C-Cube Processor Applications Programming Interface (API)

Reference Guide

DV^XPRESS/DV^XPERT Technical Document Library

92-5API-101

This document is preliminary. We expect to update this document as we receive feedback from users of the product. Contact C-Cube Microsystems for a full design review before beginning design work.

C-Cube Microsystems Inc. reserves the right to change any products described herein at any time and without notice. C-Cube Microsystems assumes no responsibility or liability arising from the use of the products described herein, except as expressly agreed to in writing by C-Cube Microsystems. The use and purchase of this product does not convey a license under any patent rights, copyrights, trademark rights, or any other intellectual property rights of C-Cube Microsystems.

Trademark Acknowledgment:
C-Cube and the corporate logo are registered trademarks of C-Cube Microsystems. All other trademarks are the property of their respective companies.

Customer Support and Feedback:
To receive product literature or technical support, contact us at

C-Cube Microsystems
1778 McCarthy Boulevard
Milpitas, CA 95035
Telephone: (408) 944-6300
Fax: (408) 944-6314
E-mail: support@c-cube.com
http://www.c-cube.com

C-Cube Part # 92-5API-101
PRELIMINARY
November 17, 1997

Preface

This document is the primary source of technical information about the Processor's Applications Programming Interface (API).

Audience

This manual is intended for:

- System designers and managers who are evaluating the Processor's API.
- Programmers and software engineers who are writing application programs that interact with the Processor.

Organization

This manual is divided into seven chapters. See the table of contents for more details.

Conventions

Please note the following notation examples and conventions that are used in this manual:

Examples/ Conventions	Explanation
0x1C3	"0x" prefix indicates a hexadecimal number.
11011_2	"2" subscript indicates a binary number.
VIE, Res	Italicized acronyms or abbreviations (initial or all caps) indicate bit field names within registers or data words.
RESERVED or *Res*	Indicates bit fields within registers that are not defined. These bits are used by the internal microcode, and modifying these bits should be done cautiously. If the register is modified during operation, the host should perform a Read-Modify-Write operation to preserve the state of the reserved bits. Writing the incorrect value to a RESERVED register bit causes indeterminate behavior. In addition, all registers and configuration parameters in DRAM that are not explicitly given names are also RESERVED, and accessing these registers may cause indeterminate results on current or future implementations.
Byte ordering	Unless otherwise stated, all byte ordering in the <Product_Name> is big-endian: byte 0 is always the most significant (leftmost) byte.
Top, bottom field	This manual uses the MPEG standard terms "top" and "bottom" to refer to the video fields. The top field can also be referred to as the first, or the even field. The bottom field can also be referred to as the second, or the odd field.
MSB, LSB	Most significant byte, least significant byte.

Contents

Tables

1
Overview

The C-Cube Processor[1] is an integrated circuit that encodes and decodes MPEG-2 data and other digital video formats. It is optimized for use in computers with an internal Peripheral Component Interconnect (PCI) bus. It has a scalable platform architecture that efficiently implements motion-compensated block/DCT-based video compression algorithms in real time.

The processor is a compelling platform for MPEG encoders, decoders and codecs. It can be used as a 4:2:2 profile encoder and decoder, ML@MP encoder, SIF encoder, and simple profile codec. Because of its versatility, it is likely that the system designer might want to use the processor in several different products. To enable this, C-Cube provides a common Application Programming Interface (API) across the several processor-based products. The common API not only reduces support

1. The term “processor” or “the processor” is used from now on.

requirements and time-to-market, but also allows code re-use and easier maintenance.

For any processor-based product, the specific product API contains the proper mix of commands derived from the common API and whatever additional commands might be required by the specific microcode product.

This document describes the basic protocol for communicating control and status information between the host and all processor microcode products. The protocol elements common to all processor products include chip initialization, software downloading, command and message formats, management of command and message buffers, bitstream I/O, picture I/O, and other relevant information.

Although not all features described here are supported in all products, supported features must adhere to the API protocol. Refer to the product-specific documentation to determine the features that any particular microcode product supports.

1.1 Features

The processor Application Programmer Interface (API) provides these features:

- All processor-based products support a standard microcode format and use the same start-up procedure.
- The processor controls all PCI traffic except for the initial shared memory accesses, microcode downloading, and PTS counter (processor internal system time clock) updates.
- The processor accesses the host using ping-ponged “command” blocks.
- The processor sends status to the host via ping-ponged “message” blocks.
- “Sequence numbers,” attached to each command and message block, maintain communication synchronization.
- Regardless of the number of chips in the product, the host always communicates with the *master* processor.
- After command parsing is complete, error checking is always done.
- All commands have a similar format:

  <timecode><type><tc ID><stream ID><size><data payload>

  The amount and format of the data payload varies appropriately for the given command.

1.2 Microcode Download and Start-up

This section describes the elements involved in downloading, initializing, and executing the microcode on the processor.These include microcode file format, shared memory initialization, state retrieval for context switches, and processor status values. For those customers using two processors in a master-slave configuration, we also describe the two-chip microcode loading protocol.

1.2.1 Microcode File Format

The processor's microcode consists of two files. The first file uses the Intel Code 99 "hex" format. This file contains the default data pattern that is programmed into the serial EPROM connected to the processor's PCI interface when a serial EPROM is used.

The second file is the actual microcode and is formatted using our proprietary "microcode executable" (.ux) file format (Tables 1-1 through 1-3).

The .ux file contains the instructions that initialize the processor's memory for getting the microcode loaded and running and for enabling the host to receive commands (see note below). The host needs only to understand the simple syntax elements contained in the .ux file and to handle a few types of data transfers under host control. This feature eliminates the need for complex host software to implement reset routines. Also, it eliminates the need to maintain different loaders for different processor-based products.

Note:

There are some minor host initializations, which are not included in the .ux file and need to be addressed explicitly by the host, because they involve pointers to host buffers. These are covered in other parts of this document.

The .ux format file has three primary regions:

1. At the beginning of the file, a very simple header with a magic number and the length of the released portion of the file. This header contains the minimum information necessary to load the microcode. See Table 1-1.

Table 1-1 C-Cube Proprietary .ux File Format --Initial Header

Initial Header	
File Element	**Definition**
■ Magic number (32 bits)	This is the 4-byte sequence 0x43, 0x33, 0x55, 0x58, which is equivalent to the ASCII string "C3UX" (without null termination).
■ Number of sections (32 bits)	This is the total number of "Sections" (defined on the next page) contained in the file. This is a little-endian number (the first byte in the file is the least significant).
■ Length of initial portion (32 bits)	The length of the initial portion is the length of that part of the file which is always distributed to customers. This is specified as a number of bytes and is a little-endian number. Optional information may be present at the end of the file beyond this length, but it is not needed for normal operation.

2. The *main region* of the file follows and it carries one or more code *sections*. Each section specifies a certain portion of memory initialization, either of SDRAM or of one of the CBUS registers. Together, the sections in this main region supply the steps for microcode start-up.

   Each section contains part of the microcode that is to be loaded into a contiguous section of the processor's memories. All sections must be contiguous within the file and occur between the initial header and the extended header. Data from each section must be loaded to the processor in the same order they occur in the file. See Table 1-2.

Table 1-2 C-Cube Proprietary .ux File Format --Section

Section	
File Element	**Definition**
Data type (8 bits)	This is the type of memory space in which this Section loads. The predetermined values are: 0 SDRAM 1 CBUS
Flags (8 bits)	■ Bit 0: Ignore Checksum. This bit is set by the microcode linker for only those Sections intended for customer modification. It should not be modified by customers. If *zero*, the checksum (see below) for this Section is valid. If *one*, it is invalid and should be ignored by the microcode loader. ■ Bit 1 and 2: Section Format. If *zero*, the Section data is a block of data loaded at the Section start address. If *one,* the Section data consists of address and data pairs, where the address is a 32-bit big-endian quantity, and the data format varies based on the data type. The loader determines how many bits of each quantity are valid according to the data type. If the data type is 1 (i.e. CBUS), the data is one big-endian 32-bit word. If the data type is 0 (i.e SDRAM), the data is one byte. Processor PCI target mode allows only data transfers in 32-bit-word multiples. Thus, the host has to read 32 bits, mask out 3 bytes and write 32 bits back to SDRAM. If *two*, the section data will have a valid big-endian address in the section header, and must contain exactly one byte. When this of type section is encountered, the loader will loop indefinitely, checking the current value of the memory byte at the address while it is not equal to the value of the section data byte. If *three*, the section is a big endian 32-bit value specifying a minimum number of microseconds for the loader to wait before proceeding. The section start field is unused. ■ Bits 3-7: unused (will be 0).

