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

Low Power H.264 and AAC codec

Document Version: 0.8

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:

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 VGA codec IC that enables mobile products to capture, play and share video with three times the processing power of competing devices, while using substantially less power. The MG1264 is a complete A/V codec solution including both a H.264 VGA 30 frames-per-second video codec, and a high fidelity two-channel AAC audio codec. Power consumption while encoding is 185mW 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 still cameras, video cameras, cell phones with integral cameras, personal media players, 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

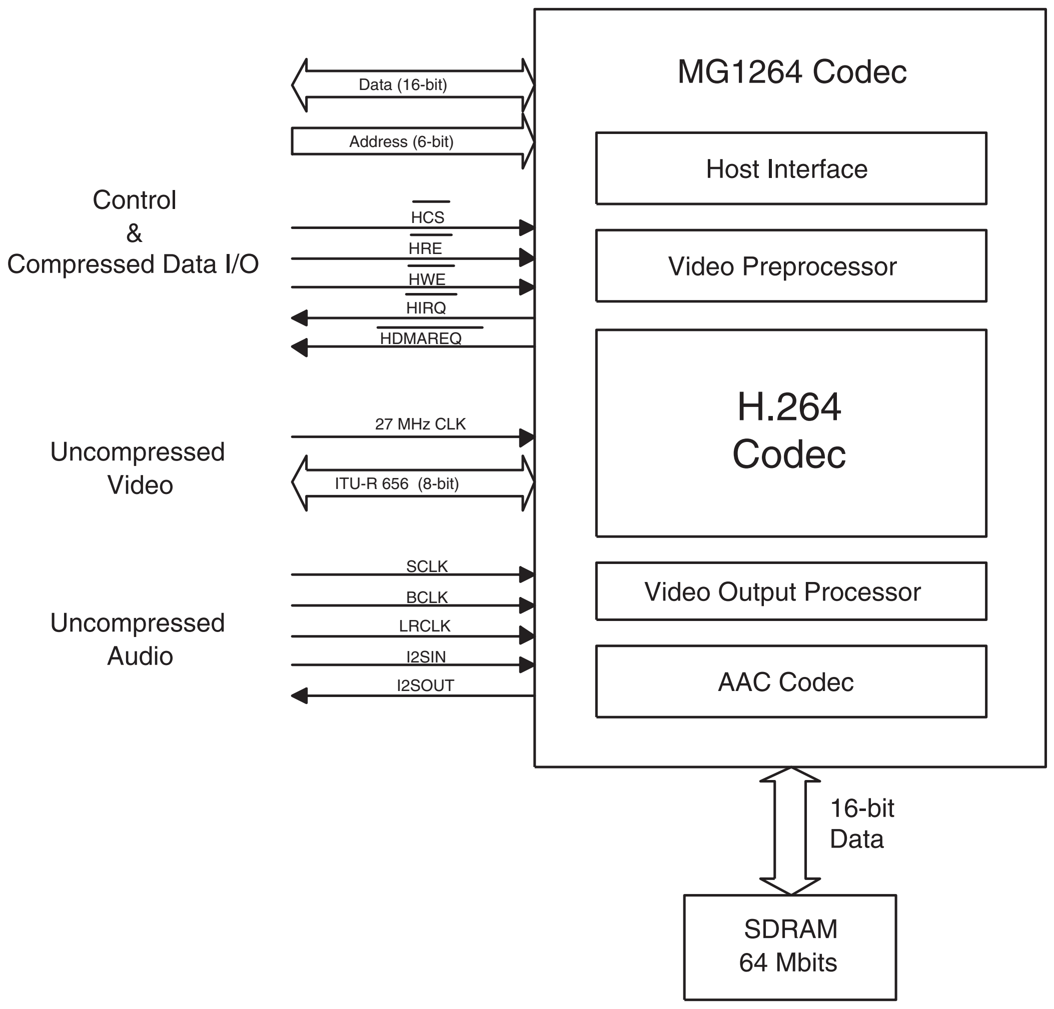


Figure 1-1 MG1264 Codec Block Diagram

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:

- Digital Still Cameras
- Solid-State Camcorders
- Portable Media Players
- Camera-enabled Cellular Phones

The MG1264 Codec produces syntactically correct A/V bitstreams that can be decoded by any standard-compliant decoder that incorporates support for H.264 and AAC playback 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. It can also play back any H.264 encoded stream using the tools we support. Figure 1-2 shows the MG1264 Codec's capabilities.

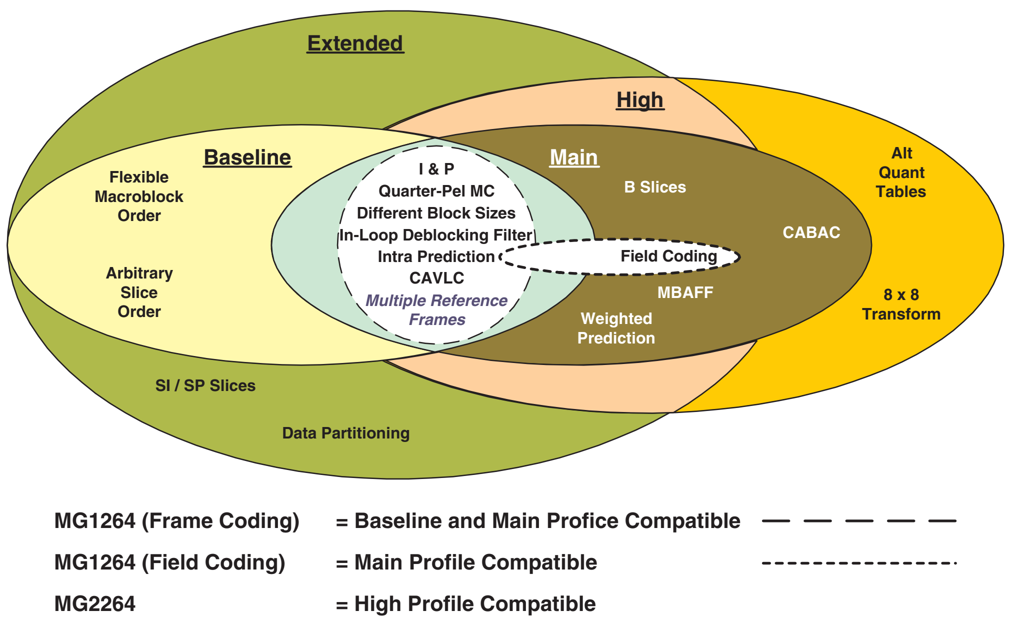


Figure 1-2 H.264/AVC Tools/Profiles

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.

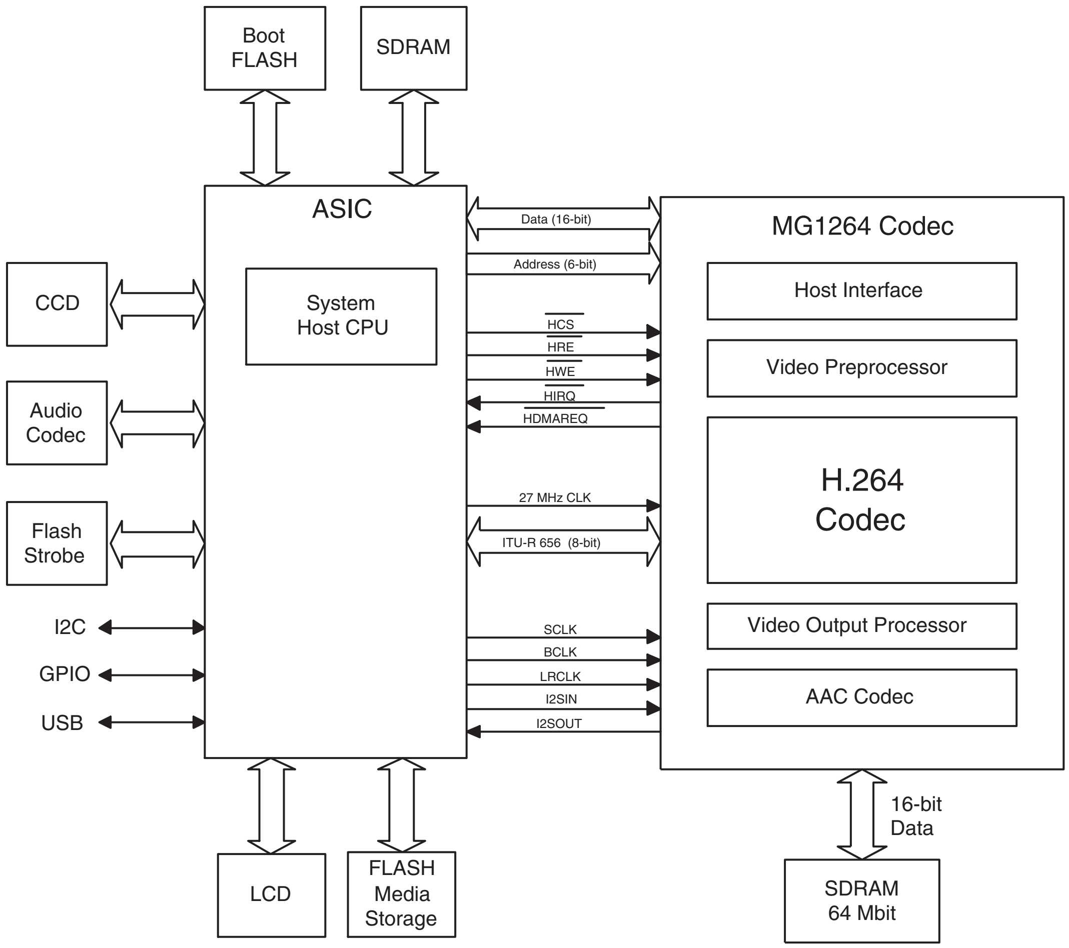


Figure 1-3 Camera System-Level Block Diagram

1.3 Features

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

1.3.1 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, 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 15). Baseline is the primary encoding mode for the MG1264 Codec because today all DSC sensors use progressive scan CCDs, however the MG1264 Codec also supports Field mode coding. Streams coded as Field mode fall within the Main Profile.

The decoder in the MG1264 Codec decodes only streams created with the same subset of tools as listed above. For Baseline Profile, ASO and FMO are not widely used, so the primary limitation is in picture resolution (maximum 800x600x25 fps) and bitrate (10 Mbps).

1.3.2 H.264 Encoder Target Performance

The MG1264 Codec is capable of standard definition full resolution (720x480 or 640x480) video encoding. While resolution down sampling can provide excellent visual results for video encoding at lower bitrates, the product emphasis is on full resolution.

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

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

Video Bitrate (kbps)	Horizontal Resolution (Pixels)	Vertical Resolution (Pixels)	fps[a]	Notes Regarding The Source Video
300	320	240	30	QVGA, progressive, square pixel sensor
500	640	240	30	Half-vertical, progressive, square pixel sensor
1000	640	480	30	VGA, progressive, square pixel sensor
1500	640	480	30	VGA, progressive, square pixel sensor
2000	640	480	30	VGA, progressive, square pixel sensor
3000	640	480	30	VGA, progressive, square pixel sensor
3000	640	720	30	HD-M
3000	800	600	25	SVGA, progressive, square pixel sensor
300	352	240	30	SIF, progressive, rectangular pixel sensor
500	720	240	30	Half-vertical, interlace, rectangular pixel sensor
1000	720	480	30	D1, interlace, rectangular pixel sensor
1500	720	480	30	D1, interlace, rectangular pixel sensor
2000	720	480	30	D1, interlace, rectangular pixel sensor
3000	720	480	30	D1, interlace, rectangular pixel sensor

a. 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.

1.3.3 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	352	288	25	QSIF, progressive, rectangular pixel sensor
500	720	288	25	Half-vertical, progressive, rectangular pixel sensor
1000	720	576	25	D1, interlace, rectangular pixel sensor
1500	720	576	25	D1, interlace, rectangular pixel sensor
2000	720	576	25	D1, interlace, rectangular pixel sensor
3000	720	576	25	D1, interlace, rectangular pixel sensor

1.3.4 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 the standard 27 MHz video clock. The maximum frame rate is 25 fps.

1.3.5 Video Input and Output Scaling

The MG1264 Codec is capable of performing video scaling both on the input during pre-encoding and on the output during post-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 VGA/D1 resolution video input and generate the target encoding resolutions listed in Table 1-1. The MG1264 Codec supports a maximum horizontal resolution of 800pixels.

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.

1.3.6 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

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 300 kbps. The bitrate can be specified in 100 kbps increments from 300 kbps to 10 Mbps.

Picture Type

The Picture Type refers to as Frame or Field coding. Frame mode is the most common mode used in Digital Still Cameras because they have progressive sensors at 30 fps. Field mode is used in most other mobile devices. 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. The default GOP length is 15.

1.3.7 The AAC Audio CODEC

The MG1264 Codec can encode two-channel AAC audio encoding with 16-bit samples at sample rates of 22.05 kHz, 24 kHz, 32 kHz, 44.1 kHz, and 48 kHz. 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 1-3 shows the key features and their associated target settings.

Table 1-3 AAC Encoder Features

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

1.3.8 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 camera 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, audio codec, etc.

Chapter 2. MG1264 Codec Host Interface

The System Host CPU controls the MG1264 Low Power H.264 and AAC Codec for Mobile Devices through the MG1264 Codec 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.

2.1 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 (HDATA[15:0], six bits of address (HADDR[6:1]), and control signals.

2.1.1 Connection Diagram

The MG1264 Codec Host Interface connection diagram is shown in Figure 2-1.

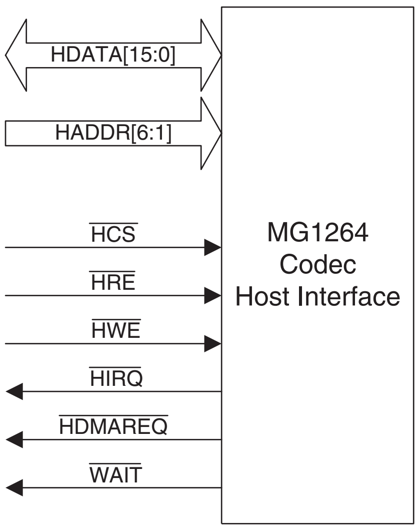


Figure 2-1 MG1264 Codec Host Interface Connection Diagrams

The MG1264 Codec Host Interface has a single Host 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 2-2.

2.1.2 MG1264 Codec Host Interface Signals

The signals that comprise the MG1264 Codec Host Interface are shown in Table 2-1.

Table 2-1 MG1264 Codec Host Interface Pin Descriptions

Pin Name	Signal Name	Direction	Description
HDATA[15:0]	Data [15:0]	Bidirectional	16-bit Host Data Bus
HADDR[6:1]	Address [6:1]	Inputs	Six bits of Host Address
$\overline{\text{HCS}}$	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.
$\overline{\text{HRE}}$	$\overline{\text{RE}}$	Input	Active Low Read Enable
$\overline{\text{HWE}}$	$\overline{\text{WE}}$	Input	Active Low Write Enable
$\overline{\text{HIRQ}}$	$\overline{\text{INT}}$	Output (Open Collector)	Active Low, Open Collector Host Interrupt Request
$\overline{\text{HDMAREQ}}$	$\overline{\text{Host DMA Request}}$	Output	Bitstream DMA Request associated with the Bitstream port
$\overline{\text{HWAIT}}$	$\overline{\text{WAIT}}$	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).

2.2 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.

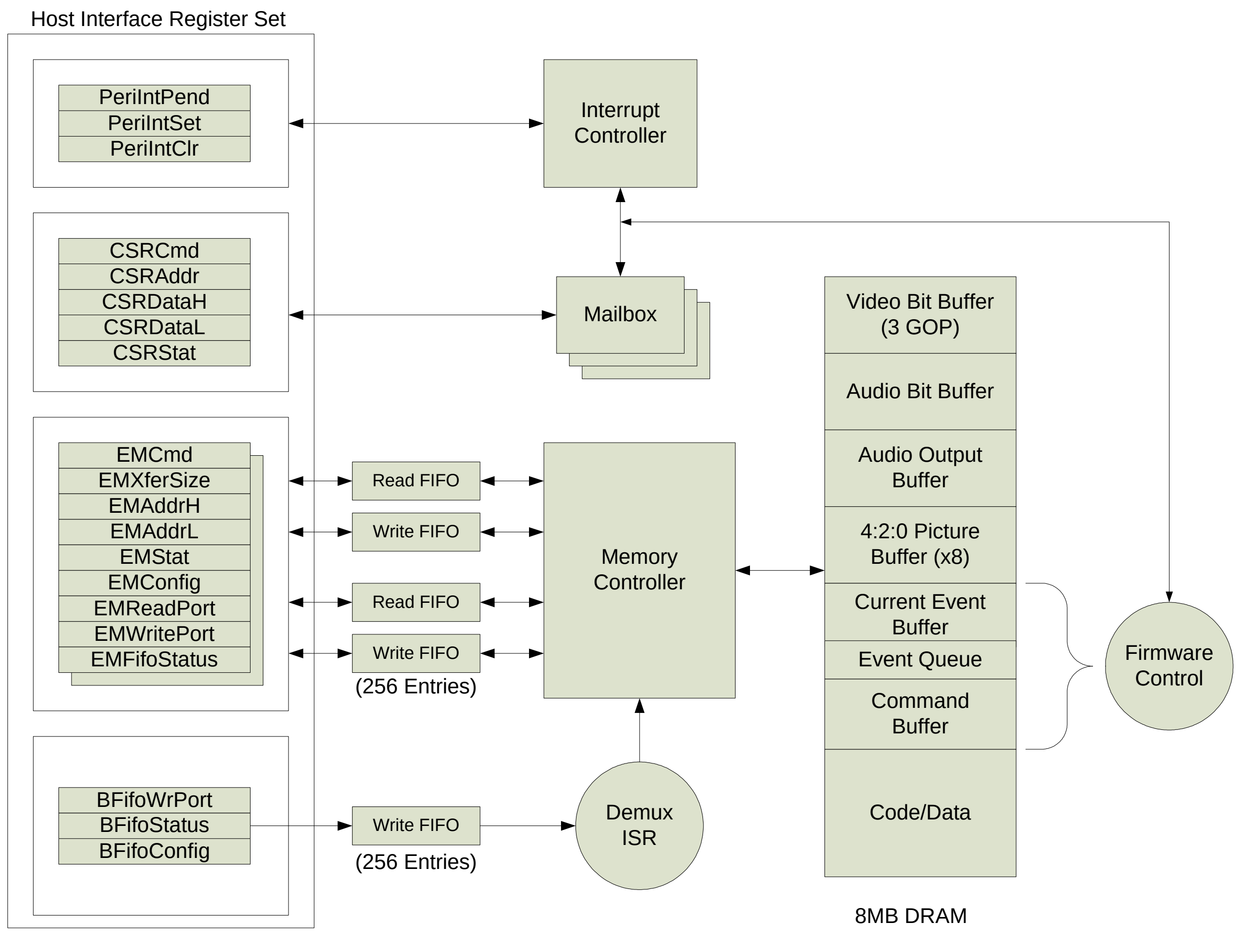


Figure 2-2 Register Logical View

2.2.1 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.

