesLeft -= bytesToRead;
```

```
            }
            // if this is the last block close the file
            if (rqst->blockType | LAST_BLOCK)
            {
              close(fd);
            }
          }
        }
```

11.8 Sample Usage from UI thread

11.8.1 Simple Playback Session

It is assumed that the UI thread has received a request to playback a file from an external source, such as a keypress or IR driver. Playback of a file is as follows.

```
void UI_Play()
{
     COMMAND cmd;
     PLAYBACK_REQUEST rqst;
     // put the codec in PLAY state
     cmd.controlObjectId = AVDECODER_CTRLOBJ_ID;
     cmd.opcode = Q_AVD_CMD_PLAY;
     cmd.arguments[0] = 0; // forward
     cmd.arguments[1] = 0; // start time
     cmd.arguments[2] = 0; // no pause trigger
     sendCommand(&cmd);
     // start data streaming. We are sending whole file as one
block
     rqst.blockType = FIRST_BLOCK | LAST_BLOCK;
     rqst.size = -1;
     sendPlaybackRequest(&rqst);
}
```

11.8.2 Sample Record Session

It is assumed that the UI thread has received a request to record a file from an external source, such as a keypress or IR driver. Record of a file is as follows. Notice how no communication is needed with the BitstreamRecord thread as it is driven by the CmdEvThread automatically.

```
void UI_Record()
{
       COMMAND cmd;
     cmd.controlObjectId = AVENCODER_CTRLOBJ_ID;
     cmd.opcode = Q_AVE_CMD_RECORD;
     sendCommand(&cmd);
     }
```

11.9 Missing Features

This sample code is designed to show the basic flow of commands and events. It is not designed to be a complete system and therefore is missing a number of features. Some of these features are:

Stopping playback: The BitstreamPlayback thread has no way to do a fast stop operation.

End of record: The BitstreamRecord thread does not notify the UI that a flush operation has been completed. The thread should check for the LAST_BLOCK flag and then notify the UI when this block is stored to Flash memory.

Reverse playback: The BitstreamPlayback thread has a queue as its interface. In the sample session, only one request is sent per file which makes the queue extraneous. However, a full implementation would send each GOP to the decoder while in reverse play. This parsing could be done in the playback thread itself or outside by a another thread which is parsing the random access data structures for the stream.

Error handling: No error handling is done at all in this implementation.

Appendix A. MG1264 Codec H.264 and AAC Compliance

This appendix explains in detail how the MG1264 Low Power H.264 and AAC Codec for Mobile Devices complies with the H.264, and AAC standards. The subject of compliance is complex, yet manageable when addressed within the context of an application. The key to dealing with compliance is to find the balance between formal specification (including all of the corner cases that accompany all MPEG specifications) and real world implementations where most corner cases do not apply.

Compliance is generally addressed in terms of Profiles and the Tools associated with each Profile. The concept of Level is a further classification in H.264/MPEG, but Level represents specific combination of resolution, frame rate, and bitrate, details more related to performance than functionality.