Section (Continued)	
File Element	**Definition**
Unused flags (16 bits)	■ Will be 0.
Section length (32 bits)	This is the length of this Section in bytes.
Section start (32 bits)	This is the product-specific little-endian address within the selected memory space for the first data word in the segment. Other data words must be loaded sequentially at increasing addresses. If Flags[1:2] is 1 or 3, then this value is unused and will be 0 (see above).
Section checksum (32 bits)	This is a simple arithmetic little-endian checksum of the Section. It is ignored by the loader if Flags[0] is 1 (see above). It is computed as the sum of each *byte* in the Section, including all fields of the header except the checksum itself.
Section Data	This variable-length field contains the data values actually written to the selected memory space (Data type). This field may contain 32-bit address values if Flags[1:2] is 1.

3. After the main region, the *extended header* follows. It contains the copyright, microcode product and revision number information. It also provides the silicon revision information of the hardware upon which the microcode should run.

   This header is distributed to customers, but is not needed for production loads of microcode. All strings are ASCII characters and are padded with zeroes. See Table 1-3.

Table 1-3 C-Cube Proprietary .ux File Format --Extended Header

Extended Header	
File Element	**Definition**
■ File format revision (16 bits)	The first byte contains the minor revision code for the file format. Changes to the minor revision indicate changes only to the C-Cube private portions of a file (debugging data, below) or changes which donot affect the basic load mechanism. The second byte is the major revision for the format. The description in this table matches a major revision value of 1.
■ Microcode version number (32 bits)	ASCII string (no null termination) indicating the version number of the microcode contained in this file.
■ Microcode product name (256 bits)	Text identifying the microcode product contained in the file, such as "AFF."
■ Customer specific name (256 bits)	Text describing customer-specific variant, or all zeroes if none.
■ Silicon product name (128 bits)	Name string such as "DVX or CL5000."
■ Minimum silicon revision (8 bits) ■ Maximum silicon revision (8 bits)	This is the minimum silicon mask revision needed to run this microcode, and the maximum revision (if any) past which this microcode will not work. If there is no maximum revision field, the maximum revision is 0. Silicon revisions are encoded as four bits for the major revision (all layer change) and four bits for the minor revision (metal mask only). 0x10 represents an "A0" mask revision.
■ Minimum extended feature revision (8 bits) ■ Maximum extended feature revision (8 bits)	These are the minimum silicon revisions needed for any conditional or extended features supported by this microcode, and the maximum silicon revision needed to run all extended features supported by this microcode. The microcode will have to examine the revision itself to decide which features can be enabled.
■ Copyright string (512 bits)	This field contains the microcode's copyright notice.
■ Header checksum (32 bits)	This is a simple arithmetic checksum of the contents of the Initial header and the other fields of the Extended header.

Each code section in the main region supplies:

- Memory ID number[2]
- Starting address
- Length
- Checksum (optional)
- There are also special sections, indicated by a flag field, that contain multiple address/data pairs to allow the data to be written to non-contiguous areas of memory, or that call for the polling of a certain location for a certain value.

A typical .ux file provides the following code groups:

- First, one or more sections containing register-writes that reset the processor
- Next, one or more sections to load the microcode itself into SDRAM
- Finally, one or more sections containing the SDRAM and register-writes necessary to put the chip in the run state and start the execution of the microcode

In summary, to load the microcode, the host needs only to:

- Load all the data from each section following the order given by the file
- Perform some other very minor initializations as outlined in the succeeding parts of this document. The host can load SDRAM using either target-mode transactions or by manipulating the processor's control registers to cause it to perform bus-master copies between the host memory and the processor’s SDRAM.

2. Indicates to the host where to write data

1.2.2 Shared Memory Initialization

The region of the processor's SDRAM between addresses 0x40000 and 0x4003F is reserved as *shared memory*. This region is used with the .ux file to complete various microcode initialization steps[3] and it needs to be initialized for the processor to run. Thus, the best time to initialize shared memory is before loading the microcode.

Table 1-4 shows the contents of memory.

Table 1-4 Shared Memory Contents

Address	Data	Description
0x40000	Even Command Buffer Address	Section 1.3.4
0x40004	Odd Command Buffer Address	Section 1.3.4
0x40008	Even Message Buffer Address	Section 1.3.4
0x4000C	Odd Message Buffer Address	Section 1.3.4
0x40010	Microcode Status	Section 1.2.4
0x40014	Reserved	n/a
0x40018 – 0x40024	PTS Counter	"System Clock Reference" on page 79
0x40028	Loader Command	Section 1.2.5
0x4002C	Loader State	Section 1.2.5
0x40030	Microcode Start Mode Rag	Section 1.2.3
0x40034	Microcode State Address	Section 1.2.3
0x40038 - 0x4003F	Reserved	n/a

The .ux file implicitly handles the Loader Command location and the Loader State location, while the processor writes the Microcode Status location (Section 1.2.4). Remaining locations in the above table must be initialized by the host. (See Table 1-4 for document references.)

3. The PTS Counter locations, however, are used throughout code execution; see "System Clock Reference" on page 79.

1.2.3 Microcode Start Mode Flag

The Microcode Start Mode Flag (0x40030) and the Microcode State Address (0x40034) shared memory locations are reserved for encode/decode context switch applications. In those applications, the processor synchronizes with data loaded through the PCI bus instead of the real-time video clock. To reduce the context switch time, the processor skips some internal initialization stages. It also preserves the previous encoder or decoder state to maintain constant picture quality. To make this possible, the processor needs to save and restore its state in order to smoothly execute the context switch and restart correctly.

The Microcode Start Mode Flag location contains two bit-flags. The first one (bit 0) controls how the processor synchronizes itself. If this bit is zero, the processor starts the microcode in the normal way, synchronizing itself with video I/O timing and operating in real time. If bit 0 is set to one, on the other hand, the processor will synchronize itself with PCI picture I/O ("Picture Input/Output Interface" on page 56) and operate in non-real-time. For context switch applications, bit 0 should be set to one; otherwise, it should be set to zero.

Bit 1 of the Microcode Start Mode Flag controls the restoration of previous processor state. If it is set to zero, the processor fully initializes itself and the Microcode State Address location is ignored. However, if bit 1 is set to one, the processor obtains previous state information from a host buffer at the address given in the Microcode State Address Location. This host buffer will have been filled by use of a Context Switch command issued before the processor was last halted ("Context Switch" on page 71).

For non-context-switch applications, bit 1 of the Microcode Start Mode Flag should always be set to zero. For context-switch applications, it should be set to zero the first time the microcode is loaded for each of the two products involved in the context switch, and set to one each time the microcode is re-loaded thereafter.

1.2.4 Microcode Status Location

The Microcode Status Register, which resides in shared memory address location 0x40010, is a 32-bit word that assists the developer in the initial debugging and bring-up of the microcode and the host communication software.

Upon boot-up, the processor sets the Microcode Status Register to “Init” to reflect its starting operation. When the initialization stage is done, the status register is set to “Idle.” If the initialization fails, “Error” bits are set. See Table 1-5 for a description of the 32 bits.

.