2.2.2 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.

2.2.3 Interrupts

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

- 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 36.

2.2.4 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 41.

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 47.

2.2.5 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.

2.3 Read/Write Timing

This section provides generic timing information for the MG1264 Codec Host Interface. For specific timing information, refer to “Specifications” on page 153. For information on the programming sequence needed to read or write a register, refer to “Register Definitions” on page 34.

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 and Enable mode. Note that these options are available in both single Host Chip Select mode and two Host Chip Select mode. Each of these modes is discussed in the sections that follow.

The MG1264 Codec defaults to the separate Read Enable and Write Enable signalling as shown in Figure 2-3 and Figure 2-4. To put the host interface into Read/Write and Enable mode (Figure 2-5 and Figure 2-6), the very first transaction on the read bus must be a Write transaction using the separate Enable and RD/$\overline{\text{WR}}$ 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 Enable and RD/$\overline{\text{WR}}$ mode. A data value of 0x0000 should be used.

2.3.1 Read Timing Sequence in Read Enable Mode

Figure 2-3 shows the timing for a System Host CPU read from the MG1264 Codec in Read Enable mode.

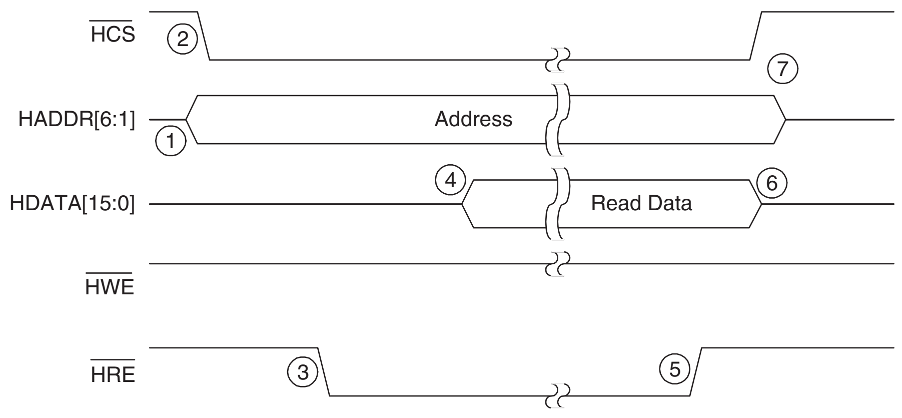


Figure 2-3 Read Access Timing in Read Enable Mode

1. The System Host CPU must assure that the address bus (HADDR[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 (HRE) signal to inform the MG1264 Codec that the operation will be a read.
4: The data becomes available on HDATA[15:0].
5: Once the data has been taken, the System Host CPU de-asserts the Host Read Enable (HRE) signal to indicate to the MG1264 Codec that the transaction is complete.
6: The MG1264 Codec removes the output data from the data bus (HDATA[15:0]).
7: The System Host CPU then de-asserts the address bus (HADDR[6:1]) and the Host Chip Select to complete the transaction.

2.3.2 Write Data Timing in Write Enable Mode

Figure 2-4 shows the timing for a System Host CPU write to the MG1264 Codec in Write Enable mode.

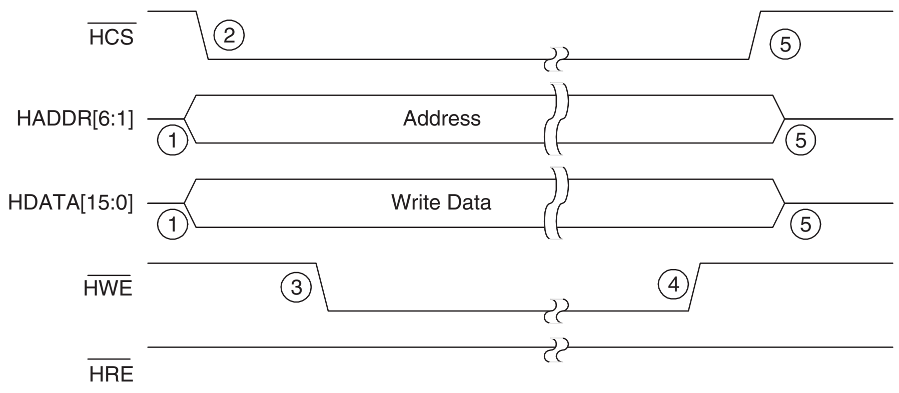


Figure 2-4 Write Access Timing in Write Enable Mode

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

2.3.3 Read Timing Sequence in Read/Write and Enable Mode

Figure 2-3 shows the timing for a System Host CPU read from the MG1264 Codec in Read/ Write mode.

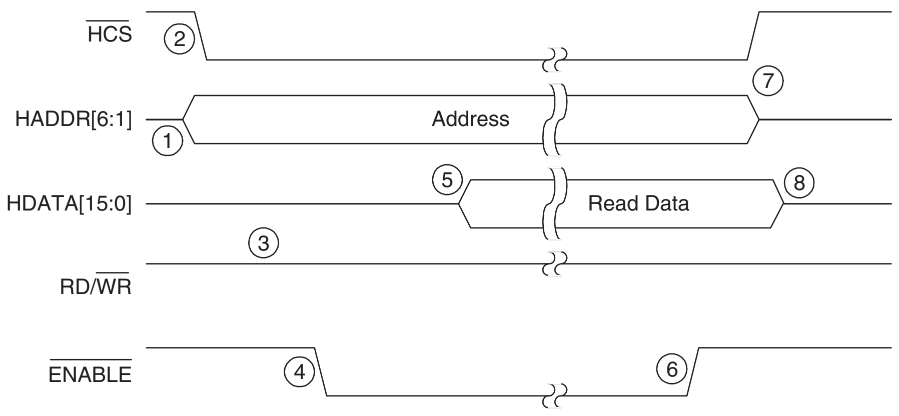


Figure 2-5 Read Access Timing in Read/Write and Enable Mode

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

2.3.4 Write Data Timing in Read/Write and Enable Mode

Figure 2-4 shows the timing for a System Host CPU write to the MG1264 Codec in Read/Write and Enable mode.

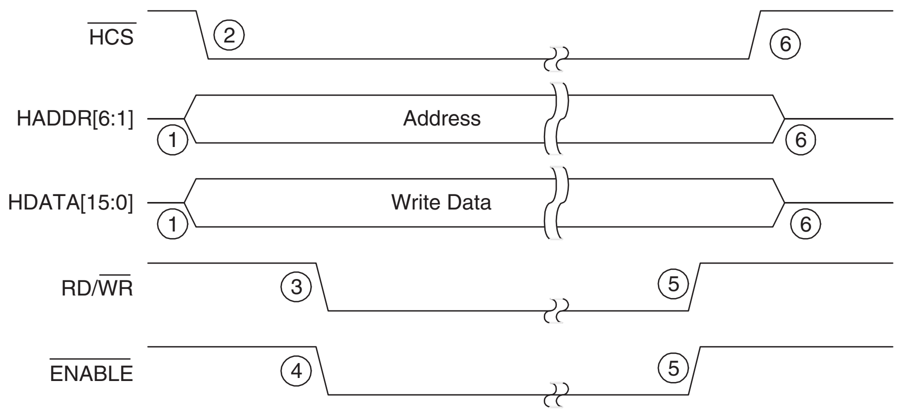


Figure 2-6 Write Access Timing in Read/Write and Enable Mode

1. The System Host CPU must assure that the address bus (HADDR[6:1]) and data to be written (on HDATA[15:0]) is stable before asserting the $\overline{\text{Host Chip Select}}$ ($\overline{\text{HCS}}$).
2: The System Host CPU asserts the $\overline{\text{Host Chip Select}}$ 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/$\overline{\text{Write}}$ 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 Read/$\overline{\text{Write}}$ signal and $\overline{\text{Enable}}$ signal to indicate to the MG1264 Codec that the write is complete.
6: The System Host CPU de-asserts the Address bus (HADDR[6:1]), Write Data bus (HDATA[15:0]), and the $\overline{\text{Host Chip Select}}$ to indicate to the MG1264 Codec that the transaction is complete.

2.4 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 39 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 $\overline{\text{HDMAREQ}}$ pin
- A register bit (EMFifoRdReq/ EMFifoWrReq)
- The external $\overline{\text{HWAIT}}$ pin

2.4.1 Pacing using the $\overline{\text{HDMAREQ}}$ Pin

The MG1264 Codec asserts the $\overline{\text{HDMAREQ}}$ pin when a programmable threshold (EMDThresh, see page 45) is reached in the DMA transfer FIFO. For a read DMA, the MG1264 Codec asserts the $\overline{\text{HDMAREQ}}$ pin when EMDThresh number of words is available to be transferred to the System Host CPU. The MG1264 Codec deasserts the $\overline{\text{HDMAREQ}}$ pin once the number of words available to be read falls below EMDThresh.

For a write DMA, the $\overline{\text{HDMAREQ}}$ pin is asserted when the MG1264 Codec is able to accept EMDThresh number of 1b-bit words to be written. The HDMAREQ pin is de-asserted once the number of words available to be written falls below EMDThresh.

2.4.2 Pacing using the EMFifoRdReq/EMFifoWrReq Bits

The EMFifoRdReq or EMFifoWrReq Bits in the EMFifoStatus Register (see page 46) are reflections of the $\overline{\text{HDMARQ}}$ pin and are set accordingly if in read or write DMA mode.

2.4.3 Pacing using the HWAIT Pin

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

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

2.5 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 2-2). 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.

2.5.1 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.

2.5.2 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.

2.6 Programming the MG1264 Codec Host Interface

2.6.1 Register Maps

Table 2-2 shows the MG1264 Codec Registers, External Memory Device Register Map, Bitstream Read Memory and Bitstream Write FIFO Device Register Map. These registers are addressed when the Host Chip Select ($\overline{HCS}$) signal is asserted.

Table 2-2 MG1264 Codec Register and External Memory Device Register Map

Register	Offset	Access	Description	Page
CSRCmd	0x0020	R/W	Configuration/Status Register Command	34
CSRAddr	0x0022	R/W	Configuration/Status Register Address	34
CSRWrDataH	0x0024	R/W	Configuration/Status Register Write Data High	34
CSRWrDataL	0x0026	R/W	Configuration/Status Register Write Data Low	34
CSRRdDataH	0x0028	Read	Configuration/Status Register Read Data High	35
CSRRdDataL	0x002A	Read	Configuration/Status Register Read Data Low	35
CSRStat	0x002C	R/W	Configuration/Status Register Status	35
PeriIntPend	0x002E	R/W	Peripherals Interrupt Pending	36
PeriIntEnSet	0x0030	R/W	Peripherals Interrupt Enable - Set	36
PeriIntEnClr	0x0032	R/W	Peripherals Interrupt Enable - Clear	36
ClkConfig	0x0034	R/W	Clock Configuration Register	37
PLL Dividers	0x0036	R/W	PLL Dividers Register	37
ChipID	0x0038	R	Chip ID Register	38
EM1Cmd	0x0000	R/W	External Memory DMA Command	41
EM1XferSize	0x0002	R/W	External Memory DMA Transfer Size	41
EM1SrcAddrH	0x0004	R/W	External Memory DMA Source Address High or Starting Vertical/ Y Source Address	42
EM1SrcAddrL	0x0006	R/W	External Memory DMA Source Address Low or Starting Horizon-tal/X Source Address	42
EM1DestAddrH	0x0008	R/W	External Memory DMA Destination Address High or Starting Ver-tical/Y Destination Address	42
EM1DestAddrL	0x000A	R/W	External Memory DMA Destination Address Low or Starting Hor-izontal/X Destination Address	42
EM1Status	0x000C	Read	External Memory DMA Status	44
EM1RemCount	0x000E	Read	External Memory DMA Transfer Remainder Count	44
EM1Config	0x0010	R/W	External Memory DMA Configuration	45
EM1FifoRdPort	0x0012	Read	External Memory DMA FIFO Read Port (from memory)	46

Table 2-2 MG1264 Codec Register and External Memory Device Register Map

Register	Offset	Access	Description	Page
EM1FifoWrPort	0x0014	R/W	External Memory DMA FIFO Write Port (to memory)	46
EM1FifoStatus	0x0016	Read	Bitstream Memory DMA Status	46
BFifoWrPort	0x0060	R/W	Bitstream FIFO Write Port (to Media Engine)	47
BFifoStatus	0x0062	Read	Bitstream FIFO Status Register	47
BFifoConfig	0x0064	R/W	Bitstream FIFO Command Register	47
EM2Cmd	0x0040	R/W	Bitstream Memory DMA Command	41
EM2XferSize	0x0042	R/W	Bitstream Memory DMA Transfer Size	41
EM2SrcAddrH	0x0044	R/W	Bitstream Memory DMA Source Address High or Starting Vertical/Y Source Address	42
EM2SrcAddrL	0x0046	R/W	Bitstream Memory DMA Source Address Low or Starting Horizontal/X Source Address	42
EM2DestAddrH	0x0048	R/W	Bitstream Memory DMA Destination Address High or Starting Vertical/Y Destination Address	42
EM2DestAddrL	0x004A	R/W	Bitstream Memory DMA Destination Address Low or Starting Vertical/Y Source Address	42
EM2Status	0x004C	Read	Bitstream Memory DMA Status	44
EM2RemCount	0x004E	Read	Bitstream Memory DMA Transfer Remainder Count	44
EM2Config	0x0050	R/W	Bitstream Memory DMA Configuration	45
EM2FifoRdPort	0x0052	Read	Bitstream Memory DMA FIFO Read Port (from memory)	46
EM2FifoWrPort	0x0054	R/W	Bitstream Memory DMA FIFO Write Port (to memory)	46
EM2FifoStatus	0x0056	Read	Bitstream Memory DMA FIFO Status	46

2.7 Register Definitions

2.7.1 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 I_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.

2.7.2 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.

2.7.3 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_clk is generated from XIN using the following equation:

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

The default, assuming that XIN = 27 MHz is core_clk = 81 MHz. When programming these bits, ClkEn must be set to 0 first 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=3 Restriction: $2<=M<=37$ for 27 MHz input clock.
PLLOutput Divider	00: The PLL output divider X is set to 8. 01: The PLL output divider X is set to 1 (Default). 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

2.7.4 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 2-2. 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 HADDR[6] is high (Bitstream write) out to a pin. $\overline{\text{HDMAREQ}}$ 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 $\overline{\text{HDMAREQ}}$ 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 HADDR = 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 HADDR[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.

2.7.5 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 40 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 40 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).

2.7.6 Checking the FIFO Status

The interface logic asserts a DMA request to the System Host CPU (by asserting $\overline{\text{HDMAREQ}}$) 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.

2.7.7 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 16-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 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 HWAIT signal generated (default) 1 = HWAIT signal is generated. EMDThresh should be set to 1 when EMWait is set to 1
EMDThresh	When this number of bytes are left in the fifo, the DMAREQ signal or the EMFIFOStatus bits are deasserted.
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.

2.7.8 Bitstream Write FIFO Access Registers

The System Host CPU sends a bitstream (e.g., MPEG transport or program stream) 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 BFi-foWrReq bit in the BFIFOStatus register is deasserted.

The interface logic asserts the DMA request to the System Host CPU by driving $\overline{\text{HDMAREQ}}$ 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 3. 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 3-1.

Table 3-1 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

3.1 Video Interface Usage

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

3.1.1 Interlaced ITU-RBT.656 Video Interfaces

The MG1264 Codec has video input and output interfaces for interlaced video that are 656-compliant. Figure 3-1 shows the diagram for NTSC 656 Interlaced Video, and Figure 3-2 shows the diagram for PAL 656 Interlaced Video.

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 3-1 and Figure 3-2.