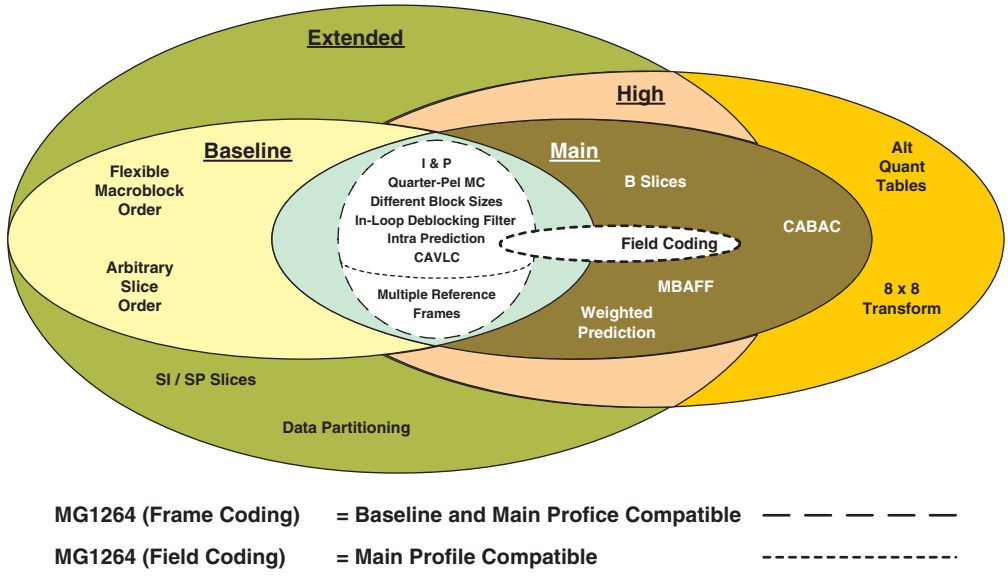


Figure A-1 H.264 Profiles and Tools

The MG1264 Codec H.264 codec is best described as a Baseline Profile codec (encoder and decoder). Technically, when the MG1264 Codec H.264 Encoder implements Field coding the bitstreams are Main Profile.

A.1 MG1264 Codec Encoder Compliance

Typically, when the subject of compliance is discussed what is meant is decoder compliance. MPEG, by definition, describes the bitstream syntax, and therefore the decoder must adhere to the complete specification to be considered “compliant”, and decode any combination of legal syntax.

Encoders are free to implement tools in any way that produces a syntactically correct bitstream. Due to implementation complexity, encoders always use a subset of the available tools, or a subset of the actual implementation of each tool. This is a key point that should not be underestimated because if a decoder will work with only a given encoder, or group of encoders, compliance testing on the decoder side can be simplified substantially.

A.1.1 MG1264 Codec H.264 Encoder Compliance

The MG1264 Codec H.264 Encoder has two modes of operation relative to compliance. These two modes are defined by the use of Frame (progressive) coding or Field (interlaced) coding. When either mode is used, the corresponding bitstreams produced are “compliant” to specific Profiles.

Frame Coding

When Frame (progressive) encoding is used, the MG1264 Codec H.264 Encoder produces streams that are fully compliant with the Baseline, Extended, Main, and High Profiles. Refer to Figure A-1. The MG1264 Codec H.264 Encoder does not implement the following Baseline tools: ASO, FMO, Multiple Reference Frames. All references are limited to the previous Frame. When Frame coding is used, it is accurate and acceptable to describe the associated bitstream as a “Baseline” Profile bitstream. This is because all of the tools used fall completely into the Baseline Profile, and a Baseline-only decoder would be capable of decoding the bitstream.

Field Coding

The only tool outside of the H.264 Baseline Profile that the MG1264 Codec encoder uses is Field (interlace) coding. Field (interlace) coding is typically associated with the Main Profile, although technically it is a part of all Profiles except Baseline.

When Field (interlace) encoding is used, the MG1264 Codec H.264 Encoder produces streams that are fully compliant with the Extended, Main, and High Profiles. Refer to Figure A-1 above. When Frame coding is used, it is accurate and acceptable to describe the associated bitstream as a “Main” Profile bitstream. This is because all of the tools used fall completely into the Main Profile, and a strictly Main Profile compliant decoder would be capable of decoding the bitstream. Although the same is true for the Extended Profile, this Profile is not commonly used and if required would be called out specifically to highlight the unique features (switching Slices and Data Partitioning).

It is exceptionally uncommon to implement only a single tool of a Profile, such as only Field coding in the MG1264 Codec. For this reason, Mobilygen typically does not refer to bitstreams

produced with Field coding as Main Profile bitstreams. This is technically inaccurate, but offers a better description of the actual bitstream.

A.2 MG1264 Codec AAC Encoder Compliance

The MG1264 Codec AAC encoder produces bitstreams that are compliant to AAC-LC.