Table 1-5 Microcode Status Register

Bits	Value	Name	Meaning
0:3	0	Uninit	Microcode hasn’t started yet or didn’t get to _main()
	1	Init	Processor is in the initialization process
	2	Idle	Processor completed initialization or was running and received a “stop” command
	3	Run	Processor is running and performing encode or decode
	4	Pause	Processor encode/decode operation is paused
	5-15	Reserved	Reserved
4:7	0	No Error	No Error
	1	Error 0	Checksum failure detected in init routine
	2	Error 1	Communication protocol failure detected
	3-15	Error2... Error14	Reserved
8:31	0-0xFFFFFF	Reserved	Reserved

1.2.5 Two-Chip Processor Loading Protocol

Figure 1-1 shows a dual processor implementation. In this two-chip system, the host accesses the secondary (slave) chip through the primary (master) chip. The host is not burdened with the two-chip processor loading protocol because it is handled by the .ux file syntax.

We describe the two-chip loading protocol here. This information might be useful if a customer finds it necessary to debug the start-up process.

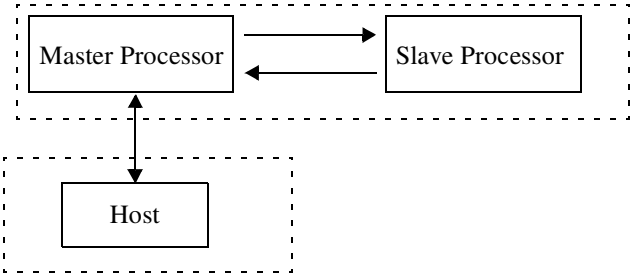

Figure 1-1 General Dual Processor System

In a two-chip system, the .ux file is set up as follows:

1. The host loads the slave's microcode onto the master chip.
2. The host sets the shared memory Loader Command word at address 0x40028 to "kLoad_Slave_uCode," which instructs the master chip to transfer the code to the slave chip (Table 1-6.)

Table 1-6 Loader Command Word

Bits	Value	Name	Meaning
0:3	1	kRun_uCode	Instructs the master processor to execute the loaded microcode.
	2	kLoadSlave_uCode	Instructs the master processor to transfer the microcode to the slave processor.
	0,3-7	Reserved	Reserved
4:31	0 - 0xFFFFFFF	Reserved	Reserved

3. The host sets the Loader State shared memory location at address 0x4002C to "kInit" (Table 1-7.)

Table 1-7 Loader State Words

Bits	Value	Name	Meaning
0:3	3	kInit	Initialization in progress.
	4	kSlave_Load_Done	The slave processor microcode is loaded.
	5	kMaster_Ready	The master processor is ready.
	0-2, 6-7	Reserved	Reserved
4:31	0 - 0xFFFFFFF	Reserved	Reserved

4. The host sets the processor to the run state and executes the microcode. The master processor transfers the code to the slave processor and sets the Loader State word to "kSlave_Load_Done." The host polls the Loader State word until it detects this value, then it loads the microcode for the master chip.
5. After the microcode for the master chip is loaded into the processor, the host initializes the Loader Command shared memory location to "kRun_uCode" and the Loader State shared memory location to "kInit", sets the master chip to the run state, and executes the microcode. The host then waits for the transition of the Loader State location to "kMaster_Ready"to indicate successful microcode loading.

   Note:
   When polling the Loader State shared memory location, limit the polling frequency to the rate specified by the Processor Hardware Specification. Also, include a host time-out counter to avoid dead locks.

6. Once the microcode loading and start-up processes are complete, normal command and message communication between the host and the processor occurs. The next section describes this process.

   Note:
   In a single chip implementation, the host must set the loader state location to "kMaster_Ready" *before executing the microcode.*

1.3 Communication Process

Once the microcode loading and start-up is complete (see previous section), normal command and message communication between the host and the processor begins. This section describes the protocol used by the host to control and monitor a processor-based product.

The processor command and messaging interface is tightly contained and well-defined across all processor-based products, leading to maintainability, portability, and ease of design for the host system designer. The basic protocol guidelines follow:

- During initialization, the host allocates two *command* buffers and two *message* buffers on the host side of the PCI bus for all communication with the processor. One pair of buffers is used for *writing commands to the processor* and one pair is used for *reading messages from the processor.* Each buffer pair has an *even* and an *odd* component.
- The hosts writes all *commands*[4] to control processor operation into one of the command buffers. Command transfers take place in a ping-ponged manner; that is, the host writes to one command buffer (e.g., odd), while the processor reads from the other command buffer (e.g., even). After each command cycle, they switch buffers to avoid race conditions.
- All processor-based products write *messages*[5] to the host in response to certain internal events such as error conditions or bitstream consumption. The processor writes these messages into the message buffers. Messaging is also done in a ping-ponged manner. For example, while the processor is writing to the even message buffer, the host is reading from the odd message buffer.
- Unless the microcode is configured otherwise, the exchange of command and message information between the host and the processor takes place once in each field time. This is synchronized either by an interrupt from the processor to the host (Section 1.4.1) or by the host polling the processor for completion of each cycle of command and message processing (Section 1.4.2).

4. See Section 1.3.2 for overall *command* structure.
5. See Section 1.3.3 for overall *message* structure.

- One example where communication does not occur once per field is in non-real-time processing. In this case, the host and processor typically communicate once for each field's worth of data that is processed. In any case, the processor never sends an interrupt to the host more often than once per field time.

To summarize the process, for each communication cycle (typically every field), the processor:

- Copies the contents of the current host command buffer into its internal command queue(s), then switches its internal command input pointer to the other host command buffer for use in the next cycle.
- Transfers the contents of its internal message queue(s) to the current host message buffer, then switches its internal message output pointer to the other host message buffer for use in the next cycle.
- Issues an interrupt to the host.

When the host receives the interrupt from the processor (or, if in polled mode, when it detects that the processor has completed the message buffer transfer), the host:

- Clears the processor interrupt pin by writing a value of 0x20 to the host control register at CBUS address 0xFC0000, unless it is in polled mode.
- Reads and parses any processor's messages from the message buffer. To avoid a race condition, the host alternates reading between odd and even message buffers with each cycle. Since the processor is also using a ping-ponged scheme, switching message buffers with each communication cycle, the host simply switches back and forth with each cycle maintaining synchronization.
- Writes commands to the processor into the command buffer that was just read by the processor. As with the message buffers, a race condition is avoided because the processor is, for example, reading the odd command buffer while the host is writing to the even command buffer.

The rest of this manual provides additional details about the host communication protocol. A diagram at the end of this section discusses timing elements of the host communication protocol.

1.3.1 Sequence Numbers

A *sequence number* provides synchronization between the host and the processor. This is a 16-bit number maintained by both the host and the processor and it is included within the message and command data. The number is incremented for each communication cycle and it rolls over from 0xFFFF to 0x0000.

During each communication cycle, the processor writes a sequence number to the current message block. The host checks this number against its internal sequence number to verify synchronization.

On the other side, the host writes a sequence number to the current command block. The processor checks it against its internal sequence number to verify synchronization.

To ensure cycle synchronization, the following conditions must be met:

- The processor must detect a sequence number in the current command buffer equal to the sequence number it is writing to the current message buffer.
- For the host, the sequence number it reads from the current message buffer should lag two digits behind the sequence number it is writing to the current command buffer.

If the above conditions are not met, then a synchronization error has occurred and error recovery is needed. Section 1.3.5 describes the error recovery process.

Because of the ping-ponged nature of the command and message buffers, the first command buffer and the first message buffer have *odd* sequence numbers, while the second command buffer and the second message buffer have *even* sequence numbers (numbers start at 1). For this reason, the two command buffers are called “odd and even command buffers,” while the two message buffers are called “odd and even message buffers.”

1.3.2 Command Block Format

The size of the odd and even command blocks depends on the requirements and constraints for each product and may be different among processor-based products. However, no product requires a command block size greater than 4K bytes. See the product-specific documentation for the exact size requirement for each product.

Regardless of the size of the command blocks, they must conform to the same basic format across all products. Table 1-8 shows this format.