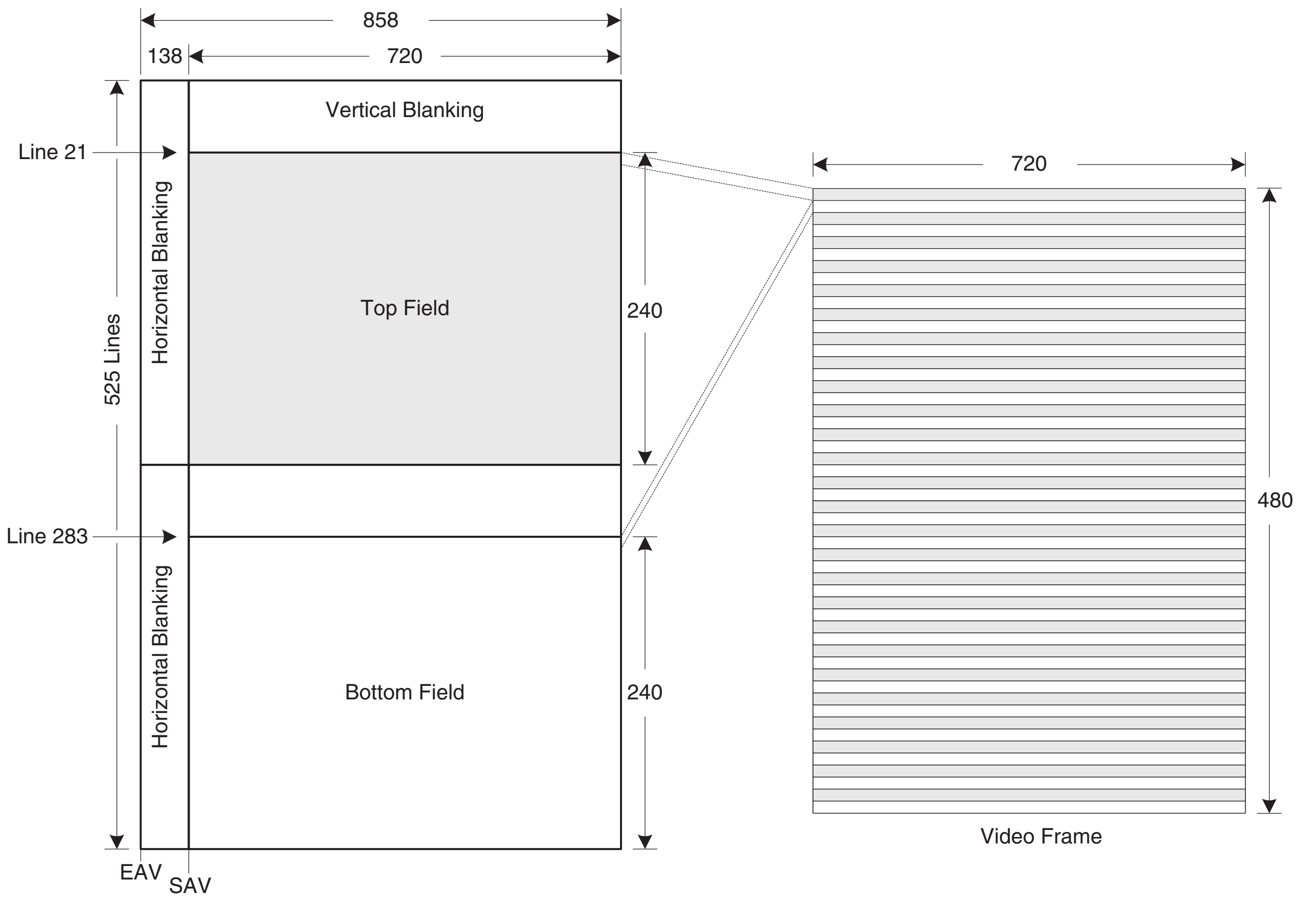


Figure 3-1 ITU-R BT.656 NTSC Interlaced Video Standard

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.

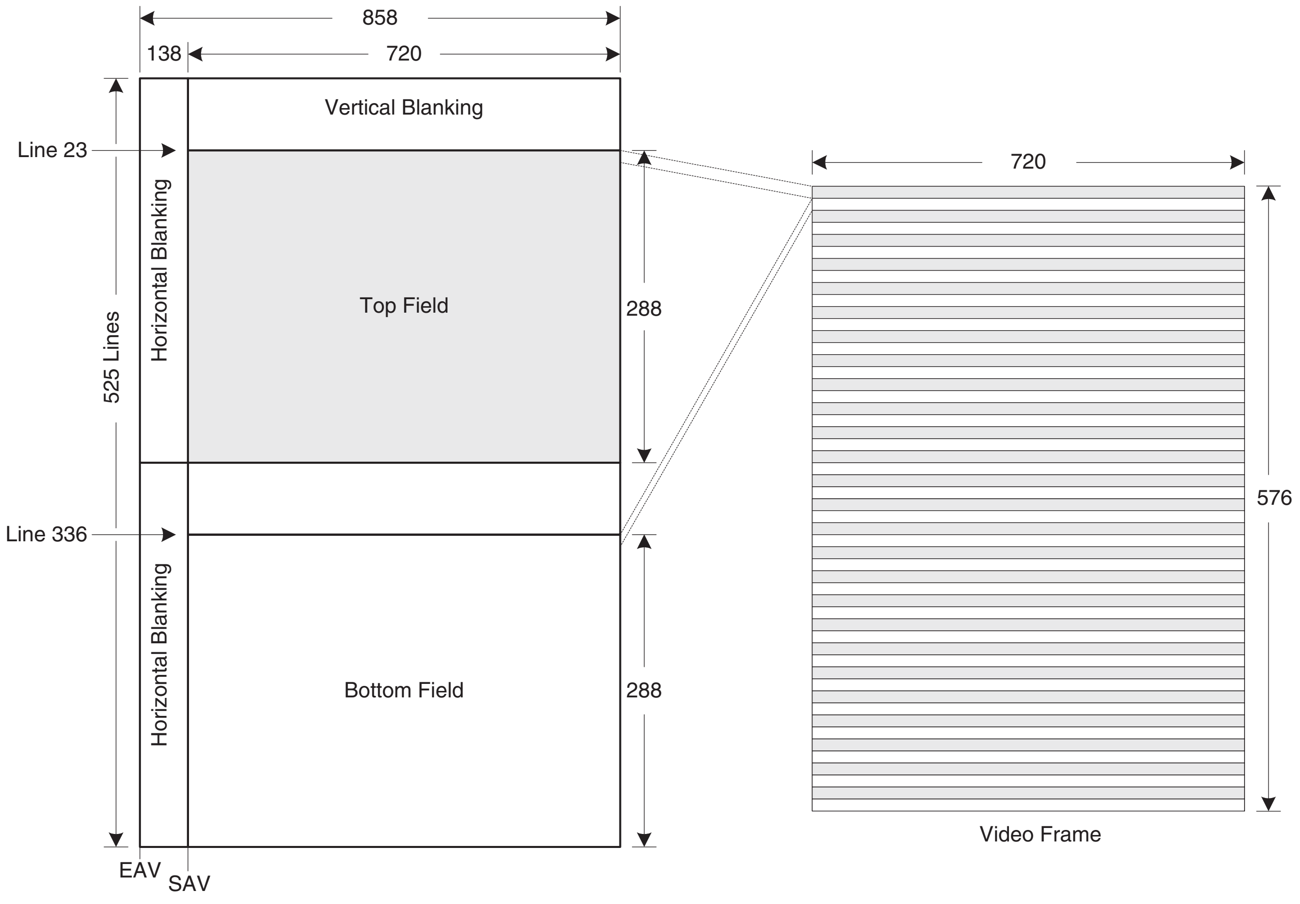


Figure 3-2 ITU-R BT.656 PAL Interlaced Video Standard

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

3.1.2 Progressive Video Interfaces for D1 Resolution and Below

There is no digital transmission standard for progressive video. The video interfaces (input and output) of the MG1264 Codec format progressive frames as shown in Figure 3-3 for NTSC video and Figure 3-4 for PAL video. The interfaces uses normal 656 timing. Instead of alternating interlaced fields, the top half of the frames are alternated with the bottom half of the frames.

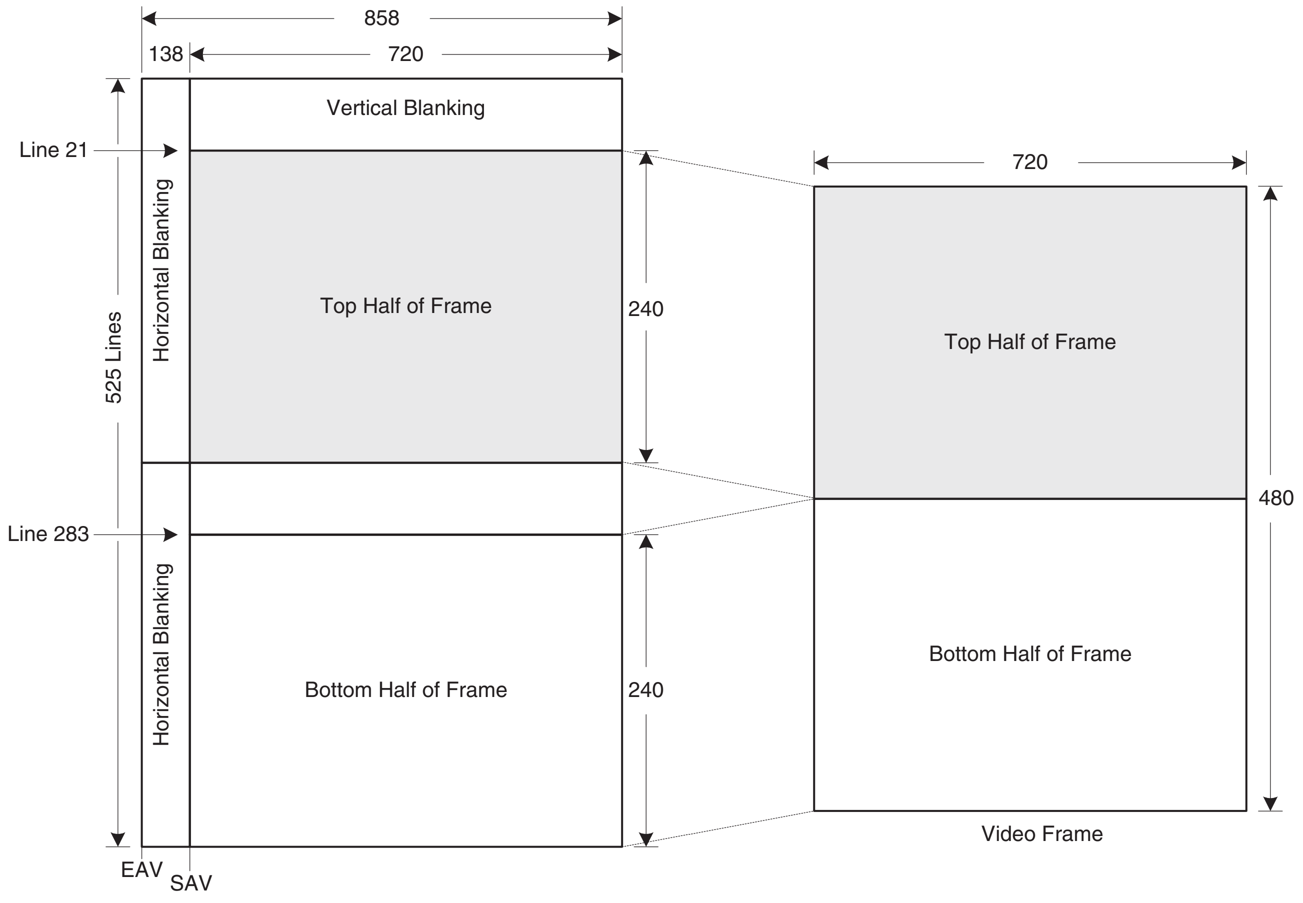


Figure 3-3 NTSC Progressive Video

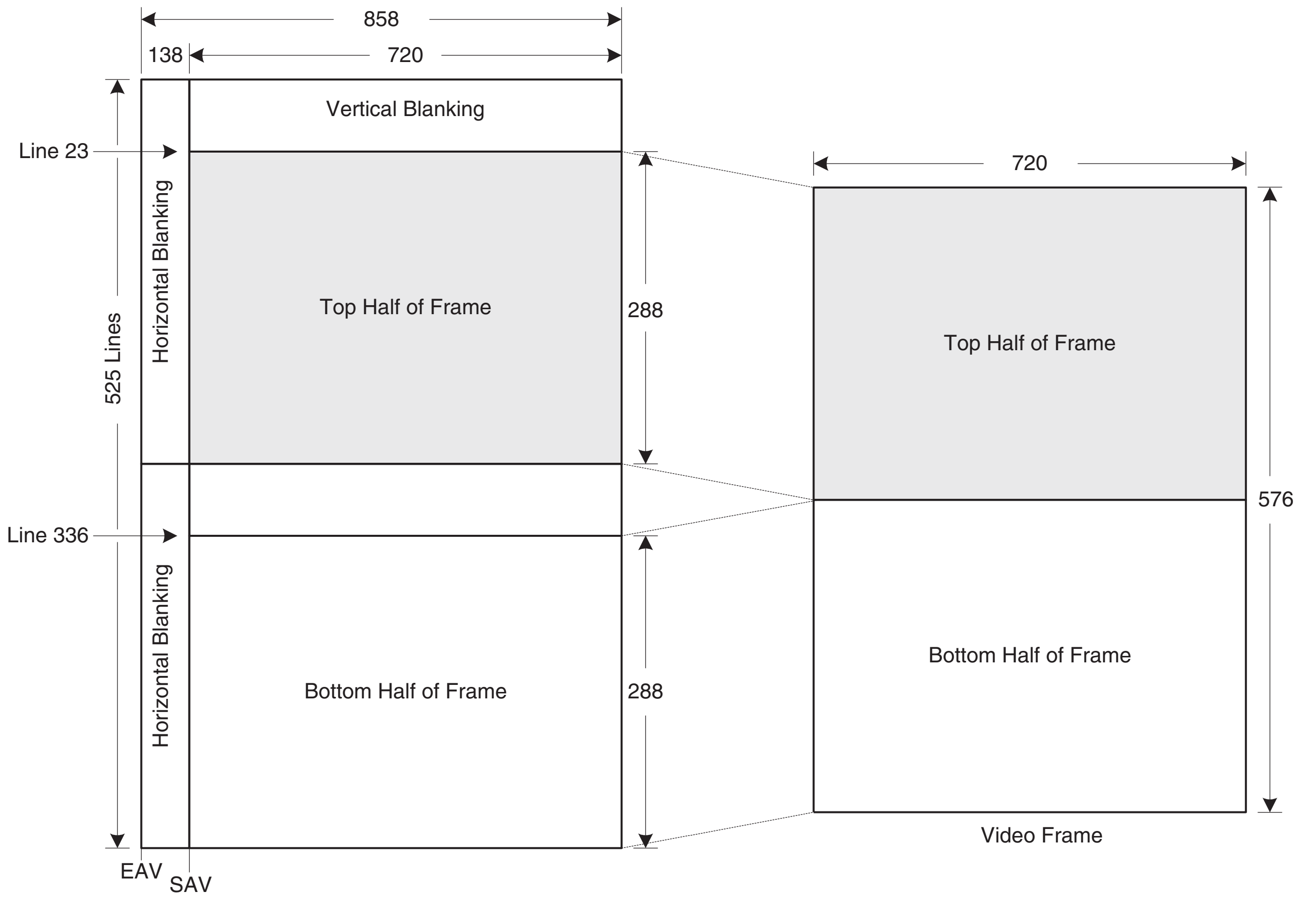


Figure 3-4 PAL Progressive Video

When the MG1264 Codec needs to reduce the number of horizontal pixels. For example, to support VGA input to 640 horizontal pixels (as shown in Figure 3-5), it discards pixels on the right-hand side of the frame. In the example shown in Figure 3-5, 80 pixels are discarded.

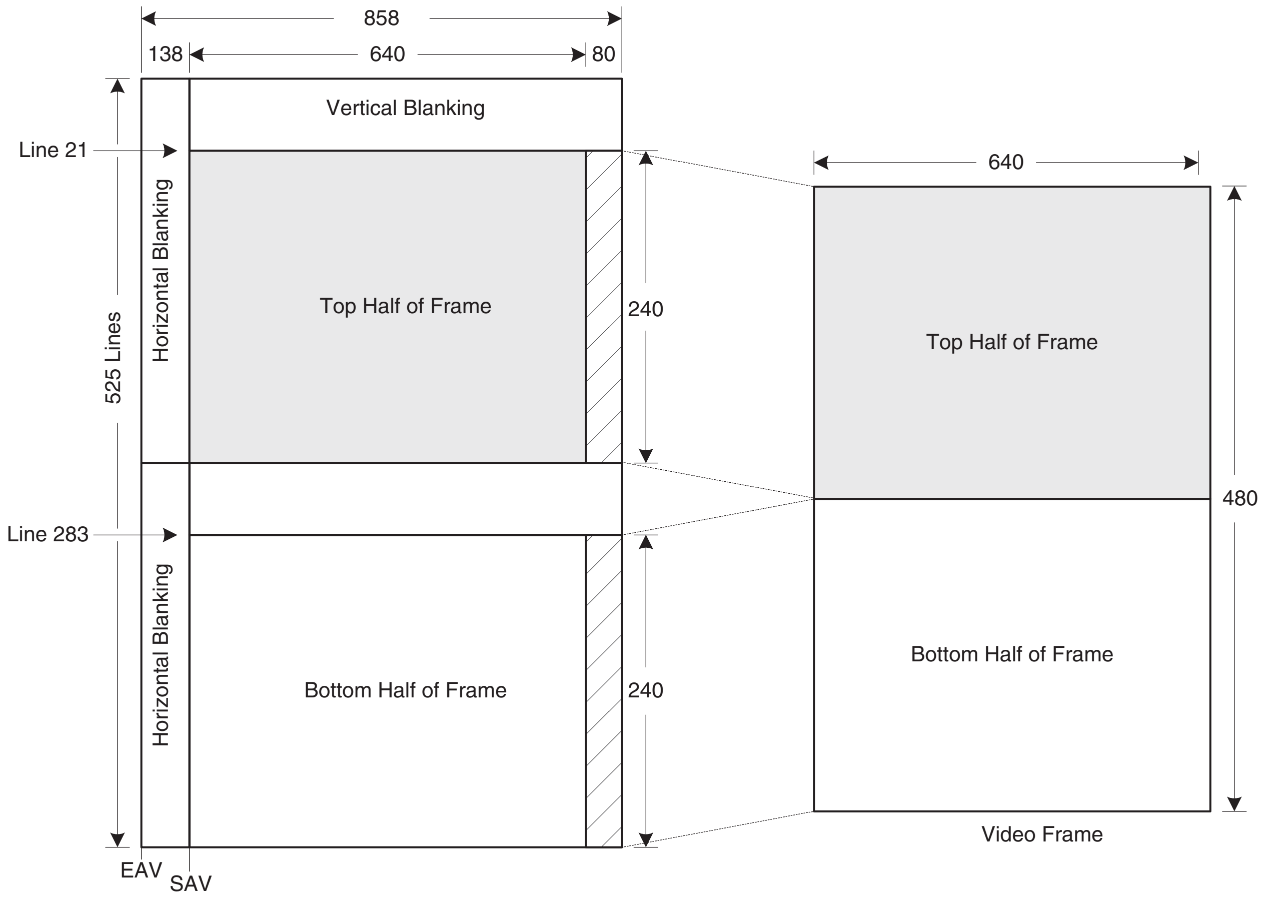


Figure 3-5 640 x 480 Progressive Video

3.1.3 Progressive Video Interface for 800x600 (SVGA) and 768x576.

Because 800x600 and 768x576 video have resolutions that are higher than the standard active region video stream, the MG1264 Codec slightly modifies the timing scheme and uses portions of the blanking region to fit the image. 800x600 video uses the PAL framing because PAL has a high enough resolution to accommodate the 800x600. 800x600 video is limited to a 25 frame per second rate. Figure 3-6 shows the framing for an 800x600 frame.