A.2.1 MG1264 Codec Decoder Compliance

The MG1264 Codec H.264 and AAC decoders are capable of decoding any bitstream that the MG1264 Codec encoders produce. Decoder conformance can only be an issue for bitstreams generated by encoders other than the MG1264 Codec.

Having a decoder be generically compliant is very difficult to prove, and most MPEG decoders do not fully achieve this. It is commonplace for applications to apply limits, or use a subset of the full MPEG spec. DVD and the various MDTV standards (ISDB, DVB-H, DMB) are good examples of applications that bounds the limits of the generic MPEG-2 / H.264 spec. Such decoders are designed to support these bounded limits rather than claim generic MPEG-2 / H.264 compliance.

A.2.2 MG1264 Codec H.264 Decoder Compliance

The MG1264 Codec H.264 Decoder can decode any bitstream that the MG1264 Codec H.264 Encoder produces. As previously noted in “MG1264 Codec H.264 Encoder Compliance” on page 242 this includes both Frame and Field coding.

The MG1264 Codec decoder is best described as a Baseline decoder, although it does not support all the tools of the Baseline Profile. The following Baseline Tools are not supported: Multiple Reference Frames, ASO, and FMO. These Tools are seldom used in the majority of applications. If the MG1264 Codec decoder encounters bitstreams that contain these Tools, visual errors are produced at the Macro block level that will propagate until the next I-slice is encountered. The MG1264 Codec decoder will continue to decode and will not stop or freeze.

The only Tool that the decoder supports outside of the Baseline Profile is Field coding. Technically this means that the MG1264 Codec H.264 Decoder is capable of decoding some level of Main Profile streams – those that only use the Field coding mode of the Main Profile Tool set. Additionally, the MG1264 Codec decoder also has limitations in the size of motion vectors that can be supported that are dependant on the Horizontal picture size, and the type (Field/Frame) of coding used.

Multiple Reference Frames

The MG1264 Codec decoder supports only a single reference frame (the previous frame). If a bitstream contains multiple reference frames, the MG1264 Codec decoder will map all motion vectors to the previous frame, producing a visual error that will propagate until the next I-Slice.

ASO and FMO

The MG1264 Codec decoder does not support ASO or FMO.

Limited Motion Vector support

The MG1264 Codec decoder can support only a limited range of Motion Vectors (MV). The range is dependant on the Horizontal picture resolution, and the type (Field/Frame) of coding used. If the MG1264 Codec decoder encounters a bitstream with MVs outside the supported

range, the MV will be mapped to the maximum limit producing a visual error that will propagate until the next I-slice.

The following two tables summarize what MV ranges the MG1264 Codec decoder can support:

Table A-1 MG1264 Codec Motion Vector Range Support for Frame Based Coding

Horizontal Picture Size	Vertical MV Range	Horizontal MV Range
0 < Hor. Size <= 480	±62	±62
480 < Hor. Size <= 560	±54	±62
560 < Hor. Size <= 656	±46	±62
656 < Hor. Size <= 784	±38	±62
784 < Hor. Size <= 800	±38	±30

Table A-2 MG1264 Codec Motion Vector Range Support for Field Based Coding

Horizontal Picture Size	Vertical MV Range	Horizontal MV Range
0 < Hor. Size <= 464	±60	±62
464 < Hor. Size <= 480	±60	±46
480 < Hor. Size <= 624	±44	±62
624 < Hor. Size <= 640	±44	±46
640 < Hor. Size <= 800	±28	±62

A.3 MG1264 Codec AAC Decoder Compliance

The MG1264 Codec AAC decoder is best described as an AAC-LC Profile decoder. The MG1264 Codec AAC decoder can decode any bitstream that the MG1264 Codec AAC encoder produces. The MG1264 Codec AAC decoder does not support one Tool in the AAC-LC Profile: TNS.