Table 1-8 Command Block Format

Data Field	Length	Description
Priority Commands	M * 32 bits	Priority commands of the format described in "Command Syntax" on page 32.
Scheduled Commands	N * 32 bits	Scheduled commands of the format described in "Command Syntax" on page 32.
Padding	(Total size of command block) - ((M + N + 2) * 32 bits)	The command block is fixed length, so this field varies in size to keep the length and sequence number at the last locations in the block.
Reserved	16 bits	Must be set to 0.
Length M	16 bits	The size of the priority command portion of the command block, in 32-bit words.
Length N	16 bits	The size of the scheduled command portion of the command block, in 32-bit words.
Sequence Number	16 bits	Communication synchronization sequence number as described above in Section 1.3.1.

The highlights of the Command Block format are as follows:

- Priority commands appear first in the host command block, followed by scheduled commands (Commands are further explained in Chapter 2).
- Since there may be a varying number of commands (each of a variable size), length fields (M and N) are provided so that the processor can discriminate between the end of the priority command and the scheduled command portions of the command block.

- Since the length fields and the sequence number are at fixed places at the end of the command block, there will be a variable amount of unused space in the middle of the command block between the commands and the length fields.
- The host writes the current command block during each command cycle, even if it has no commands to send, then writes the sequence number as the last field. If there are no commands, M and N are set to zero, but the sequence number is updated to maintain communication synchronization with the processor. Also, in order to keep from repeatedly reading the same command block during error recovery, the processor sets the most significant bit of the N field after it has finished reading the block. This means that the host must clear this bit during each communication cycle.
- Since the API is generalized for all processor products, some fields of the command block may not apply to all products. For example, some products may only support priority commands, rendering useless the scheduled command region of the command block. Unused fields are set to zero by the host to ensure compatibility with future microcode revisions.

1.3.3 Message Block Format

Like the command blocks, the odd and even message blocks may have different size requirements for different products. However, they are never larger than 1K bytes. Table 1-9 shows the general message block format that is common to all processor-based products. The highlights of the Message Block format are as follows:

- The actual messages occupy the first part of the message block. Since there may be a variable number of messages of varying lengths in the block, a length field (N) is provided so that the host software can determine where the message portion of the message block ends.
- The length field and the sequence number occupy the last word of the message block. Thus, in between the actual messages and the length field there is an unused area of variable size depending on the amount of message data present.
- The processor always writes the current message block, even if there are no messages. If there are no messages, N is set to zero,

but the sequence number is still updated to maintain synchronization. The sequence number is always the last data written to the message block. As such, it can act as a semaphore to the host if the host wishes to perform communication in polled mode instead of receiving interrupts from the processor (Section 1.4.2).

Table 1-9 Message Block Format

Data Field	Length	Description
Messages	N * 32 bits	Messages of the format described in "Messages" on page 39
Padding	(Total size of message block) - ((N + 2) * 32 bits)	Message block is fixed length, so this field varies in size to keep the length and sequence number at the last locations in the block.
Reserved	32 bits	Undefined, should be ignored
Length N	16 bits	Total length of the actual messages in the block, in 32-bit words.
Sequence Number	16 bits	Communication synchronization sequence number as described above in Section 1.3.1.

1.3.4 Communication Initialization and Bootstrap

Before starting the processor, the host allocates its memory for even and odd command buffers and even and odd message buffers of the sizes required by the specific product. Because of limitations of the PCI bus, the addresses of the command and message buffers must be eight-byte aligned.

The host must communicate to the processor the command and message buffers addresses via shared memory at the time required for initializing shared memory, i.e. just before the microcode is loaded. Table 1-4 shows the locations set aside for this purpose and the rest of the shared memory locations.

Note:
The processor expects the shared memory addresses to contain legal values before microcode execution begins. If the host does not provide legal values as part of the microcode download process, host memory could be corrupted by the processor.

To bootstrap the communication process, the host prepares the first two command buffers before starting microcode execution. The host writes the odd command buffer with a sequence number of one, and the even command buffer with a sequence number of two. These command blocks may or may not have actual commands in them; if they do not, their M and N fields are set to zero (Section 1.3.2).

When the processor executes, it checks that the odd command buffer is ready with a sequence number of one. (This is the case if the host follows the proper communication bootstrap procedure.) It then reads this first command block and writes the first message block to the odd message buffer with a sequence number of one. First, the processor sends the Initialization Message ("Initialization Message" on page 41). This message serves as a handshake back to the host, signalling that the processor has started the communication process. It also informs the host of parameters such as its internal checksum result and the software revision number.

Note:
We do not specify whether or not the processor sends the first message within the same field-time as when it detects the host's first command. The processor can delay this action one or more fields. This should be transparent to the host and the host can wait until the processor signals that the first message is written.

Once the processor completes the first command transfer and message blocks, it sends its first host's interrupt. After receiving the interrupt (or, if in polled mode, after detecting the first message block's sequence number), the host reads the first message block and sends the third command block in the odd command buffer using a sequence number of three.

From this point, communication proceeds as normal. For example, just prior to the second host's interrupt, the processor reads the even command block with a sequence number of two (prepared by the host prior to microcode execution) and writes the even message block with a sequence number of two. On reception of the second interrupt, the host reads the second message block and writes the even command block with a sequence number of four.

The odd-even-odd-even pattern continues. Due to the nature of the bootstrap process, in which the host prepares the first two command blocks before executing the microcode, the message block sequence number always lags two digits behind the sequence number of the next host's command block.

Table 1-10 graphically exemplifies the entire host communication process, which includes the bootstrap routines.

1.3.5 Error Recovery

Due to performance constraints or system hardware glitches, errors may occur in the host communication process. These errors are detected either by the host or the processor as follows:

- The processor records an error if the sequence number it reads from the current command block does not match the sequence number it writes to the current message block.
- The host registers an error if the sequence number it reads from the current message buffer does not lag two digits behind the sequence number it is writing to the current command buffer.

The above cases indicate that either the host or the processor missed a communication cycle.

If the processor detects an error, it writes a sequence number of 0x7FFF to the even message block[6]. It then waits until the host writes a sequence number of one in the odd command block and clears the most significant bit of the "N" field (Section 1.3.2). In the meantime, it reduces the host interrupt rate by a factor of four to prevent amplifying any interrupt-servicing problem that caused the error in the first place.

6. This is an odd number in the even blockand it never occurs normally.

When the host sees a sequence number of 0x7FFF in the even message block, indicating that the processor detected a communication error, it restarts the communication process. That is, it writes a sequence number of two to the even command buffer (sets the M and N fields to zero), and writes a sequence number of one to the odd command buffer. Since the processor is waiting for the sequence number in the odd buffer to become one before it restarts the communication process, the host writes the even command block before the odd command block to prevent the processor from restarting the communication process before the even command block is ready.

Once the processor detects a sequence number of one in the odd command block, it assumes that the host is ready to resume normal communications. Consequently, it restarts the communication process on its side and resumes the normal rate of interrupts. Now, communications proceeds normally.

> ***Note:***
> *As with the communication bootstrap process, in error recovery, the processor does not necessarily respond immediately to the host writing the first command block. However, this should be transparent to the host since it waits until the processor writes the first message.*

If the host detects a sequence number mismatch, it forces an error by writing the next command block's sequence number with an *out-of-range* value. The processor detects this error and starts the protocol resynchronization process.

While in error recovery mode, the processor continues the encoding or decoding process using current parameters and available bitstreams or buffers.[7] Thus, if the host decides not to keep up with the interrupts from the processor, it can operate indefinitely in error recovery mode while the processor continues its operations.

7. Refer to "Data Stream Input and Output" on page 45 for the default behavior of the data stream I/O process.

1.3.6 Sample Communication Sequence

Table 1-10 shows a typical processor-host communication sequence including the bootstrap process and an error detection and correction scenario. In the table, the first row represents the initialization period while other rows represent half of a VSYNC cycle (i.e. VSYNC high or VSYNC low),

Processor actions preceding the host interrupt occupy the upper part of each cell; host actions occurring after the host interrupt occupy the lower part. The cells displaying an action by the processor or the host indicate the command or message buffer accessed and the sequence number read from or written to that buffer.