The MG1264 Codec uses a similar video timing for 768x576 video. Alternately, it can use the exact same video timing as 800x600 and crop the unused portions.

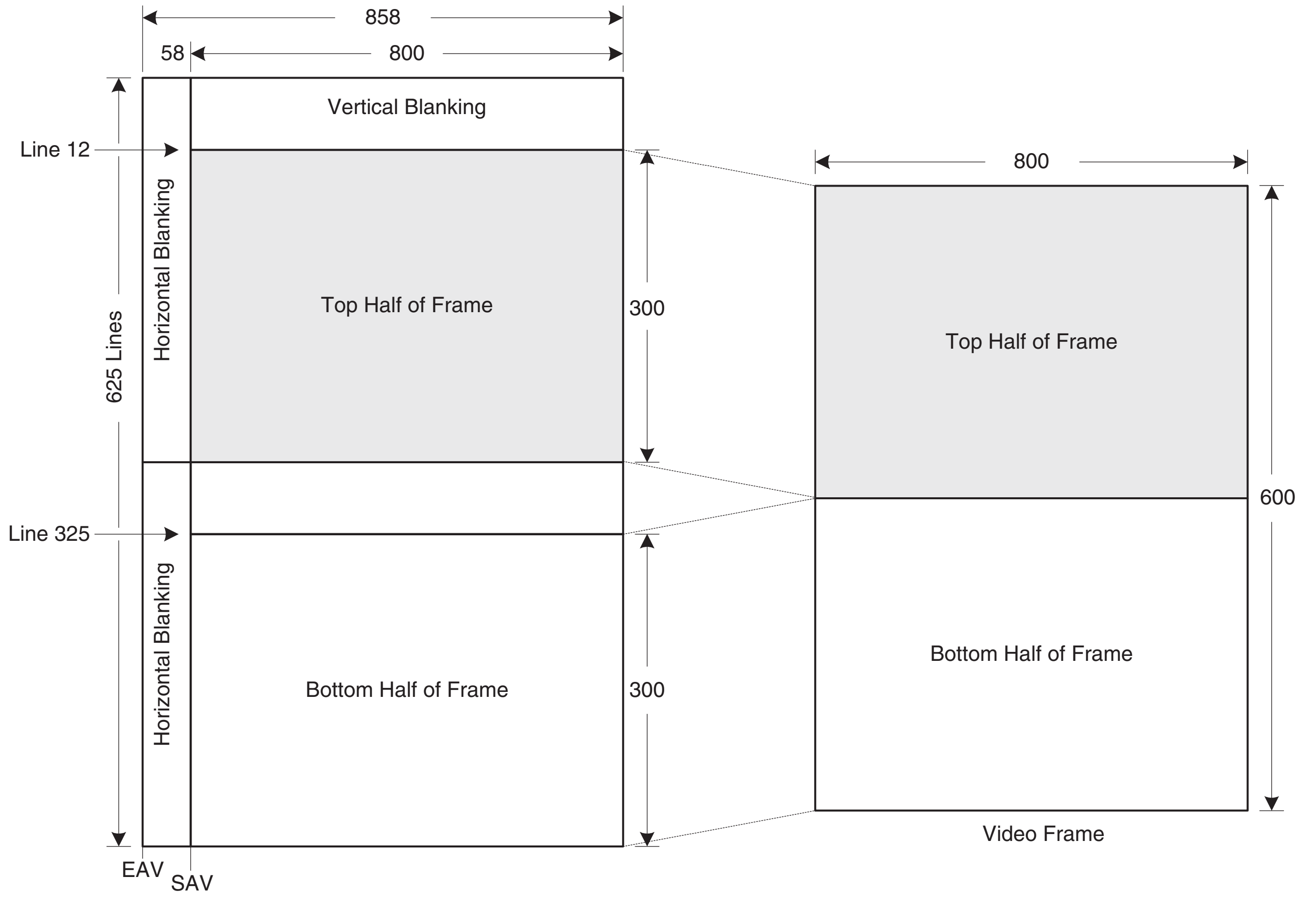


Figure 3-6 800 x 600 (SVGA) Progressive Video

3.2 Video Interface Signals

This section describes the signals used to interface the MG1264 Codec into a system. Table 3-2 shows the signals and Figure 3-7 shows the connections.

Table 3-2 Video Interface Signals

SIGNAL	Dir	# Bits	Description
VID_CLK	IO	1	Video Clock, 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 bus is an input when the MG1264 Codec is sinking the video data (encoding), and an output when the MG1264 Codec is sourcing the video data (decoding).
VIDOUT_DATA[7:0]	O	8	Video Output Data in development only

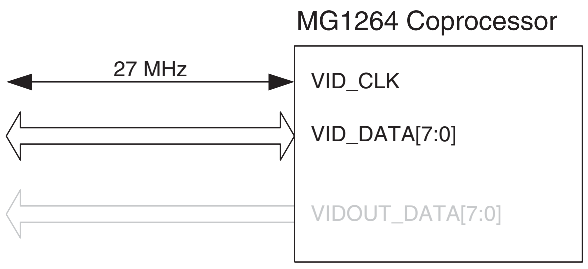


Figure 3-7 Video Interface Connections

3.3 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) as shown in Figure 3-8. 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.

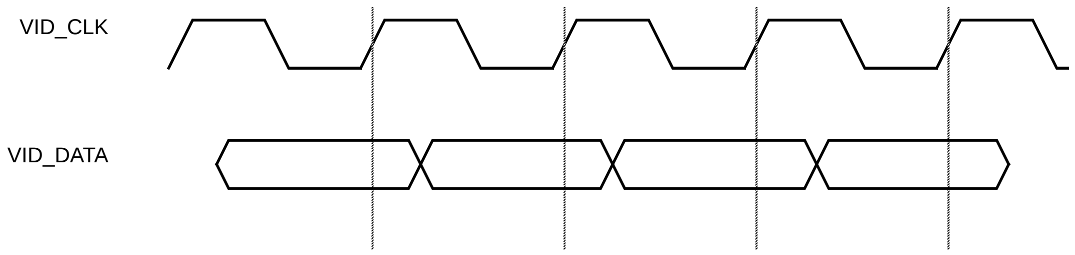


Figure 3-8 Video Interface Timing

Chapter 4. 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 BGA 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.

4.1 The SDRAM Interface

The MG1264 Codec connects to the SDRAM as shown in Figure 4-1. Table 4-1 lists the connections and describes their functions.

Table 4-1 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_ADDR[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 4-1 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

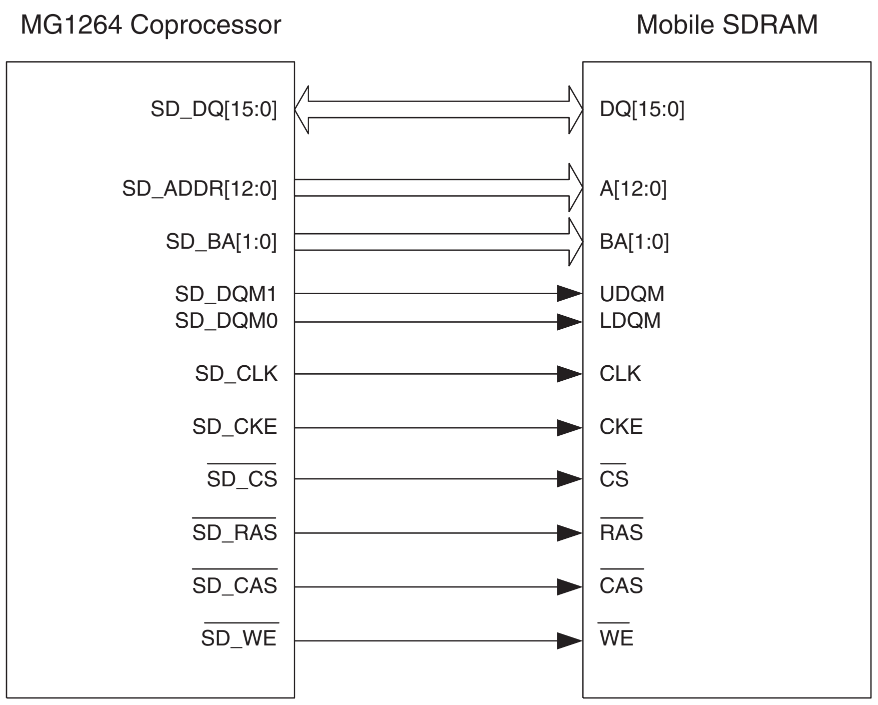


Figure 4-1 MG1264 Codec SDRAM Interface

4.2 Mobile SDRAM Features

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

- Support for 3.3, 2.5, and 1.8 Voltage Operation (Core and I/O)
- Temperature Compensated Self-Refresh
- Partial Array Self Refresh
- Deep Power Down
- Drive Strength Control

4.2.1 Voltage Operation (3.3V, 2.5V, 1.8V)

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 and 1.8V. The MG1264 Codec supports both the 3.3V and 2.5V options.

4.2.2 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.

4.2.3 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.

4.2.4 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 5. Audio Interface

5.1 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 kHz, 44.1 kHz, 32 kHz, 24, kHz, and 22.05 kHz.

The MG1264 Codec encodes two-channel AAC audio encoding with 16-bit samples at both 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 5-1 shows the key features and their associated target settings.

Table 5-1 AAC Encoder Features

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

5.2 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 5-2.

Table 5-2 Audio Interface Signal List

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

Since the MG1264 Codec is an audio Slave, the clocks have to be supplied by either the System Host CPU (refer to Figure 5-1) or the audio DAC/ADC (refer to Figure 5-2).

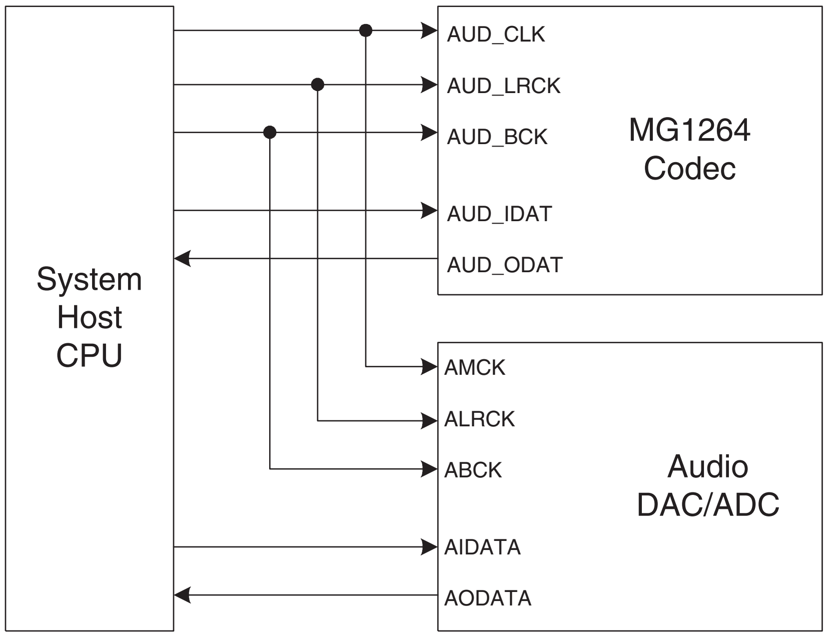


Figure 5-1 Audio Interface Connections with the System Host CPU as the Audio Clock Master

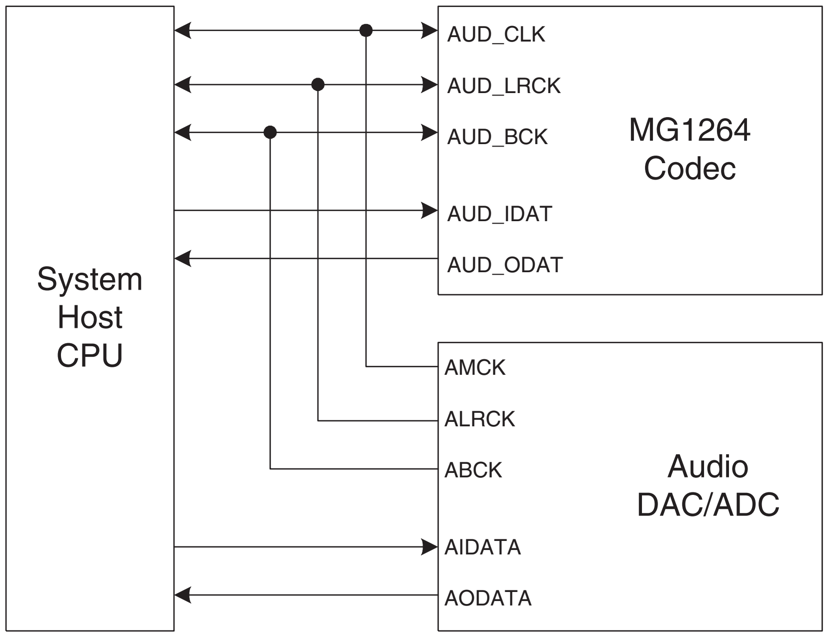


Figure 5-2 Audio Interface Connections with the DAC/ADC as the Audio Clock Master

5.3 I^2S Audio Waveforms

A sample waveform for I^2S audio is shown in Figure 5-3. 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)

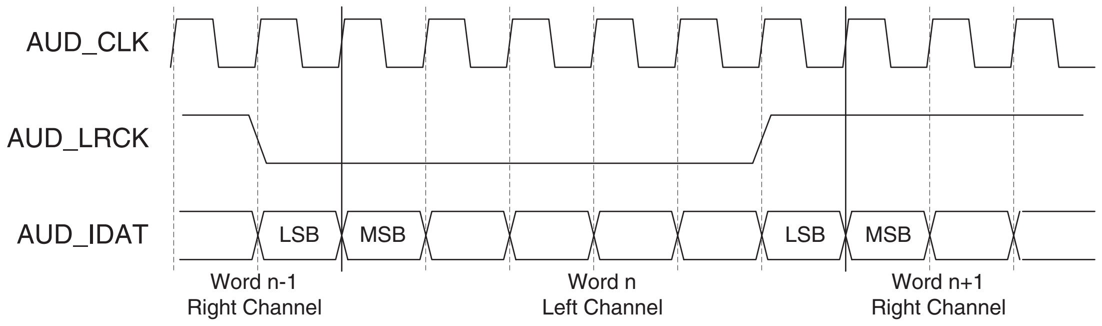


Figure 5-3 I^2S Left-justified Audio Waveform

5.4 Left Justified Audio Waveform

A sample waveform for Left Justified audio shown in Figure 5-4. 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)

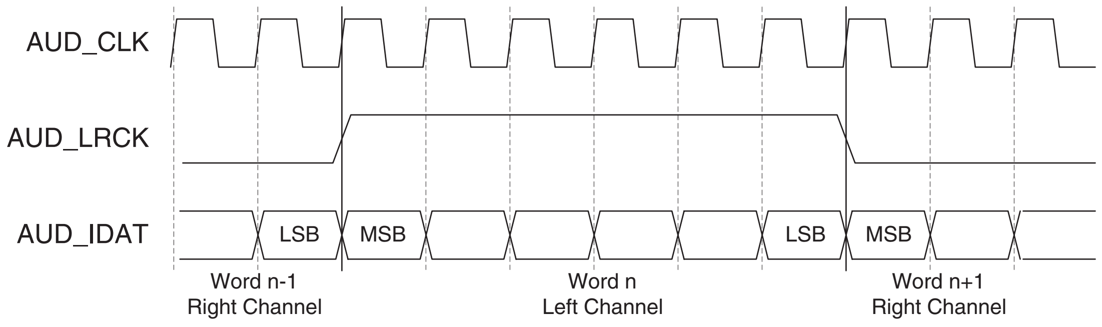


Figure 5-4 Left-justified Audio Waveform

5.5 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 5-5. 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.