A.3.1 TNS

The MG1264 Codec AAC decoder does not support the TNS Tool in the AAC-LC Profile. The MG1264 Codec AAC decoder firmware performs preemptive bitstream parsing that detects TNS and modifies (removes) the TNS codes before the bitstream reaches the actual decoder block. The result is that TNS is not applied as intended. The audible errors of this parsing work-around are dependant on the content and strength with which TNS was applied by the encoder.

A.3.2 HE-AAC support

HE-AAC support is not listed as a feature of MG1264 Codec. The MG1264 Codec AAC decoder has the ability to render HE-AAC streams by discarding the enhancement (SBR) layer and decoding only the base layer.

Appendix B. Errata to the MG1264 Codec User Manual

This section contains errata regarding the MG1264 Low Power H.264 and AAC Codec for Mobile Devices.

B.1 Phase Lock Loop Restrictions

The maximum frequency for the MG1264 Codec Core Clock is 110 MHz. at worse case conditions. The Core Clock frequency (core_clk) is generated using an internal Phase Lock Loop (PLL) from the clock input on the XIN pin. The Core Clock frequency is calculated using the following equation:

$$\text{core\_clk} = XIN \times \frac{M}{X}$$

where M is set using the PLLFeedBackDivider field and X is set using the PLLOutputDivider field of the PLLDivider register (see page 75).

However, the MG1264 Codec has a restriction on the relationship between the clock input on the VID_CLK pin (video Input Clock) and the Core Clock. The relationship can best be described as follows: The maximum Core Clock frequency of the MG1264 Codec is one PLL resolution below four times the clock on the VID_CLK pin. (See “Phase Lock Loop Restrictions” on page 245.)

For instance, if VID_CLK = 27 MHz, the Core Clock must be less than 4 x 27 MHz (108 MHz.), and 104.625 MHz. is the highest Core Clock frequency below the 4 x 27 MHz (108 MHz.) limit. The equation for generating a 104.625 MHz Core Clock is:

$$104.625MHz = 27MHz \times \frac{31}{8}$$

Where the M/X ratio of 31/8 meets the requirement of being one PLL resolution below four times the clock on the VID_CLK pin.

B.2 Minimum Picture Size

The minimum picture size that can be encoded is 96 x 96. The resolution can be obtained by either setting the capture rectangle to that resolution, or by scaling a larger capture rectangle to that resolution. See the crop and scaling commands for more information.

However, note that you must use one slice per macroblock row for any horizontal resolution below 128, meaning that pictures that are 112 or 96 pixels wide must use one slice per row.

Revision History

The Revision History table shows recent changes to the document. Please note that the page number refers to the page where the section heading occurs, and that the actual change or changes may be on one or more of the following pages.

Revision	Description of Change	Pages Affected
0.95	Pin Change: Pin A16 was changed from GND to PFILTER. This was done to allow filtering on the PLL power supply to minimize jitter.	27, 36, 37
	The Core Clock frequency was increased from 104 MHz to 104.625 MHz.	37
	The Operating Temperature Range (Case) was changed to –20 to +125 °C, and the $T_{Ambient}$ temperature range was changed to –20 to +85 °C.	44
1.00	Removed all Change Bars (no change to content). This document will no longer use change bars. Use the Revision History to track specific changes.	Entire Document
	First Paragraph - removed two references to VGA in the description.	15
	Added Note: The minimum resolution of encoding using the internal scaler is 144x96 pixels.	21, 181
	Added the OmniVision OV7220 sensor to the Compatible CMOS Sensors list.	92
	Added description on processing AAC Audio QBox bitstream format.	138
	Add clarification regarding sending encoded bitstreams to the decoder.	159
	Add numerous clarifications regarding sending encoded bitstreams to the decoder.	159
	Added new section “Overview of the Video Capture Process”	184
1.1	Rewrote the section “Input Video Scaling” to indicate the new minimum picture size of 96 x 96 pixels.	21