Table 1-10 Sample Communication Sequence

VSYNC	Interrupt	Command Read By processor	Message Written By processor	Message Read By Host	Command Written By Host	Comment
Init	Ucode not started	Ucode not yet started	Ucode not yet started	Ucode not yet started	Odd: 1 and Even: 2	Initialization by host before starting the microcode (Ucode).
0	1	Odd: 1	Odd:1	Odd:1	Odd: 3	Processor detects that host is ready and sends the first host interrupt after writing the first message buffer.
1	2	Even: 2	Even: 2	Even: 2	Even: 4	Normal communication cycle.
2	3	Odd: 3	Odd: 3	Odd: 3	Transfer error	Host attempts to write 5, but the data doesn't get there for some reason.
3	4	Even: 4	Even: 4	Even: 4	Even: 6	Error not yet detected, so this cycle is normal.
4	5	Odd: 3 (Error: same as last odd command)	Even: 0x7FFF (would have been Odd: 5)	Even: 0x7FFF	Even: 2, then Odd: 1	Command 5 never made it. The processor detects this and sets the even message to 0x7FFF. The host then detects this and starts recovery.
5	6	Odd: 1	Odd:1	Odd:1	Odd: 3	Back to odd buffers and odd VSYNC, so restart communication.
6	7	Even: 2	Even: 2	Even: 2	Even: 4	Back to normal.
7	8	Odd: 3	Odd: 3	Odd:3	Odd: 4	. . . ad infinitum.

The above table does not reflect the additional delay that, in the bootstrap and error recovery processes, may or may not occur between the time that the processor detects that the host wrote the first command buffer and the time that it writes the first message buffer and sends a host interrupt. Any such delay should be transparent to the host and the host should not depend on any specific response time.

1.4 Application Models

This section briefly describes two possible models for host applications that need to communicate with processor-based products. The first is an interrupt-driven model, while the second is a polled-mode model.

1.4.1 Interrupt-Driven Model

In an application using interrupt-driven communication, the host attaches an interrupt service routine (ISR) to the interrupt generated from the processor. The interrupt indicates completion of the command and message data transfer. The ISR handles the processing of the messages from the host and the commands to the processor as follows:

- The ISR stores messages in an internal message queue for further processing.
- The ISR transfers pending commands from an internal command queue instead of immediately parsing messages and deciding which commands should be sent next. This minimizes the time spent inside the ISR, which is usually a design concern for most systems.

When the above method is used, the main processing loop of the host application parses the messages from its internal messages queue and puts whatever commands it decided were appropriate into its internal command queue for transfer to the processor at the next host interrupt. Figure 1-2 illustrates this type of application.

In addition to reading messages and writing commands, the host's ISR verifies the sequence number of the current message block against the sequence number it writes to the next command block. The sequence number of the message block that the host reads with each communication cycle always lags two digits behind the sequence number of the command block that the host writes (Section 1.3.4). If this is not the case, the ISR needs to enter error recovery mode (Section 1.3.5).

The host ISR also needs to clear the host interrupt. Otherwise, since the host interrupt is a level-triggered interrupt, the host will continue to be interrupted. The interrupt can be cleared by writing a value of 0x20 to the host control register at address 0xFC0000.

In a normal situation (i.e., real-time decoding or encoding), the processor ensures that the host interrupts are one field time apart. However, the phase difference between a VSYNC transition and the host interrupt may be different in each microcode product. Refer to "System Timing" on page 75 and the product-specific documentation for more exact timing information.

In a non-realtime scenario such as copying images from the host to the encoder via the PCI bus or sending images through the PCI bus back to the host, a host interrupt is issued whenever the processor completes an image (field or frame) of data. In any case, the processor ensures that there is never more than one interrupt per field time.

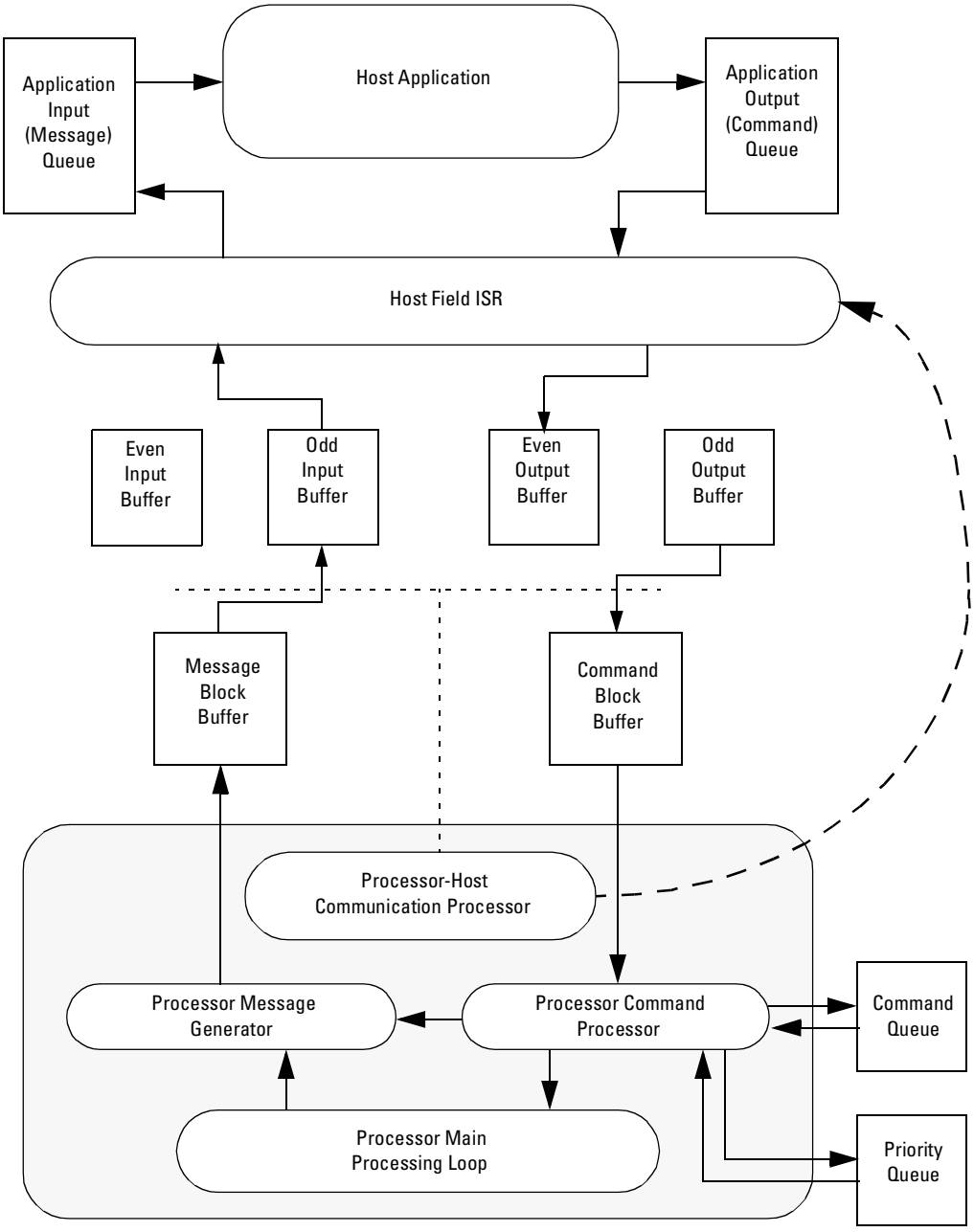

Figure 1-2 Typical Interrupt-Driven Host Application

1.4.2 Polled-Mode Model

The host can also communicate with the processor in polled mode. Since the processor writes the current message block after it reads the current command block, and the sequence number is the last word written to the message block, the message block sequence number serves as a semaphore for the host to detect when the processor has completed transfer of all the command and message data. The host can simply ignore the interrupt from the processor and wait for the sequence number in the current message block to change. At this point, it can proceed with message and command processing as it would with an interrupt-driven application. Figure 1-3 illustrates a typical polled-mode host application.

Note that to the processor, it is irrelevant whether the host runs in polled mode or interrupt driven mode.