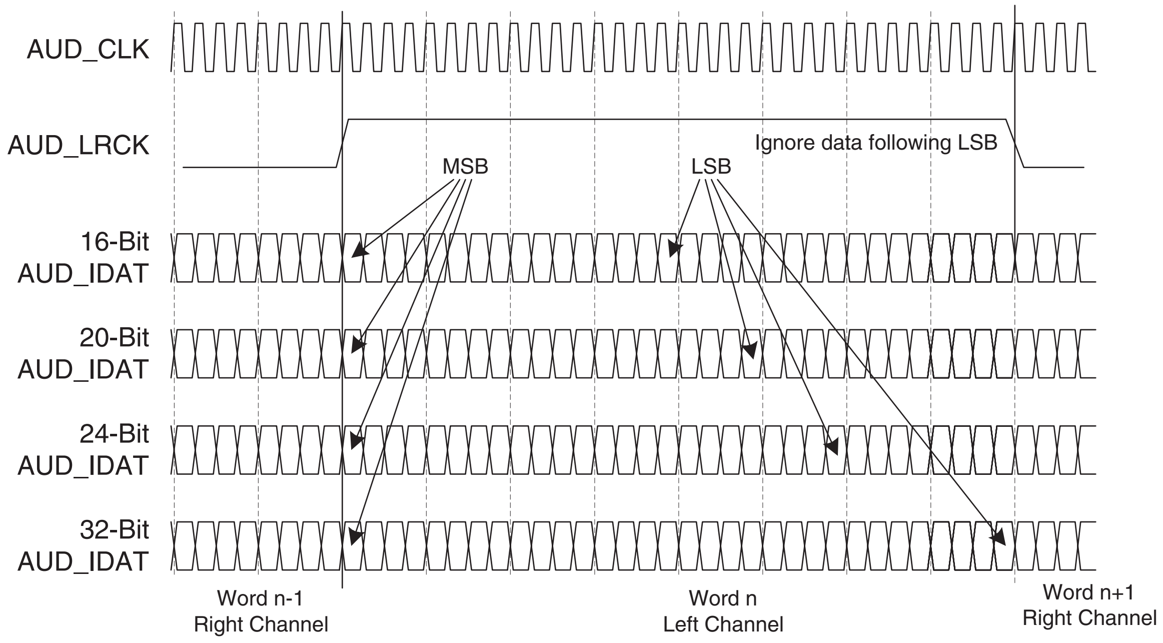


Figure 5-5 16, 20, 24, and 32-Bit Left Justified Audio Waveform

Chapter 6. Miscellaneous Signals

There are many signals on the MG1264 Low Power H.264 and AAC Codec for Mobile Devices that are not associated with a specific interface. These signals are described in Table 6-1.

Table 6-1 Miscellaneous Signals

Signal	Dir	# Bits	Description
SOUT	O	1	UART Transmit Data
SIN	I	1	UART Receive Data
XIN	I	1	Clock Input
PLL_AVDD	P	1	PLL Analog VDD
PLL_AVSS	P	1	PLL Analog VSS
CORE_VDD	P	4	Core Logic VDD
CORE_VSS	P	4	Core Logic VSS
TCK	I	1	Test JTAG Clock
TDI	I	1	Test JTAG Data Input
TDO	O	1	Test JTAG Data Output
$\overline{\text{TRST}}$	I	1	Test JTAG Reset
TMODE	I	1	Test JTAG Mode

Chapter 7. Programming

7.1 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.

7.2 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 85.

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.

7.3 Encode and Decode Mode

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

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 76.

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 DMAREQ 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 73.

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 71.

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 currently unimplemented.

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 73.

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 73.

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 5 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 78.

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 79.

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 five 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 78.

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 79.

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 in mobile 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 DRAM, read/write 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 the 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 structure of the QHAL is shown in Figure 10-1.

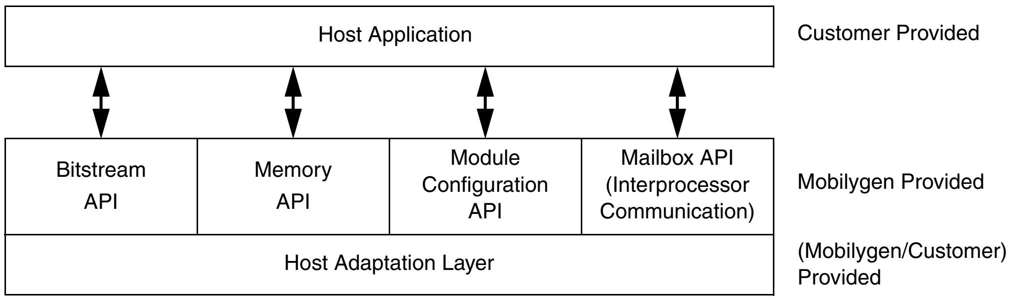


Figure 10-1 QHAL Structure

10.1.1 QHAL_EM

The *qhal_em* is the external DRAM driver. 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 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 QCC 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_READY 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_READY interrupts are generated using *qhalmbox_read* operations. The meaning of COMMAND_DONE and EVENT_READY are explained in “Command Transfer Protocol” on page 98.

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 (called the System Input Stream Controller or SISC). 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 without endianness conversion. Refer to “H.264/ACC Decoder Interface Object” on page 111 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." Events are stored in the 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 98 and "Reading Events from the Media Processor Firmware" on page 99.

10.2.3 Global Pointer Block

There are a number of data structures stored in the 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;
    void                            *reserved;
    int                             productConfiguration;
    void                            *meStatus;
} 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 external 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 99. 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.

The product configuration word is used by the System Host CPU to control how the firmware initializes itself. The System Host CPU write overrides the contents of the this field during the boot loading phase of the firmware (with the *qmmloader* application). Details concerning the specific product configurations are contained in a separate application note.

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.

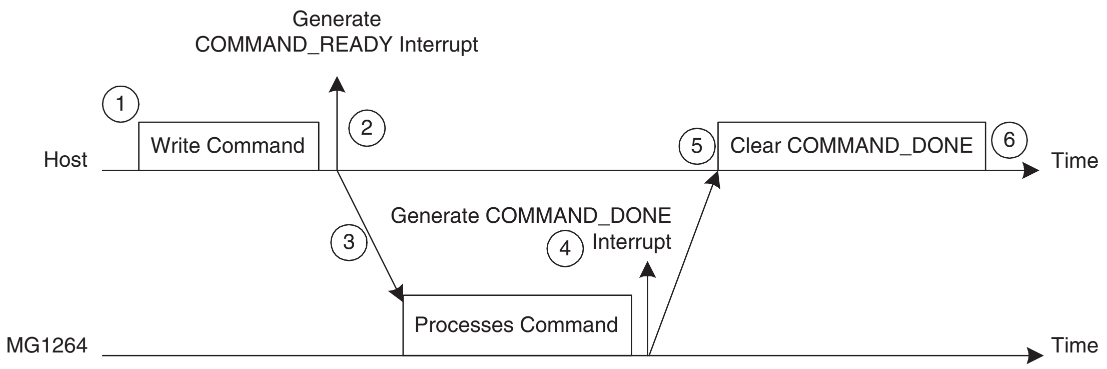


Figure 10-2 Command Transfer Timing

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 will only see 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.

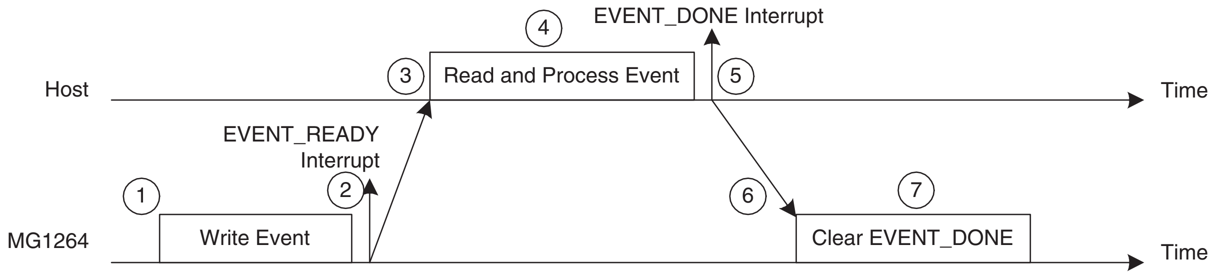


Figure 10-3 Event Transfer Timing

The complete Event Transfer protocol is:

1. The Media Processor firmware writes the event ID, control ID, and payload to the event block.
2: The Media Processor firmware writes to the mailbox register to assert the EVENT_READY interrupt and clear the EVENT_DONE interrupt.
3: 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.
4: The System Host CPU processes the event.
5: 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.
6: The Media Processor firmware waits for and receives the EVENT_DONE interrupt.
7: The Media Processor firmware clears the EVENT_DONE interrupt.

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

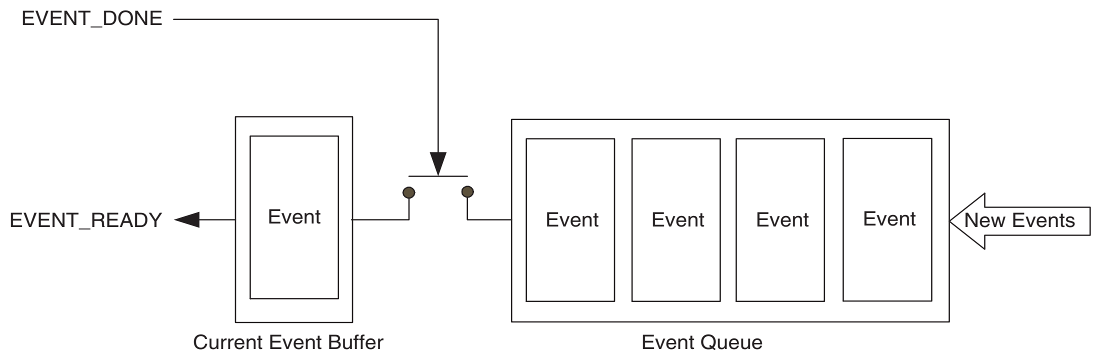


Figure 10-4 Event Queuing

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 even 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 System Host CPU application will subscribe to a set of events at start-up.

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 even 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.

A configuration parameter has a unique ID and an associated 32-bit value. The 32-bit value can include multiple bit fields. Configuration parameters are set using the CONFIGURE command, which has the same opcode for all control objects. Parameters can only be changed when the target control object is in an IDLE state.

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 configuration parameters are set using the CONFIGURE command. Each parameter has a 32-bit value associated with it that is stored by the firmware.

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 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 standard read cycles. 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. 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 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.

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_type: 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 codec demultiplexer.

cts: Sample composition time in 90 kHz ticks.

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. Note that when the decoder is in elementary video mode, it cannot accept or generate compressed audio data at the same time.

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 and video input or output, and the OSD display.

For the OSD Bitmap functions, these restrictions are in the APIs:

1. Bitmap Format Contraints:
    - Biplanes = 0
    - BiBitCount = 8; 8 bpp
    - BiCompression = 0; no compression

2: Width of the Bitmap and OSD Screen Size must be multiple of 4

3: Start position for the OSD destination screen has to be multiple of 4

4: Default Bitmap Index for "Transparent"

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
ID	1
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
ID	5
Arguments	0 = To exit 1 = To enter sleep
Return Code	Cannot be checked, see below
Return Values	None
Valid States	All
Description	The POWERDOWN command is used to transition the 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 in an auto-refresh state. The command cannot fail and it is assumed that if a COMMAND_DONE interrupt is received, the command was accepted.

10.4.5 Configuration Parameters

AUDIO_NUM_CHANNELS

Parameter	Q_SYS_CFG_AUDIO_NUM_CHANNELS
ID	6
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
ID	7
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
ID	8
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
ID	10
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
ID	9
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.

10.4.6 Events

Q_SYS_EV_HEARTBEAT

Event	Q_SYS_EV_HEARTBEAT
ID	0x10001
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

Event	Q_SYS_EV_ECHO
ID	0x10002
Payload	0 = Value of the first argument to 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.

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.

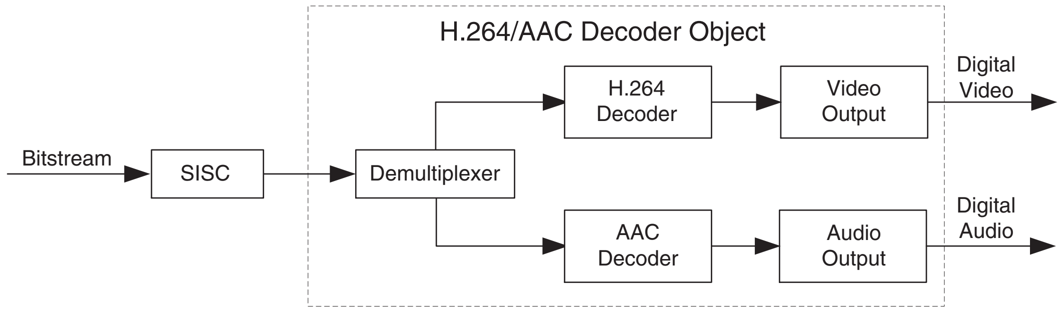


Figure 10-5 Idealized Decoder Datapath

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.

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.

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 play-back.

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 132.

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 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).

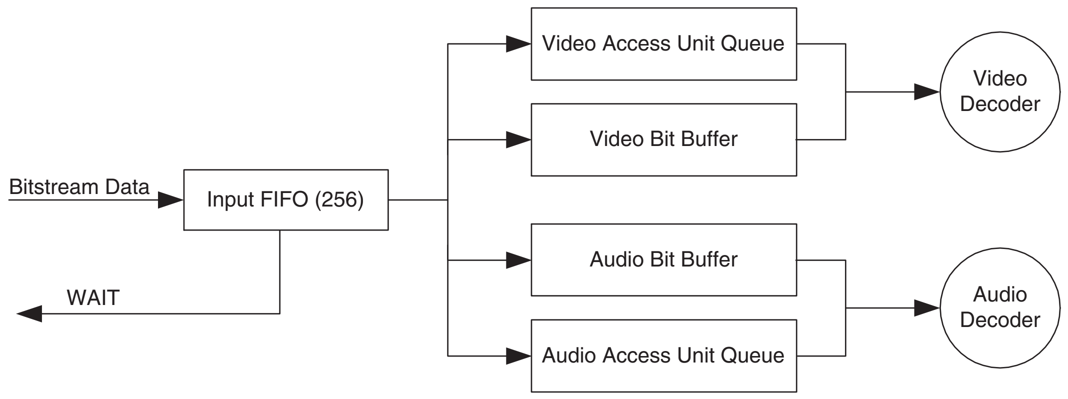


Figure 10-6 Decoder Buffer Structure

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.

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 106. 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)
          {
            if (videoFIFOEmptiness - qboxSize < 0)
            {
               break;
            }
            if (videoQueueEmptiness == 0)
            {
               break;
            }
            videoQueueEmptiness--;
            videoFIFOEmptiness -= qboxSize;
          }
          else if (qboxType == AUDIO_QBOX)
          {
            if (audioFIFOEmptiness - qboxSize < 0)
            {
               break;
            }
            if (audioQueueEmptiness == 0)
            {
               break;
            }
            videoQueueEmptiness--;
            videoFIFOEmptiness -= qboxSize;
          }

          // Calculate the padded size of a qbox to 32 bit bound-
ary
          paddedQboxSize = (qboxSize + 3) & 0xfffffffc;

          // set the flag in the qbox header saying there is pad-
ding

          // Send paddedQboxSize bytes to the codec

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

It is important to note the calculation at the end of the loop of the *paddedQboxSize*. Because the MG1264 Codec's Host Interface is implemented as a 16-bit bus without individual byte enables, it is not possible to send an odd number of bytes. Additionally, some devices have an internal limitation of being only able to send 32-bit word sized data. In order to manage these limitations, the QBox header has a flag (see "Bitstream Formats" on page 104) in the box flags that indicate that the QBox being sent is padded to 32-bit alignment. If this flag is set, the padding bytes are automatically dropped by the QBox demultiplexer.

10.6.5 Object ID

The H.264/AAC decoder object ID is 2.

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, it is often required to flush the data pipeline on the MG1264 Codec 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 bit-stream. 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
ID	1
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
ID	2
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 direction 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 132 for more information. As described in the state transition tables on page 118, 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
ID	20
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 which were causing any bitstream sending to block.

I-FRAME_PLAY

Command	Q_AVD_CMD_IFRAME_PLAY
ID	4
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
ID	3
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
ID	19
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
ID	5
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
ID	6
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
ID	7
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
ID	8
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
ID	11
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
ID	14
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 113 for additional information.

AUDIO_BUFFER_EMPTINESS

Command Name	Q_AVD_CMD_AUDIO_BUFFER_EMPTINESS
ID	15
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 System Host CPU to query the firmware about the emptiness of the audio 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 113 for additional information.

VIDEO_DISPLAY_RECT

Command Name	Q_AVD_CMD_VIDEO_DISPLAY_RECT
ID	17
Arguments	[0] Width [1] Height [2] X Offset [3] Y Offset
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Any
Description	The VIDEO_DISPLAY_RECT is used to set the video display area relative to active video. A typical video display will be one of 720x480, 640x480, or 720x576. All typical is a X and Y offset of (0,0). However, the display rectangle can be used to achieve other effects such as the display of 640x480 on a 720x480 physical display by setting the rectangle to 640x480 with an X offset of 40 and a Y offset of 0. In a typical operation, the System Host CPU sets the display rectangle at initialization time and does not change it. The display rectangle supports generalized up-scaling but can downscale only the following ratios: 480/576, ½, and ¼. For example, the ¼ can be used to create a 160x120 thumbnail of a 640x480 image.

VIDEO_ZOOM

Command Name	Q_AVD_CMD_VIDEO_ZOOM
ID	18
Arguments	[0] Source Size as a 16-bit fraction [1] Source X Offset as a 16-bit fraction [2] Source Y Offset as a 16-bit fraction
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	Any
Description	The VIDEO_ZOOM is used to perform an arbitrary horizontal and vertical crop of the source material and then scaled to fit the display rectangle. The size, X offset, and Y offset are all specified as 16-bit fractions (so that 65536/2 is ½, 65536/4 is ¼ etc.) and specify the crop window. The x and y coordinates of the crop window are similarly specified by the X and Y offsets relative to the top left corner of the source. An X and Y off-set of (0,0) means the crop window is positioned at the top-left, an X and Y offset of (½, ¼) means that the crop window is positive at ½ the width and ¼ the height from the top.

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
ID	4
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
ID	5
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
ID	13
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
ID	14
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
ID	15
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
ID	16
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

VOUT_SCALING_ENABLE

Parameter	Q_AVD_CFG_VOUT_SCALING_ENABLE
ID	17
Values	Setting to 1 enables the output scaler, setting to 0 disables it.
States	Any
Effective	Immediately.
Description	This variable is used to enable or disable the video output scaler. If the scaler is enabled, it will automatically resize the decoded video to fit the display rectangle. Note that upscaling is arbitrary in that any smaller image size can be fit to the display rectangle, but downscaling is only supported for fixed ratios. These fixed ratios are 720 to 640 horizontally and 576 to 480 vertically.