	Added new section “MG1264 Codec SDRAM Requirements by Function”	21
	Chapter 2, throughout. Added information on the new 169-pin Thin & Fine-Pitch Ball Grid Array package (TFBGA). This included a new package pinout drawing (“Pinout Diagram for the MG1264 Codec in the 169-pin TFBGA Package” on page 26), additional columns in the Pin List tables starting on 30, and a new package physical drawing (“169-pin TFBGA Package Mechanical Dimensions” on page 38).	25 - 41
	Table 2-1, “MG1264 CODEC Host Interface Pins” Added cross references in the H_DMARQ pin description to the ports used in DMA operations.	30
	Added a note to the TMS, TDI, and TRST IU pins: “This pin has an internal 20 kOhm - 150 kOhm (50 kOhm nominal) pull-up resistor.”	34

Revision	Description of Change	Pages Affected
1.1	Moved the description of the test pins from Chapter 8 (now removed) to sections Section 2.2.1 through Section 2.2.3.	34
	Updated “XIN Core Clock Considerations” and “VID_CLK Video Clock Considerations” on page 37 to reflect the errata described in “Phase Lock Loop Restrictions” on page 245.	37
	Added new section “Ordering Information”	39
	“DC Characteristics” on page 45: Added new parameter I_{PU} describing the internal Pullup resistor current draw.	45
	Added new section “Standby Power” describing the standby power requirements.	46
	Figure 3-3, “MG1264 Codec Host Interface AC Timing Waveform”: Inverted the H_DMARQ signal.	49
	Figure 3-3, “MG1264 Codec H_DMARQ Timing”: Inverted the $\overline{\text{H\_DMARQ}}$ signal.	50
	Table 3-5, “Host Interface Timing”: Made the following changes: • Added 110 MHZ Maximum Core Clock frequency and a footnote referencing the Phase Lock Loop Restrictions errata. • Changed the value for parameter t_{WAS} from 37 ns. to 20 ns. • Added the following footnote to the t_{RAS} parameter: H_ADDR[6:1] must be stable before H_RD is asserted. Make sure that delays caused by the printed circuit board layout are taken into account when programming the bus timings. • Changed the value for parameter t_{WDC} from 37 ns. to 20 ns.	52
	Table 3-6, “Video Interface AC Timing Values”: Added a minimum value of 25 ns. to the t_{VC} parameter	53
	Broke the MG1264 Codec Register and External Memory Device Register Map table into four separate tables to improve clarity.	68
	Updated the Phase Lock Loop Divider register description to reflect the errata described in “Phase Lock Loop Restrictions” on page 245.	75
	Section “Interlaced ITU-R BT.656 Video Interfaces” Added descriptions regarding adjustable timing in non-standard video modes and minimum values for the Horizontal and Vertical Blanking intervals.	88
	Removed the section: “Progressive Video Interface for D1 Resolution and Below” and replaced it with a new section: “Progressive Video Interface in Free-run Mode”	90
	Added a note to the “Working With CMOS Sensors” section: “Because there is a great deal of variance between different sensors with respect to video clock gating, compliance, etc., we strongly recommend that you contact Mobilygen Technical Support before starting a design that includes a CMOS sensor.”	92
	Added new Section 5.5, “Video Pre-Processing Filters” describing the four video pre-processing filters that can be used to improve the encoded picture quality of source video.	93
	Added information regarding the audio clock, AUD_LRCK, and AUD_BCK signals during master and slave operation.	100

Revision	Description of Change	Pages Affected
1.1	Removed existing Chapter 8 “Miscellaneous Signals” and moved the relevant information into Chapter 2. This caused the chapter numbers on all of the subsequent chapters to decrease by 1.	
	Added a definition for the `sample_flags` parameter.	110
	Added information to the “AV Encoder Features” section regarding the minimum picture size problem discussed in the errata on page 246.	181
	Added Appendix B “Errata to the MG1264 Codec User Manual”	245

Mobilygen Corporation
2900 Lakeside Drive #100
Santa Clara, CA 95054
Tel: (408) 869-4000
Fax: (408) 980-8044
email: info@mobilygen.com