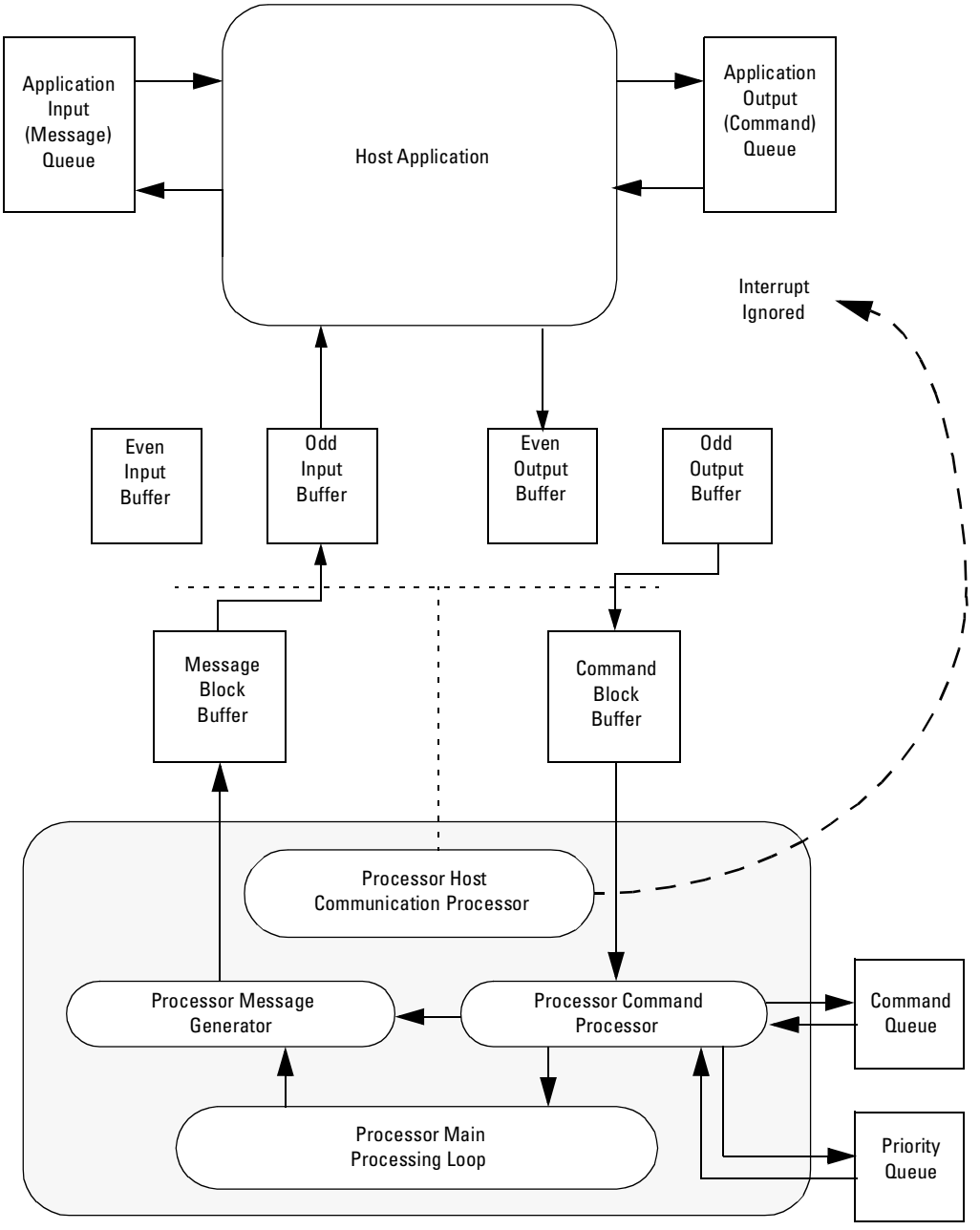

Figure 1-3 Typical Polled-Mode Host Application

2
Commands

The C-Cube Processor[1] receives, interprets and executes host commands over the PCI bus. There are two basic types of commands: *scheduled* and *priority*. Scheduled commands are executed according to a time reference and may be sent far in advance of their execution time. Priority commands are executed immediately and they have priority over conflicting scheduled commands. (See Section 2.4 for more details.)

This chapter describes the general syntax of all commands and how the host should manage them. The syntax of specific commands is discussed elsewhere in this document or in the product-specific documentation.

1. For brevity, the term “processor” or “the processor” is used from now on.

2.1 Command Syntax

All commands use a common syntax. Table 2-1 shows the general syntax for Scheduled Commands and Table 2-2 shows the general syntax for Priority Commands.

Table 2-1 Scheduled Command Syntax

Field	Length	Comment
SMPTE[1] Timecode	32 bits	"Timecodes Format" on page 34
Command Type	16 bits	
TimeCode ID	4 bits	
Stream ID	4 bits	
Command Size "K"	8 bits	In multiples of 32 bits
Data	K*32 bits	Command Syntax

1. Society of Motion Picture Technical Engineers

Table 2-2 Priority Command Syntax

Field	Length	Comment
Reserved	32 bits	0
Command Type	16 bits	
Reserved	4 bits	0
Stream ID	4 bits	
Command Size "K"	8 bits	In multiples of 32 bits
Data	K*32 bits	Command Syntax

These are the highlights of the command syntax:

- The Command Size field allows the processor to skip over syntax errors and un-implemented commands in the queue and maintain synchronization with the host.
- The K-value in the size field *does not* include the length of the K field itself, SMPTE Timecode, Command Size, Command Type or StreamID fields.

- The Stream ID field is used to select the various streams processed by the processor. Encoders, for example, use it to differentiate between audio and video streams, while codecs use it to further select between encode and decode. Commands applied to all streams should use a Stream ID of 0xF.
- The TimeCode ID is used by multi-stream products (e.g., the 4:2:2 Profile Decoder) to indicate the time code counter referred by the SMPTE Timecode. A TimeCode ID of zero is used for reference time code and single time code products.

2.2 Command Types

The processor API supports several different types of commands. Table 2-3 lists these commands.

Table 2-3 Command Types

Type	Product	Range	Comment
Reset	All	0x00 - 0x01	
Data Stream I/O Commands	All	0x02 - 0x08	"Data Stream Input and Output" on page 45
Serial Interface Command	All	0x09	"Serial Interface" on page 64
Picture I/O Commands	All	0xB - 0xC	"Picture Input/Output Interface" on page 56
Audio Configuring	All	0xD	"Audio I/O Configuration" on page 81
Context Switch	All	0xE	"Context Switch" on page 71
Init Data Stream I/O	All	0x0F	"Protocol A Data Stream I/O" on page 47
Reserved		0x10-0x8F	
Decode Commands	Decoders	0x90 - 0xAF	
Codec Commands	Codecs	0xB0 - 0xCF	
Reserved		0xD0 - 0x7FFF	
Encode Commands	Encoders	0x8000-0xFFFF	

In the above table, commands that have a reference in the "comments" column are discussed in this document. The remaining commands are specified in complete detail in the documentation for the microcode product associated with the command.

2.3 Scheduled Command Execution

Scheduled command execution occurs periodically at microcode-determined intervals. Field-pipelined microcode products can process commands each field time while frame-pipelined products can process commands each frame or picture time. Each scheduled command includes a timecode that indicates when (on which picture) it should be executed. Because the host can fill the queue with initialization and run-time commands prior to start of encoding (or decoding) and poll to ensure that the queue has not emptied, real-time host intervention to synchronize the processor and the video and audio sources is not required for all applications.

2.3.1 Timecodes Format

The processor uses the SMPTE timecode format.

31 ... 28	27 ... 24	23 ... 20	19 ... 16	15 ... 12	11 ... 8	7 ... 4	3 ... 0
Hour Tens	Hour Units	Minute Tens	Minute Units	Second Tens	Second Units	Frame Tens	Frame Units

This format is used in edit control lists and presented in the VITC[2] and LTC[3] by video tape recorders. You can configure the processor encoder software to take its reference timecode from various sources. Refer to the command specifications for each product for supported timecode sources.