10.6.9 Events

Q_AVD_EV_VIDEO_DECODER_ERROR

Event	Q_AVD_EV_VIDEO_DECODER_ERROR
ID	0x20003
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
ID	0x20004
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
ID	0x20001
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
ID	0x2000A
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
ID	0x2000E
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
ID	0x2000F
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
ID	0x20007
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
ID	0x20005
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.

10.6.10 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 which were dropped (not displayed) due to audio or video synchronization requirements.

avsyncVideoRepeats

This field contains the number of video frames which were repeated due to audio or video synchronization requirements.

10.6.11 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 133.

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.

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.

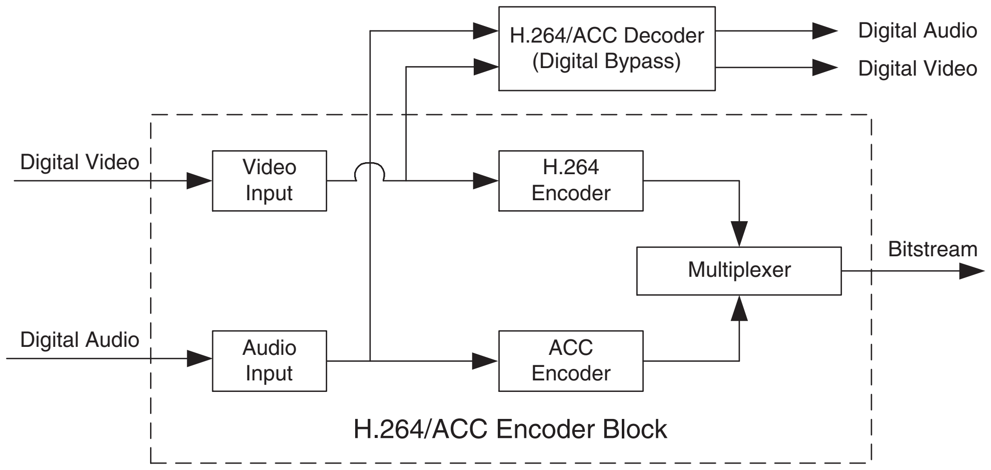


Figure 10-7 Idealized Encoder Datapath

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 can encode AAC 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 in half to create 15 fps sequences.

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. Together, 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”.

10.7.4 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.

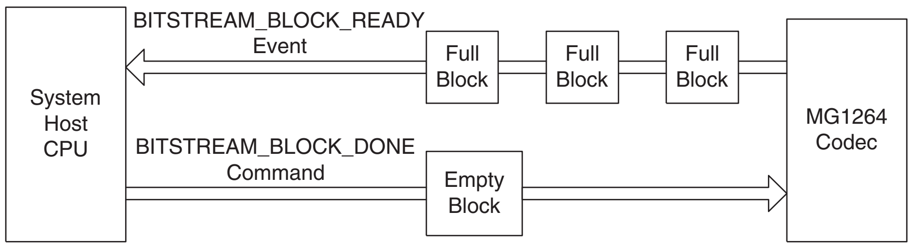


Figure 10-8 Circular Buffer Management of Bitstream Blocks

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* function as 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 inttimestamp;
     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 present in the QBox header, but is not available in elementary video mode. The timestamps are separated by the sample duration, which is the reciprocal of the frame rate expressed in 90 kHz ticks:

- For audio frames, this corresponds to (samples per frame/sampling rate) * 90000
- For 24 kHz, this is (1024/24000) * 90000 = 3840
- For 48 kHz it is (1024/48000) * 90000 = 1920
- 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.

10.7.5 Controlling the Video Bitrate

The MG1264 Codec runs multiple concurrent algorithms that are used to control the video bitrate. All of these algorithms attempt to manage the bitrate and quality of the video stream so that they do not violate certain constraints. These constraints can be selectively enabled or disabled, and can run concurrently.

Bitstream-Buffer Constraint

The bitstream buffer algorithm ensures that the bitrate does not overflow the memory buffers on both the MG1264 Codec and the System Host CPU. The algorithm is essentially a leaky-bucket model that prevents overflow. The fill-rate of the bucket is the actual bitrate of the video bitstream, which over the long-term should match the target average bitrate but can vary over the short-term. The drain-rate of the bucket is set by the System Host CPU as is the size of the buffer in bits. The bucket accumulates bits whenever the short-term fill-rate exceeds the drain-rate. The bucket loses bits whenever the short-term fill-rate is smaller than the drain-rate.

In storage applications (writing the bitstream to a Flash card for instance), the drain rate of the bucket should be set to the maximum sustained transfer rate of the System Host CPU from the

MG1264 Codec's memory to the Flash card. Often this rate is significantly higher than the average bitrate of the video.

Note that the constraint can take a buffer size as a parameter that does not match the actual internal buffer of the MG1264 Codec (which is 16 Mbits). The buffer size can be set larger if there is buffer space available on the System Host CPU, or it can be set smaller if the System Host CPU wishes to constrain the bitrate swings. Similarly, the bucket drain-rate can be set smaller than the actual transfer rate in order to reduce the bitrate swings.

Peak Video Quality Constraint

The peak video quality algorithm attempts to save bits on very easy-to-code scenes by setting a maximum image quality (as measured in dB). The maximum measurable SNR is 48.13, which corresponds to a single bit out of 8 bits (10log(255^2)). Therefore, setting a maximum SNR of 49 or greater is essentially the same as disabling the constraint. However, by assuming a maximum SNR of less than 48, the algorithm will prevent the bitrate from being increased once the SNR has reached the specified level. This has the net effect of saving bits for later use in more difficult scenes.

The System Host CPU should set the peak video quality to a very high level to ensure that quality is not compromised on typical video. Therefore, a value of 43 or greater is recommended.

File-Size Constraint

The file size constraint algorithm ensures that the actual file size matches the target file size (as measured by the average bitrate multiplied by time) within a certain margin that is specified by the System Host CPU. This margin is both positive and negative. For example, the System Host CPU could request that the actual file size not exceed the target by 12 Mbits (1.5 MBytes).

Note that the actual file size can be lower than the specified margin if the peak video quality constraint is being used. In the case of very easy-to-code scenes where the SNR exceeds the maximum SNR, the file size constraint will not attempt to force the bitrate higher.

10.7.6 Object ID

The H.264/AAC encoder object ID is 0x3.

10.7.7 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.

10.7.8 Commands

FLUSH

Command Name	Q_AVE_CMD_FLUSH
ID	1
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
ID	2
Arguments	None
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.

PAUSE

Command Name	Q_AVE_CMD_PAUSE
ID	3
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
ID	4
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.

VIDEO_CAPTURE_RECT

Command Name	Q_AVE_CMD_VIDEO_CAPTURE_RECT
ID	20
Arguments	[0] Rectangle width [1] Rectangle height [2] Rectangle X offset [3] Rectangle Y offset
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command changes the video input module's capture rectangle. This rectangle is relative to the start of active video and can be used to crop part of the input (no scaling is done). Typical configurations will have the X and Y offsets set to (0x0), and the capture rectangle set to one of 720x480, 720x576, or 640x480. The rectangle is typically set at initialization time, but it can be used if the sensor is reconfigured to do VGA or QVGA for example.

BITSTREAM_BLOCK_DONE

Command Name	Q_AVE_CMD_BITSTREAM_BLOCK_DONE
ID	5
Arguments	0 = Block address 1 = Block size
Return Codes	0 = Failure 1 = Success
Return Values	None
Valid States	All
Description	This command indicates to the firmware that a specific bitstream block has been read by the System Host CPU and is now free to be overwritten by new data. The address and data implicitly indicate to the firmware whether the bitstream block is an audio or video block.

RC_BUFFER_MODEL

Command	Q_AVE_CMD_RC_BUFFER_MODEL
ID	23
Arguments	[0] 1 to enable, 0 to disable [1] Buffer size in bits [2] Buffer drain rate
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	Idle
Description	This command changes the video bitrate control parameters for the buffer model. The System Host CPU can enable or disable the constraint and can change the size and drain-rate of the buffer. For storage applications, the buffer drain rate is typically equal to the maximum sustained rate at which data can be stored to the memory device. As the transfer rate increases, the harder it is to fill the buffer and therefore, the higher bitrate swings can be. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

RC_FILE_SIZE

Command	Q_AVE_CMD_RC_FILE_SIZE
ID	22
Arguments	[0] 1 to enable, 0 to disable [1] Maximum file size deviation in bits
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	Idle
Description	This command changes the video bitrate control's parameters for managing file size. The System Host CPU can enable or disable this constraint and can control how tightly the MG1264 Codec controls the bitrate. When enabled, the algorithm will try to control the actual file size from the target file size to a maximum deviation specified in this command. This deviation applies to both positive and negative sizes (bigger or smaller file sizes). Note that the actual file size can be smaller than the specified deviation if the maximum SNR constraint is enabled and very easy content (low-bitrate) is being encoded. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

RC_SNR_LIMIT

Command	Q_AVE_CMD_RC_SNR_LIMIT
ID	24
Arguments	[0] 1 to enable, 0 to disable [1] Maximum SNR in dB
Return Code	0 = Failure 1 = Success
Return Values	None
Valid States	Idle
Description	This command changes the video bitrate control parameters for managing peak quality. Any SNR value greater than or equal to 49 is essentially infinity. It is recommended to use a value greater than or equal to 43. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

10.7.9 Configuration Parameters

These parameters can only be set when the encoder interface object is in an IDLE state and takes 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_AVE_CFG_BITSTREAM_TYPE
ID	1
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 set-up when the system is in the IDLE state.

NUMBLOCKSPEREVENT

Parameter	Q_AVE_CFG_NUMBLOCKSPEREVENT
ID	56
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.

AV_SELECT

Parameter	Q_AVE_CFG_ENC_AV_SELECT
ID	18
Value	1 = Q_AVE_CFP_ENC_AV_SELECT_AV 2 = Q_AVE_CFP_ENC_AV_SELECT_VIDEO_ONLY 3 = Q_AVE_CFP_ENC_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 encoding, audio-only encoding, or audio and video encoding.

VENC_BITRATE

Parameter	Q_AVE_CFG_VENC_BITRATE
ID	21
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 video stream. The encoder will attempt to meet this bitrate constraint over 90 second intervals. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

AENC_ BITRATE

Parameter	Q_AVE_CFG_AENC_BITRATE
ID	35
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.

VIN_PROG_SOURCE

Parameter	Q_AVE_CFG_VIN_PROG_SOURCE
Value	0 = Interlace video input 1 = Progressive video input
ID	17
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter indicates to the video encoder whether the video capture unit is receiving progressively-scanned video or interlace-scanned video. If the source is progressive, then only video frame pictures will be created. If the source is interlaced, then the VENC_FIELD_CODING parameter controls the picture coding type.

VENC_FIELD_CODING

Parameter	Q_AVE_CFG_VENC_FIELD_CODING
ID	34
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	In the case where 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.

VIDEO_INPUT_STANDARD

Parameter	Q_AVE_CFG_VIDEO_INPUT_STANDARD
ID	51
Value	1 = Q_AVE_CFP_VIDEO_INPUT_STANDARD_NTSC 2 = Q_AVE_CFP_VIDEO_INPUT_STANDARD_PAL
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter selects the video input standard, either NTSC or PAL. The System Host CPU must also set the correct capture rectangle using the VIDEO_CAPTURE_RECT command.

VENC_GOP_SIZE

Parameter	Q_AVE_CFG_VENC_GOP_SIZE
ID	24
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.

VIN_FRAMERATE_DECIMATION

Parameter	Q_AVE_CFG_VIN_FRAMERATE_DECIMATION
ID	32
Value	1 or 2
Valid States	IDLE
Effective	On the next AV encoder state transition out of IDLE
Description	This parameter sets the frame rate decimation ratio for the video input. If the decimation rate is 1, then no frame rate decimation is done. If the value is 2, then the frame rate is decimated by taking every other video frame. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

VIN_DECIMATION_H

Parameter	Q_AVE_CFG_VIN_DECIMATION_H
ID	13
Value	1 or 2
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 1, then no scaling is done and the encoded pictures will have the width of the VIDEO_CAPTURE_RECT. If the ratio is 2, then the video is horizontally downscaled by a factor of 2 and to the nearest multiple of 16. For example, 640 pixel-wide video is downscaled to 320 wide, and 720 wide video is downscaled to 352 pixels. The values set by this command are reset by setting the VENC_OPERATIONAL_MODE configuration parameter.

VIN_DECIMATION_V

Parameter	Q_AVE_CFG_VIN_DECIMATION_V
ID	14
Value	1 or 2
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 1, then no scaling is done and the encoded pictures will have the height of the VIDEO_CAPTURE_RECT. If the ratio is 2, then the video is vertically downscaled by a factor of 2 and to the nearest multiple of 16. For example, 480 pixel-high video is downscaled to 240 high, and 576 high video is downscaled to 288 pixels. 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
ID	10
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
ID	19
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
ID	39
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
ID	53
Value	0 is mute off, 1 is 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 encoded, although the encoded video frames will be black.

AUDIO_MUTE

Parameter	Q_AVE_CFG_AUDIO_MUTE
ID	54
Value	0 is mute off, 1 is 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.

OUTSAMPLE_ALIGN

Parameter	Q_AVE_CFG_OUTSAMPLE_ALIGN
ID	59
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.

10.7.10 Events

Q_AVE_EV_BITSTREAM_BLOCK_READY

Event	Q_AVE_EV_BITSTREAM_BLOCK_READY
ID	0x30001
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. Note that the first three payload entries are reserved.

Q_AVE_EV_BITSTREAM_FLUSHED

Event	Q_AVE_EV_BITSTREAM_FLUSHED
ID	0x30003
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
ID	0x30005
Payload	None
Description	This event is generated once for every audio frame that is encoded.

Q_AVE_EV_AUDIO_FRAME_ENCODED

Event	Q_AVE_EV_AUDIO_FRAME_ENCODED
ID	0x30004
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
ID	0x30009
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
ID	0x3000A
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.

10.7.11 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. Specifications

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

Note: The information contained in this section is based on simulation values and early characterization of prototype parts. As such, it should be considered **Preliminary** data. For the most current values, contact the Mobilygen technical support department.

11.1 Electrical Characteristics

This section specifies the electrical characteristics of the MG1264 Codec.

11.1.1 Absolute Maximum Ratings

Table 11-1 gives the absolute maximum ratings. Exposure to stresses beyond those listed in this table may result in device unreliability, permanent damage, or both.

Table 11-1 Absolute Maximum Ratings

Parameter	Value	Units	Notes
CVDD	1.6	V	—
VDDP	1.6	V	—
VDD_IO	4.5	V	Applies to both SD and non-SD VDDO_IO
Maximum Input Voltage	IO_VDD + 0.3	V	Referenced to associated VDD_IO
Storage Temperature Range	-40 to 150	°C	—
Operating Temperature Range (case)	0 to 125	°C	—

11.1.2 Operating Conditions

Table 11-2 specifies the operating conditions for the MG1264 Codec.

Table 11-2 Operating Conditions

Parameter	Minimum	Maximum	Units
CVDD	1.08	1.32	V
VDDP	1.08	1.32	V
VDD_IO	2.97	3.63	V
VDD_SD_IO	2.25	3.63	V
$T_{Ambient}$	0	70	°C

11.1.3 DC Characteristics

Table 11-3 defines the DC characteristics.

Table 11-3 DC Characteristics

Symbol	Parameters	Test Conditions	Min	Max	Units
V_{IH}	Input High Level	V_{DD} = Maximum	0.7* VDD_IO	—	V
V_{IL}	Input Low-Level Voltage	V_{DD} = Minimum	—	03* VDD_IO	V
V_{OH}	Output High-Level Voltage	V_{DD} = Minimum, I_{OH} = –0.1 mA	0.9* VDD_IO	—	V
V_{OL}	Output Low-Level Voltage	V_{DD} = Minimum, I_{OL} = 0.1 mA	—	0.1* VDD_IO	V
I_{IH}	Input Leakage	V_{DD} = Maximum, $V_{IN} = V_{DD}$	–5	5	µA
I_{IL}	Input Leakage	V_{DD} = Maximum, V_{IN} = 0V	–5	5	µA
I_{OZ}	TriState Leakage	V_{DD} = Maximum, V_{IN} = 0V – VDD_IO	–5	5	µA
IDD_{Core}	Core Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	175	mA
IDD_{IO}	I/O Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	5	mA
IDD_{SD_IO}	SD I/O Supply Current	V_{DD} = Maximum, Frequency = 81 MHz	—	20	mA
C_{PIN}	Capacitance[a]	—	—	5	pF

a.Not 100% tested.

11.1.4 Power Supply Pin Voltages

This section provides the recommended voltages for the power supply pins.

11.1.5 Power-Up and Power-Down Constraints

This section provides the recommended power-up and power-down constraints.

11.2 AC Timing

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

- "MG1264 Codec Host Interface Timing" on page 157
- "Video Interface AC Timing" on page 160
- "Audio Interface AC Timing" on page 161
- "SDRAM Interface AC Timing" on page 163

11.2.1 MG1264 Codec Host Interface Timing

Figure 11-1 shows the timing diagram for the MG1264 Codec Host Interface, Figure 11-2 shows the DMA Timing, Figure 11-3 shows the Wait timing, and Figure 11-4 shows the Interrupt Request timing. Table 11-4 lists the timing parameters for each of these diagrams.

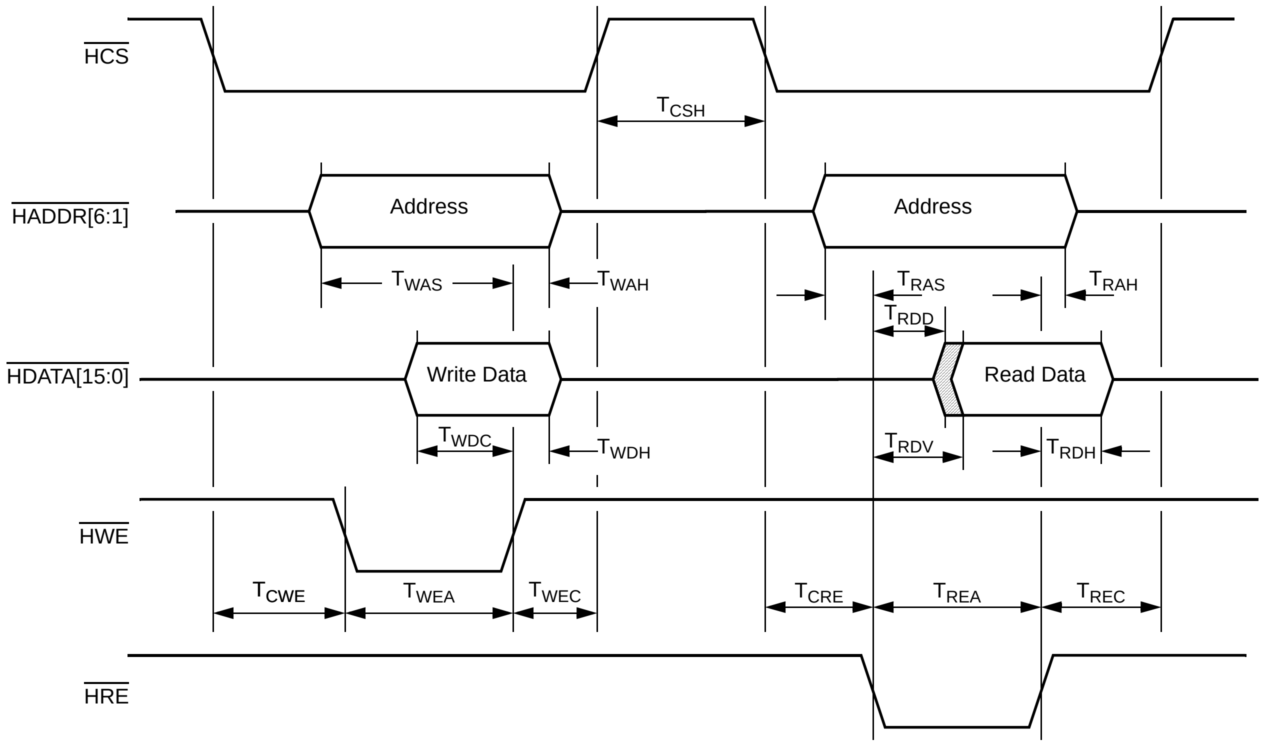
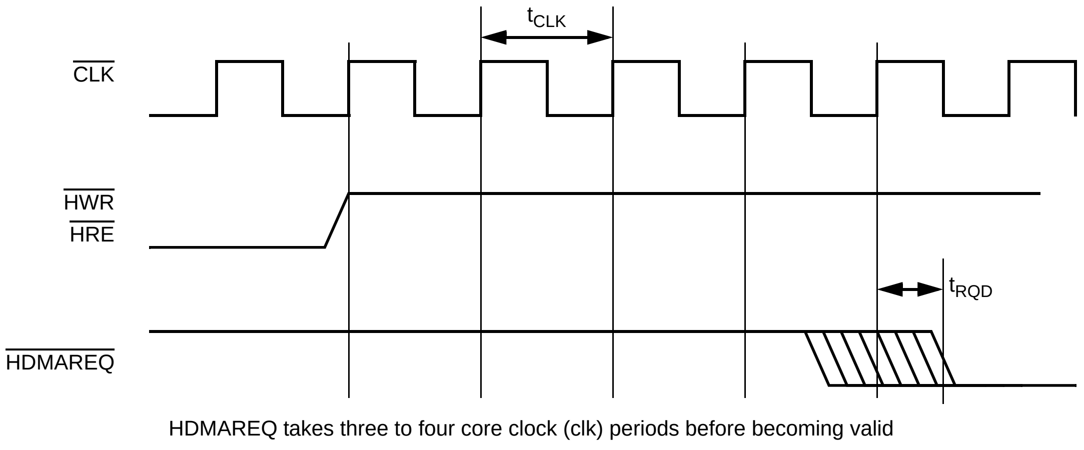


Figure 11-1 MG1264 Codec Host Interface AC Timing Waveform



HDMAREQ takes three to four core clock (clk) periods before becoming valid

Figure 11-2 MG1264 Codec $\overline{\text{HDMAREQ}}$ Timing

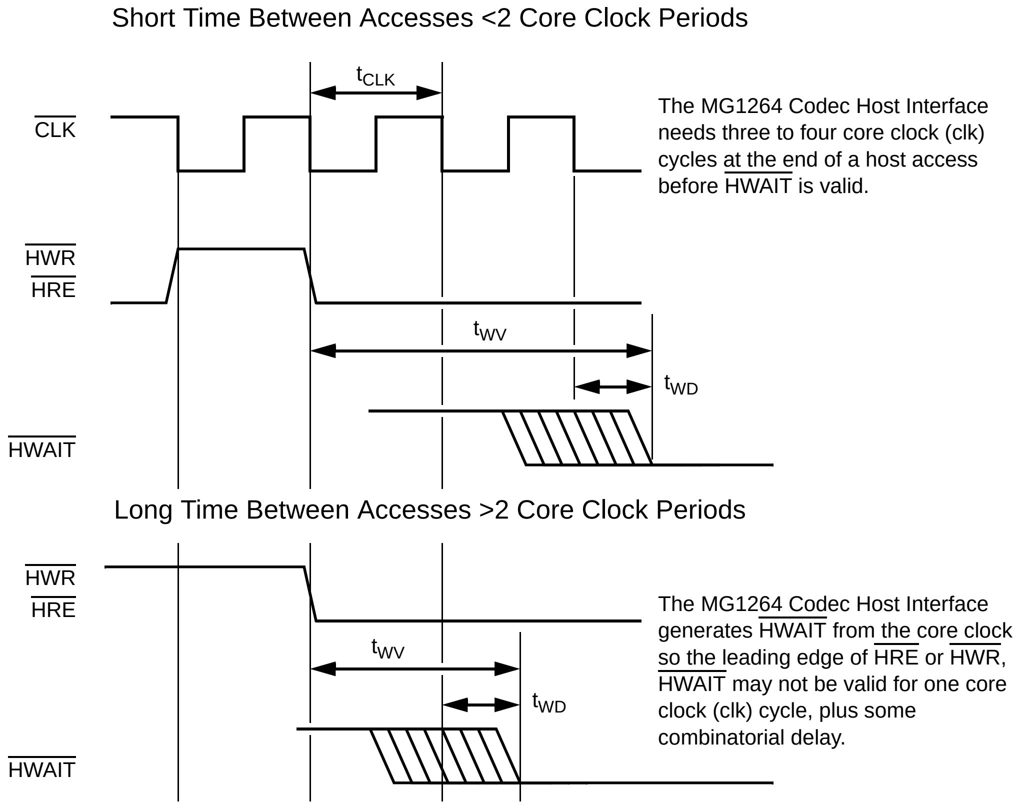

Short Time Between Accesses <2 Core Clock Periods
t_{CLK}
$\overline{CLK}$
The MG1264 Codec Host Interface needs three to four core clock (clk) cycles at the end of a host access before $\overline{HWAIT}$ is valid.
$\overline{HWR}$
$\overline{HRE}$
t_{WV}
t_{WD}
$\overline{HWAIT}$
Long Time Between Accesses >2 Core Clock Periods
$\overline{HWR}$
$\overline{HRE}$
The MG1264 Codec Host Interface generates $\overline{HWAIT}$ from the core clock so the leading edge of $\overline{HRE}$ or $\overline{HWR}$, $\overline{HWAIT}$ may not be valid for one core clock (clk) cycle, plus some combinatorial delay.
t_{WV}
t_{WD}
$\overline{HWAIT}$

Figure 11-3 $\overline{\textbf{HWAIT}}$ **Timing**

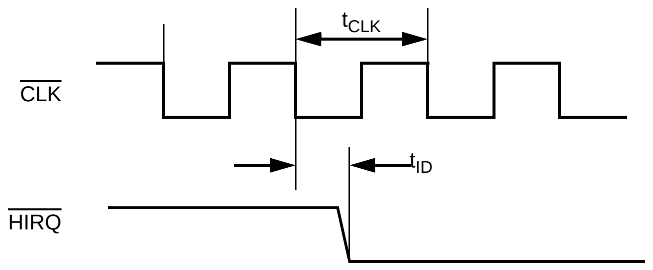


Figure 11-4 $\overline{\textbf{HIRQ}}$ **Timing**

Table 11-4 Host Interface Timing

Signal	Parameter	Description	Min	Max	Units
HADDR[6:1]	t_{WAS}	HADDR setup to trailing edge HWEn for write cycles	37	—	ns
	t_{WAH}	HADDR hold from trailing edge HWEn for write cycles	3	—	ns
	t_{RAS}	HADDR setup to leading edge HREn for read cycles	0	—	ns
	t_{RAH}	HADDR hold from trailing edge HREn for read cycles	0	—	ns
HDATA[15:0]	t_{WDC}	HDATA setup to trailing edge HWEn for write cycles	37	—	ns
	t_{WDH}	HDATA hold from trailing edge HWEn for write cycles	3	—	ns
	t_{RDD}	HDATA driven from leading edge HREn for read cycles	0	—	ns
	t_{RDV}	HDATA valid from leading edge HREn for read cycles	—	15	ns
	t_{RDH}	HDATA hold from trailing edge HREn for read cycles	2	15	ns
HWEn	t_{CWE}	HCSn Active to HWEn Active	0	—	ns
	t_{WEC}	HWEn Inactive to HCSn Inactive	3	—	ns
	t_{WEA}	HWEn active time	37	—	ns
HREn	t_{CRE}	HCSn Active to HREn Active	0	—	ns
	t_{REC}	HREn Inactive to HCSn Inactive	0	—	ns
	t_{REA}	HREn active time	$3*t_{CLK} + 8$	—	ns
HCSn	t_{CSH}	HCSn inactive time between accesses	$2*t_{CLK}$	—	ns
HDMARQn	t_{RQD}	HDMARQn valid from internal clock	—	8	ns
HIRQn	T_{ID}	HIRQn valid from internal clock	—	8	ns
HWAITn	t_{WD}	HWAITn valid from internal clock	—	8	ns
HWAITn	t_{WV}	HWAITn valid from HREn/ HWEn	—	12	ns

11.2.2 Video Interface AC Timing

Figure 11-5 and Table 11-5 show the AC timing parameters for the video interface.

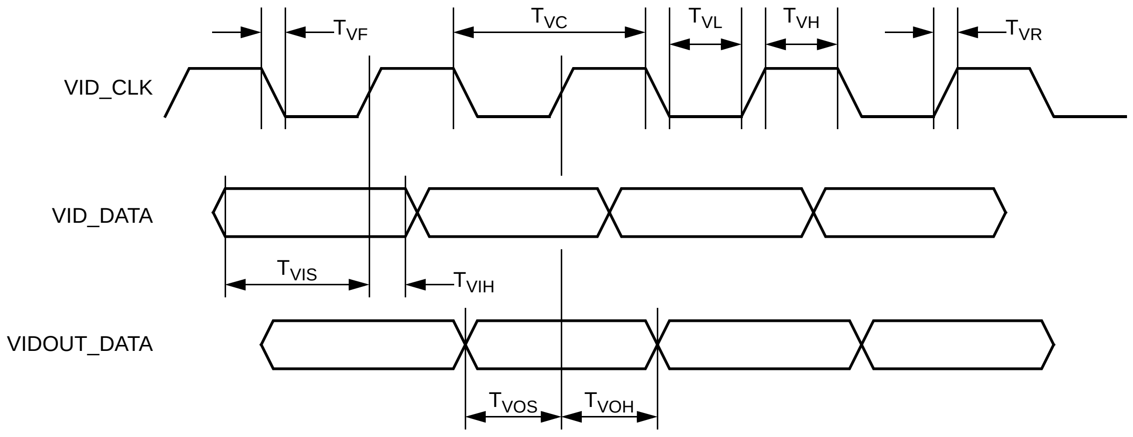


Figure 11-5 Video Interface Timing Diagram

Table 11-5 Video Interface AC Timing Values

Signal	Parameter	Description	Timing Value (ns.)		
			Min	Typ	Max
VID_CLK	T_{VC}	VID_CLK Cycle Time (27 MHz)	—	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	—	—

11.2.3 Audio Interface AC Timing

This section gives the AC timing parameters for the MG1264 Codec's audio interface. Figure 11-6 shows the relationships between the three audio clocks. Figure 11-7 shows the timing waveforms. Table 11-6 lists the AC timing for Audio Operations.

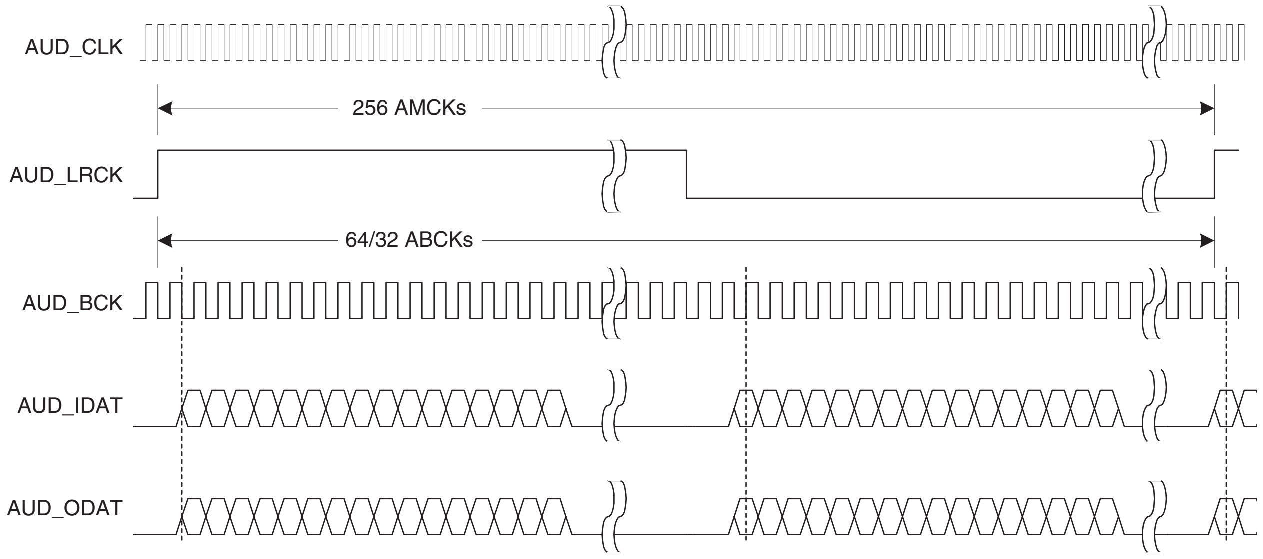


Figure 11-6 Audio Timing Diagram

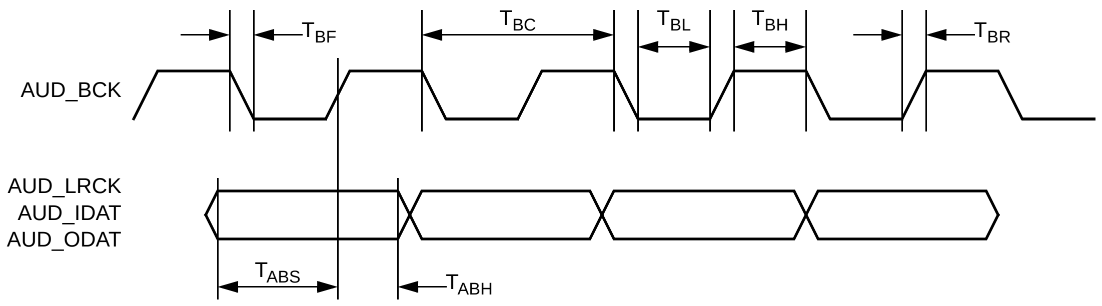


Figure 11-7 Audio Interface Timing Diagram

Table 11-6 Audio Interface AC Timing Values

Signal	Parameter	Description	Timing Value (ns.)		
			Min	Typ	Max
AUD_BCK	T_{BC}	AUD_BCK Cycle Time (48 kHz)	—	20833	—
	T_{BC}	AUD_BCK Cycle Time (32 kHz)	—	31250	—
	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 AIODATA AODAIA	T_{ABS}	Set-up Time to AUD_BCK	8	—	—
	T_{ABH}	Hold Time from AUD_BCK	3	—	—

11.2.4 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.

11.3 Packaging

Figure 11-8 shows the pinout for the MG1264 Codec. This figure is continued on the next page. Table 11-7 shows the pin list sorted alphabetically.