The SMPTE format allows command scheduling on frame boundaries only. However, some products may additionally support a C-Cube-defined timecode format that allows field-resolution timing.

2.3.2 Command Queue Buffer Management

Each product handles its command queue differently, depending on it's pipeline structure. In general, the processor attempts to shield the host application from having to track the MPEG reordering, command implementation, and pipeline structure of each microcode product by handling these details internally. In this way, the entire product family can present a consistent interface to the host. Consult the individual product microcode reference guide for more information.

2. Vertical Interval Time Code
3. Linear Time Code

2.3.3 Timecode Wrap-Around and Expired Commands

Because SMPTE timecodes wrap around at 24 hours, there is an ambiguity as to whether a command refers to a time in the future or the past. Typically, timecodes that appear to be intended for the distant future were simply delivered to the processor too late. To resolve this ambiguity, the processor adopts the convention that any timecode for a frame more than 23 hours ahead is, in fact, an expired timecode (i.e. was intended for a frame within the previous hour). Such commands are discarded and generate an error message to the host.

Note:
LTC and free-running counters provide continuous timecodes, while VITC may have a discontinuity at each edit point. Some products may support disabling of expired timecode checking (see the product-specific documentation). This would be useful, for example, when encoding a tape of trailers where the VITC for each clip returns to 01:00:00:00.

2.3.4 Escape Timecode

When dealing with broadcast and other applications that require real-time processing, you may choose to issue commands every frame (e.g. user data), rather than queueing them up ahead of time. For this purpose, we define a special “escape” timecode (0xFFFFFFFF). When the processor encounters this timecode in the command queue, it always interprets the command as current and executes it as soon as possible.

Commands that use escape timecodes can be freely interspersed with those that use normal timecodes in the command queue. For each command cycle, the whole queue is scanned for any commands with escape timecodes. This means that commands with escape timecodes are not blocked by commands with normal timecodes even if the commands with escape timecodes are queued up after scheduled commands.

Note:
Escape timecodes may not be supported on all products.
Refer to the documentation for the specific product for details.

2.3.5 Illegal Timecodes

If the encoder encounters a command with an out-of-range timecode (e.g. greater than 24 hours or non-drop-frame timecode in drop-frame mode), the encoder discards the command and issues an error message to the host.

2.4 Priority Command Execution

Unlike scheduled commands, which are set to execute at a prescribed future time, priority commands are executed immediately.

To ensure immediate execution, when the processor receives a priority command from the host, it stores the command in a special priority command queue. This queue enables the priority command to bypass the scheduled command queue and to receive immediate attention.

The processor executes priority commands within the same command cycle in which it receives them. However, within that command cycle, the microcode executes priority commands after any scheduled commands pending for that command cycle.

Since the processor executes priority commands last in each command cycle, their effects will take precedence over any scheduled commands that perform similar functions. For example, if the host sends a scheduled command to start an encoder, but then it sends a priority command to stop the encoder during the same command cycle the previous command was scheduled for, the priority command will take precedence and the encoder will not start.

2.5 Command Execution Errors

During command execution, error messages may be generated for the following reasons:

- Expired Timecode -- the command was received too late to execute or the command was for a missing timecode.
- Syntax Error -- the command type, length, timecode, or parameter value is illegal.
- Configuration Error -- the command is illegal in the context of the current settings (e.g. NTSC with 608 lines per picture).

If the processor detects an error, it sends the host an error message. "Error Messages" on page 43 discusses error messages.

3
Messages

The C-Cube Processor[1] communicates error conditions and bitstream usage status to the host via message blocks. The host's Interrupt Service Routine scans the incoming messages and separates them by types routing them to the appropriate application's input queues.

1. For brevity, the term "processor" or "the processor" is used from now on.

3.1 Message Format

The processor uses the same format for all outgoing messages. Table 3-1 shows this format, which is similar to the format used with commands.

Table 3-1 Outgoing Message Format

Field	Length	Comment
Message Type	8 bits	
Reserved	4 bits	0
Stream ID	4 bits	
Message Size "K"	16 bits	in multiples of 32 bits
Data	K*32 bits	

The 16-bit size field permits messages as large as 64K words (256Kbytes). In practice, such large messages are not used because they exceed the 1 KB size limit for host message buffers.

3.2 Message Types

The various message types are listed in Table 3-2.

Table 3-2 Message Types

Type	Product	Range	Comment
Initialization	All	0x00 – 0x01	Section 3.2.1
Stream I/O Status	All	0x02 – 0x08	"Data Stream Input and Output" on page 45
Serial Interface Status	All	0x09	"Serial Interface" on page 64
Picture I/O Status	All	0x0B	"Picture Input/Output Interface" on page 56
Reserved		0x0C – 0x0D	
Context Switch		0x0E	"Context Switch" on page 71
Command Buffer Status	All	0x0F	Section 3.2.2
Encode Status	Encoders	0x10 – 0x1F	
Preprocessing	Encoders	0x20 – 0x2F	
Decode Status	Decoders, Codecs	0x30 – 0x3F	
Common Error Message	All	0x40	Section 3.2.3

Table 3-2 Message Types

Type	Product	Range	Comment
Encoder Error Message	Encoders	0x41	
Decoder Error Message	Decoders	0x42	
Codec Error Message	Codec	0x43	
Reserved Error Messages		0x44 – 0x5F	
Debug Information	All	0x60 – 0x9F	Section 3.2.3
Reserved		0xC0 – 0xFF	

In the above table, commands that have a reference in the “comments” column are discussed in the section identified. Some messages are common to all microcode products. We describe them below.

3.2.1 Initialization Message

The Initialization Message informs the host that the processor is initialized and is ready to accept commands. The messages in the first eight fields have a common definition across all C-Cube processor-based products. These fields provide the microcode set, revision number[2] and size of the command and message buffers. The size and contents of the last field is microcode-specific as Table 3-3 shows.

Table 3-3 Initialization Message

Message Field	Size	Comment
Type	8 bits	1 = OK, 0 = checksum error
Reserved	4 bits	0
Stream ID	4 bits	0xF
Size K	16 bits	in multiples of 32 bits
Product Code	32 bits	
SW Revision	32 bits	in ASCII (i.e. “1.02”)
Command Buffer Size	16 bits	in multiples of 32 bits
Message Buffer Size	16 bits	in multiples of 32 bits
Data	(K-3)*32 bits	Product Specific

2. Example: PAL ML@MP encoder rev 1.02

The Type Field of the Initialization Message is either *one* (normal) or *zero* (abnormal). The processor boot code does an internal checksum on its instruction and it indicates an error by setting the message type to zero.

The Input Size Field specifies the size of the host command buffers in 32-bit words and it is never greater than 1024 words. The Output Size Field specifies the size of the host message buffers and it is never greater than 256 words. We provide these fields to enable the host to double-check that it has allocated the correct size for the command and message buffers.

3.2.2 Command Buffer Status Message

The Command Buffer Status Message returns the processor's internal command queue status. The host can use this status message to control the rate of its command input. Each message includes a stream ID Field and a TimeCode ID Field. These fields are provided to accommodate microcode implementations that use separate command queues for each stream or timecode processed.[3] If only one queue is maintained for all streams, a stream ID of 0xF is used. Table 3-4 shows the format for the Command Buffer Status Message.

Table 3-4 Command Buffer Status Message

Message Field	Size	Comment
Type	8 bits	0x0F
TimeCode Id	4 bits	
Stream Id	4 bits	
Size K	16 bits	1
Status	16 bits	Nonzero is error
Emptiness	16 bits	Number of 32-bit words available in the command queue

3. The 4:2:2 profile decoder maintains three command queues, one for each time-code.

When processing commands from an internal command queue, the processor sends one Command Buffer Status Message during each command cycle. The host can use this message to determine how many more commands it can send during the next command cycle before overflowing the processor's internal command queues. The documentation for each product provides more information on the number and sizes of the command queues for each product.

3.2.3 Error Messages

Although each microcode product defines its own error messages, the general format is consistent for all products as Table 3-5 shows. As Table 3-2 earlier showed, a range of message type values is reserved for specific errors to facilitate selective logging and generation.