	1	2	3	4	5	6	7	8
A	HADDR1	VIDOUT_ DATA0	VIDOUT_ DATA2	VIDOUT_ DATA4	VIDOUT_ DATA6	VOFIELD	VOHSYNC	VID_DATA7
B	HADDR2	HCS	VIDOUT_ DATA1	VIDOUT_ DATA3	VIDOUT_ DATA5	VIDOUT_ DATA7	VOVSYNC	VID_CLK
C	HADDR4	HADDR3	—	—	—	—	—	—
D	HADDR6	HADDR5	—	—	CVDD	—	—	—
E	IVDDW33	$\overline{\text{HWE}}$	—	CVDD	—	—	—	—
F	$\overline{\text{HINT}}$	$\overline{\text{HRE}}$	—	—	—	—	OVDDW33	OVDDW33
G	$\overline{\text{HDMARQ}}$	$\overline{\text{HWAIT}}$	—	—	—	OVDDW33	GND	GND
H	HDATA1	HDATA0	—	—	—	OVDDW33	GND	GND
J	HDATA2	HDATA3	—	—	—	GND	GND	GND
K	HDATA4	HDATA5	—	—	—	GND	GND	GND
L	HDATA6	HDATA7	—	—	—	GND	OVDDW25	OVDDW25
M	HDATA8	HDATA9	—	CVDD	—	—	—	—
N	HDATA10	HDATA11	—	—	CVDD	CVDD	—	—
P	HDATA12	HDATA13	—	—	—	—	—	—
R	HDATA14	$\overline{\text{RESET}}$	TMS	TDI	TDO	TMODE	AUD_CLK	AUD_LRCK
T	HDATA15	SIN	SOUT	TCK	$\overline{\text{TRST}}$	AUD_IDAT	AUD_ODAT	AUD_BCK

Figure 11-8   MG1264 Codec Pinout Diagram

9	10	11	12	13	14	15	16	
VID_DATA6	VID_DATA4	VID_DATA2	VID_DATA0	VID_VSYNC	XIN	PLL_AVDD	PLL_AVSS	**A**
VID_DATA5	VID_DATA3	VID_DATA1	VID_FIELD	VID_HSYNC	IVDDW33	SD_CLK	SD_DQ0	**B**
—	—	—	—	—	—	SD_DQ1	SD_DQ15	**C**
—	—	—	CVDD	—	—	IVDDW25	SD_DQ13	**D**
—	—	—	—	CVDD	—	SD_DQ2	SD_DQ14	**E**
GND	GND	GND	—	—	—	SD_DQ4	SD_DQ3	**F**
GND	GND	OVDDW25	—	—	—	SD_DQ12	SD_DQ11	**G**
GND	GND	OVDDW25	—	—	—	SD_DQ6	SD_DQ10	**H**
GND	GND	OVDDW25	—	—	—	SD_DQ9	SD_DQ5	**J**
GND	GND	OVDDW25	—	—	—	SD_DQ8	SD_CKE	**K**
OVDDW25	OVDDW25	OVDDW25	—	—	—	SD_DQM1	SD_DQM0	**L**
—	—	—	—	CVDD	—	SD_ADDR11	SD_ADDR12	**M**
—	—	—	CVDD	—	—	IVDDW25	SD_ADDR9	**N**
—	—	—	—	—	—	SD_ADDR8	SD_ADDR7	**P**
SD_ADDR2	SD_ADDR1	$\overline{\text{SD\_CS}}$	SD_BA0	IVDDW25	$\overline{\text{SD\_CAS}}$	SD_ADDR6	SD_ADDR5	**R**
SD_ADDR10	SD_ADDR3	SD_ADDR0	SD_BA1	$\overline{\text{SD\_WE}}$	$\overline{\text{SD\_RAS}}$	SD_ADDR4	SD_DQ7	**T**

Figure 11-8 MG1264 Codec Pinout Diagram (Continued)

Table 11-7 MG1264 Codec Pin List Sorted Alphabetically

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
T8	AUD_BCK	E2	$\overline{\text{HWE}}$	GND	OVSS	NC	SD_DQ30
R7	AUD_CLK	G2	$\overline{\text{HWAIT}}$	GND	OVSS	NC	SD_DQ31
T6	AUD_IDAT	D15	IVDDW25	GND	OVSS	F15	SD_DQ4
R8	AUD_LRCK	N15	IVDDW25	GND	OVSS	J16	SD_DQ5
T7	AUD_ODAT	R13	IVDDW25	GND	OVSS	H15	SD_DQ6
D12	CVDD	B14	IVDDW33	GND	OVSS	T16	SD_DQ7
D5	CVDD	E1	IVDDW33	R2	$\overline{\text{RESET}}$	K15	SD_DQ8
E13	CVDD	GND	IVSS	T11	SD_ADDR0	J15	SD_DQ9
E4	CVDD	GND	IVSS	R10	SD_ADDR1	L16	SD_DQM0
M13	CVDD	GND	IVSS	T9	SD_ADDR10	L15	SD_DQM1
M4	CVDD	GND	IVSS	M15	SD_ADDR11	NC	SD_DQM2
N12	CVDD	VDD2	OVDDW25	M16	SD_ADDR12	NC	SD_DQM3
N5	CVDD	VDD2	OVDDW25	R9	SD_ADDR2	T14	$\overline{\text{SD\_RAS}}$
N6	CVDD	VDD2	OVDDW25	T10	SD_ADDR3	T13	$\overline{\text{SD\_WE}}$
GND	CVSS	VDD2	OVDDW25	T15	SD_ADDR4	T2	SIN
GND	CVSS	VDD2	OVDDW25	R16	SD_ADDR5	T3	SOUT
GND	CVSS	VDD2	OVDDW25	R15	SD_ADDR6	T4	TCK
GND	CVSS	VDD2	OVDDW25	P16	SD_ADDR7	R4	TDI
GND	CVSS	VDD2	OVDDW25	P15	SD_ADDR8	R5	TDO
GND	CVSS	VDD2	OVDDW25	N16	SD_ADDR9	R6	TMODE
GND	CVSS	VDD2	OVDDW25	R12	SD_BA0	R3	TMS
GND	CVSS	VDD2	OVDDW25	T12	SD_BA1	T5	$\overline{\text{TRST}}$
GND	CVSS	VDD2	OVDDW25	R14	$\overline{\text{SD\_CAS}}$	A15	PLL_AVDD
A16	PLL_AVSS	VDD2	OVDDW25	K16	SD_CKE	B8	VID_CLK
A1	HADDR1	VDD2	OVDDW25	B15	SD_CLK	A12	VID_DATA_0
B1	HADDR2	VDD1	OVDDW33	R11	$\overline{\text{SD\_CS}}$	B11	VID_DATA_1
C2	HADDR3	VDD1	OVDDW33	B16	SD_DQ0	A11	VID_DATA_2
C1	HADDR4	VDD1	OVDDW33	C15	SD_DQ1	B10	VID_DATA_3
D2	HADDR5	VDD1	OVDDW33	H16	SD_DQ10	A10	VID_DATA_4
D1	HADDR6	VDD1	OVDDW33	G16	SD_DQ11	B9	VID_DATA_5
B2	$\overline{\text{HCS}}$	VDD1	OVDDW33	G15	SD_DQ12	A9	VID_DATA_6
H2	HDATA0	VDD1	OVDDW33	D16	SD_DQ13	A8	VID_DATA_7
H1	HDATA1	VDD1	OVDDW33	E16	SD_DQ14	B12	VID_FIELD
N1	HDATA10	GND	OVSS	C16	SD_DQ15	B13	VID_HSYNC
N2	HDATA11	GND	OVSS	NC	SD_DQ16	A13	VID_VSYNC

Table 11-7 MG1264 Codec Pin List Sorted Alphabetically

Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
P1	HDATA12	GND	OVSS	NC	SD_DQ17	A2	VIDOUT_DATA_0
P2	HDATA13	GND	OVSS	NC	SD_DQ18	B3	VIDOUT_DATA_1
R1	HDATA14	GND	OVSS	NC	SD_DQ19	A3	VIDOUT_DATA_2
T1	HDATA15	GND	OVSS	E15	SD_DQ2	B4	VIDOUT_DATA_3
J1	HDATA2	GND	OVSS	NC	SD_DQ20	A4	VIDOUT_DATA_4
J2	HDATA3	GND	OVSS	NC	SD_DQ21	B5	VIDOUT_DATA_5
K1	HDATA4	GND	OVSS	NC	SD_DQ22	A5	VIDOUT_DATA_6
K2	HDATA5	GND	OVSS	NC	SD_DQ23	B6	VIDOUT_DATA_7
L1	HDATA6	GND	OVSS	NC	SD_DQ24	A6	VIDOUT_FIELD
L2	HDATA7	GND	OVSS	NC	SD_DQ25	A7	VIDOUT_HSYNC
M1	HDATA8	GND	OVSS	NC	SD_DQ26	B7	VIDOUT_VSYNC
M2	HDATA9	GND	OVSS	NC	SD_DQ27	A14	XIN
G1	$\overline{\text{HDMARQ}}$	GND	OVSS	NC	SD_DQ28	—	—
F1	$\overline{\text{HIRQ}}$	GND	OVSS	NC	SD_DQ29	—	—
F2	$\overline{\text{HRE}}$	GND	OVSS	F16	SD_DQ3	—	—

Table 11-8 MG1264 Codec Pin List by Side

Left side		Bottom side		Right side		Top side	
Pin	**Signal**	**Pin**	**Signal**	**Pin**	**Signal**	**Pin**	**Signal**
GND	OVSS	R4	TDI	GND	IVSS	NC	
B2	$\overline{\text{HCS}}$	R5	TDO	N15	IVDDW25	NC	
A1	HADDR1	T5	$\overline{\text{TRST}}$	N16	SD_ADDR9	NC	
GND	CVSS	GND	CVSS	M15	SD_ADDR11	VDD2	OVDDW25
NC		N5	CVDD	NC	SD_DQM3	NC	
NC		NC		NC	SD_DQM2	NC	
NC		NC		NC	SD_DQ31	GND	OVSS
E4	CVDD	NC		GND	OVSS	NC	
B1	HADDR2	R6	TMODE	VDD2	OVDDW25	NC	
C2	HADDR3	T6	AUD_IDAT	M16	SD_ADDR12	NC	
C1	HADDR4	R7	AUD_CLK	M13	CVDD	VDD1	OVDDW33
D2	HADDR5	N6	CVDD	NC	SD_DQ30	D12	CVDD
NC		T7	AUD_ODAT	L15	SD_DQM1	NC	
NC		R8	AUD_LRCK	NC	SD_DQ29	A16	PLL_AVSS
NC		NC		GND	CVSS	NC	
D1	HADDR6	NC		NC	SD_DQ28	A15	PLL_AVDD
E2	$\overline{\text{HWE}}$	T8	AUD_BCK	GND	OVSS	GND	CVSS
GND	IVSS	NC		VDD2	OVDDW25	GND	IVSS
E1	IVDDW33	T9	SD_ADDR10	L16	SD_DQM0	B14	IVDDW33
F2	$\overline{\text{HRE}}$	R9	SD_ADDR2	NC	SD_DQ27	A14	XIN
NC		T10	SD_ADDR3	K15	SD_DQ8	B13	VID_HSYNC
NC		NC		GND	OVSS	NC	
NC		NC		VDD2	OVDDW25	NC	
F1	$\overline{\text{HIRQ}}$	NC		GND	OVSS	NC	
G2	$\overline{\text{HWAIT}}$	VDD2	OVDDW25	GND	OVSS	A13	VID_VSYNC
G1	$\overline{\text{HDMARQ}}$	R10	SD_ADDR1	K16	SD_CKE	B12	VID_FIELD
NC		T11	SD_ADDR0	NC	SD_DQ26	A12	VID_DATA_0
NC		GND	OVSS	J15	SD_DQ9	B11	VID_DATA_1
NC		NC		VDD2	OVDDW25	A11	VID_DATA_2
VDD1	OVDDW33	NC		GND	CVSS	NC	
H2	HDATA0	NC		J16	SD_DQ5	NC	
H1	HDATA1	R11	$\overline{\text{SD\_CS}}$	NC	SD_DQ25	NC	
GND	OVSS	T12	SD_BA1	NC	SD_DQ24	VDD1	OVDDW33
J1	HDATA2	VDD2	OVDDW25	H16	SD_DQ10	B10	VID_DATA_3
NC		GND	OVSS	VDD2	OVDDW25	A10	VID_DATA_4
J2	HDATA3	NC		H15	SD_DQ6	NC	

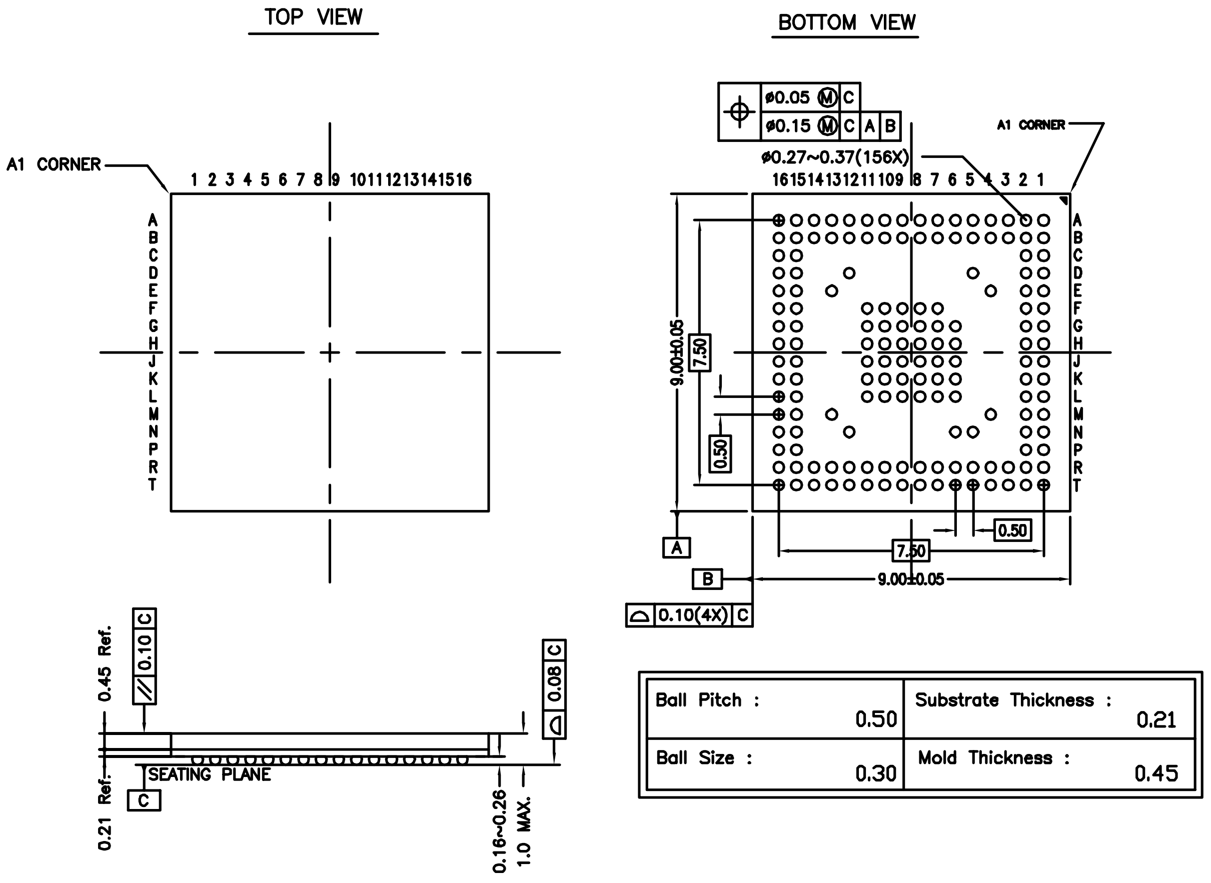
Table 11-8 MG1264 Codec Pin List by Side

Left side		Bottom side		Right side		Top side	
Pin	Signal	Pin	Signal	Pin	Signal	Pin	Signal
VDD1	OVDDW33	R12	SD_BA0	G16	SD_DQ11	GND	OVSS
GND	OVSS	T13	$\overline{\text{SD\_WE}}$	NC	SD_DQ23	B9	VID_DATA_5
K1	HDATA4	GND	IVSS	NC	SD_DQ22	NC	
K2	HDATA5	R13	IVDDW25	G15	SD_DQ12	A9	VID_DATA_6
NC		NC		GND	OVSS	VDD1	OVDDW33
NC		NC		F16	SD_DQ3	GND	OVSS
NC		NC		NC	SD_DQ21	A8	VID_DATA_7
L1	HDATA6	T14	$\overline{\text{SD\_RAS}}$	VDD2	OVDDW25	NC	
L2	HDATA7	R14	$\overline{\text{SD\_CAS}}$	F15	SD_DQ4	NC	
GND	OVSS	GND	OVSS	NC	SD_DQ20	NC	
VDD1	OVDDW33	VDD2	OVDDW25	NC	SD_DQ19	B8	VID_CLK
M1	HDATA8	NC		GND	OVSS	A7	VIDOUT_HSYNC
NC		NC		E13	CVDD	B7	VIDOUT_VSYNC
NC		NC		NC	SD_DQ18	A6	VIDOUT_FIELD
NC		GND	CVSS	NC	SD_DQ17	NC	
M2	HDATA9	N12	CVDD	E16	SD_DQ14	NC	
N1	HDATA10	T15	SD_ADDR4	E15	SD_DQ2	NC	
N2	HDATA11	T16	SD_DQ7	VDD2	OVDDW25	B6	VIDOUT_DATA_7
VDD1	OVDDW33	NC		D16	SD_DQ13	GND	OVSS
GND	OVSS	NC		GND	OVSS	A5	VIDOUT_DATA_6
NC		NC		GND	CVSS	B5	VIDOUT_DATA_5
NC		VDD2	OVDDW25	D15	IVDDW25	NC	
NC		GND	OVSS	NC	SD_DQ16	NC	
P1	HDATA12	R15	SD_ADDR6	NC		NC	
P2	HDATA13	R16	SD_ADDR5	NC		GND	CVSS
R1	HDATA14	NC		C15	SD_DQ1	D5	CVDD
T1	HDATA15	NC		VDD2	OVDDW25	VDD1	OVDDW33
R2	$\overline{\text{RESET}}$	NC		C16	SD_DQ15	A4	VIDOUT_DATA_4
M4	CVDD	GND	OVSS	NC		NC	
NC		P16	SD_ADDR7	GND	OVSS	NC	
NC		P15	SD_ADDR8	B16	SD_DQ0	B4	VIDOUT_DATA_3
T2	SIN	GND	CVSS	NC		A3	VIDOUT_DATA_2
T3	SOUT	NC		NC		B3	VIDOUT_DATA_1
R3	TMS	NC		NC		A2	VIDOUT_DATA_0
T4	TCK	VDD2	OVDDW25	B15	SD_CLK	NC	

11.4 Package Dimensions

Figure 11-9 shows the 156-pin BGA package dimensions.

Note: These dimensions are preliminary and subject to change. Contact Mobilygen Technical Marketing for up-to-date information.



Ball Pitch : 0.50	Substrate Thickness : 0.21
Ball Size : 0.30	Mold Thickness : 0.45

Figure 11-9 156-pin BGA Package Mechanical Dimensions