Table 3-5 Error Message

Message Field	Size	Comment
Type	8 bits	Error type
Reserved	4 bits	0
Stream ID	4 bits	Indicates data stream
Size K	16 bits	in multiples of 32 bits
Error Sub Type	32 bits	Facilitates message filtering
Data	(K-1)*32 bits	Depends on error subtype

An error type of 0x40 has a common definition across all products, with error sub-types as Table 3-6 shows.

Table 3-6 Common Error Sub-Types

Error Sub-Type	Name	Comment
0	Command Buffer Overflow	Host sent too much command data to fit into command queue
1	Message Buffer Overflow	More messages were generated than could fit into message block
2	PCI Parity Error	A PCI parity error was detected on incoming data
3	Stream Direction Error	An inappropriate transfer direction has been indicated for a given data stream

All other error types are covered either elsewhere in this document in association with specific commands, or are described in the product-specific documentation.

4
Data Stream Input and Output

This chapter describes the two methods for performing general stream I/O with the C-Cube Processor[1]. These methods are called Protocol A and Protocol B. The most obvious use of the streaming protocols is for transfers of MPEG bitstreams, but they are general enough to support other types of data streams as well. Some other uses of the processor stream I/O interface include the transfer of audio to and from the processor and the transfer of picture data over the PCI bus for use in previewing or rendering.

In Protocol A, the host allocates a contiguous block of its memory for a stream FIFO[2] during the entire stream I/O period. Also, the host and the processor exchange read and write pointers for the stream FIFO during each command cycle. Thus, Protocol A provides a simple communication interface but does not allow the host much flexibility in its use of memory.

1. For brevity, the term “processor” or “the processor” is used from now on.
2. A first-in-first-out circular buffer

Protocol B, on the other hand, allows the host to dynamically change the stream buffer addresses and sizes during run time. During each command cycle, the host provides the processor with a new list of buffer pointers. In return, the processor reports its usage of those buffers. Consequently, Protocol B provides memory flexibility at the cost of some loss of simplicity in the streaming interface.

Each microcode product supports different uses for the general data stream I/O interface. Also, some products may only support one of the two stream I/O protocols. Refer to the product-specific documentation for details on how stream I/O is supported for each particular product.

4.1 Protocol A Data Stream I/O

To use protocol A, the host first specifies the stream FIFO address and size after receiving the initialization message from the processor, but before issuing a start command. This is done with the Init Data Stream I/O command (Table 4-1).

Table 4-1 Init Data Stream I/O Command

Command Field	Length	Value
Reserved	32 bits	0
Command Type	16 bits	0x0F
TimeCode ID	4 bits	0
Stream ID	4 bits	Indicates data stream
Command Size "K"	8 bits	3
Reserved	24 bits	0
Flags	8 bits	■ Bits 2--7: reserved ■ StopOnError Flag (Bit 0): If *zero*, the processor continues the data stream transfers if a communication error occurs. If *one*, the processor stops the data stream transfer if a communication error occurs. ■ Direction Flag (Bit 1): If *zero*, the processor loads a stream of data from the host's data stream buffer(s). If *one*, the processor stores a stream of data on the host's data stream buffer(s).
Address	32 bits	Stream buffer address in host memory
Size	32 bits	Stream buffer size in bytes

Different products require different Stream ID field values. In most cases, only one of the two possible values for the Direction flag is allowed for a given Stream ID. If the host gives an invalid direction, the processor sends a Stream Direction Error message to the host ("Error Messages" on page 43).

The host sets the StopOnError bit-flag to direct the processor to take some action on the stream I/O data transfers when the processor detects a communication error. The processor's options are to continue data transfers or to stop them until communication error recovery is done.

The PCI data bus uses eight-byte aligned buffers to do stream I/O data transfers. Consequently, the Address and the Size fields must be multiples of eight bytes.

The Init Data Stream I/O command does not actually start the I/O process, but it rather initializes the I/O process parameters. The I/O transfers are started when the I/O process associated with the given stream ID is started. For example, the Init Data Stream command contains a stream ID for the MPEG output of an encoder. In such a case, the stream output starts when the encoding I/O process starts. Then, the processor uses the stream FIFO specified by the Address and Size fields.

4.1.1 Protocol A Data Stream Loading

When using Protocol A to load data from the host to the processor, the host updates the write pointer of the stream data FIFO in host memory to enable the processor to read more data from the FIFO. This can be done as often as every command cycle. This enables the processor to determine the maximum amount of data it can read. After it performs a data transfer, the processor sends the new read pointer back to the host enabling it to determine the space left in the stream data FIFO for writing additional data.

The host uses the Protocol A Load Data Stream command, as Table 4-2 shows, to inform the processor of the stream data FIFO write pointer, while the processor uses the Protocol A Load Data Stream Message to communicate the updated stream data FIFO read pointer to the host.

Table 4-2 Protocol A Load Data Stream Command Format

Command Field	Length	Value
Reserved	32 bits	0
Command Type	16 bits	0x03
Reserved	4 bits	0
Stream ID	4 bits	Indicates data stream
Command Size “K”	8 bits	1
Write Pointer	32 bits	Current write pointer of stream data FIFO in host memory

Table 4-3 Protocol A Load Data Stream Message Format

Message Field	Size	Value
Type	8 bits	3
Stream ID	8 bits	Indicates data stream
Size K	16 bits	1
Read Pointer	32 bits	Current read pointer of stream data FIFO in host memory

In the case of a communication error, if the StopOnError bit was clear in the Init Stream I/O command, the processor uses the last two write pointers to estimate the new write pointer and continue the data transfer. Otherwise, the processor continues stream I/O until the read pointer reaches the 8-byte aligned address closest to the last known write pointer and then stops.

4.1.2 Protocol A Data Stream Storing

The host uses the Protocol A Store Data Stream command (Table 4-4) to store processor's data on the host. This command requests the processor to write the stream data to the stream data FIFO in host memory during each command cycle. The Xfer_N_Bits field specifies the maximum number of bits the processor should write. When the processor receives the store command, it writes the requested number of bits to the stream data FIFO within the next field time. If the Xfer_N_Bits exceeds the maximum of the bits available, the processor outputs all the available bits.

The host receives the updated stream FIFO write pointer during the next field via the Protocol A Store Data Stream message (Table 4-5). From this, the host determines how many bits the processor wrote. With this message, the processor also returns the status of its internal SDRAM stream data buffer. With this information, the host can track the processor SDRAM buffer fullness and modify either the rate at which it accepts bits or the rate at which the processor generates them to prevent buffer exceptions[3] in processor SDRAM.

3. Underflows or overflows

Table 4-4 Protocol A Store Data Stream Command Format

Command Field	Length	Value
Reserved	32 bits	0
Command Type	16 bits	0x04
TimeCode ID	4 bits	0
Stream ID	4 bits	Indicates data stream
Command Size “K”	8 bits	1
Xfer_N_Bits	32 bits	Maximum number of bits that should be sent per field.

Table 4-5 Protocol A Store Data Stream Message Format

Message Field	Size	Value
Type	8 bits	4
Stream ID	8 bits	Indicates data stream
Size K	16 bits	2
Write Pointer	32 bits	The write pointer of the stream data FIFO in host memory.
Buffer Emptiness	32 bits	Emptiness of processor SDRAM buffer in bytes. Can be useful to the host as status information.

In the case of a communication error, if the StopOnError bit in the Init Data Stream I/O command was clear, the processor uses the last Xfer_N_Bits value it received to continue the data stream transfer. Otherwise, the processor stops the data stream transfer immediately.

4.2 Protocol B Data Stream I/O

In Protocol B stream I/O, the host continually provides the processor with new stream buffers as needed. The processor maintains an internal queue of the host's buffers and uses them in the order received. It then informs the host of its buffer usage so that the host can keep the buffer queue full.

All host's buffers must be eight-byte aligned and be multiples of eight bytes in size. In some products, the three least significant bits of the buffer address are set to indicate that the buffer address and size values are used to carry a PTS value. Refer to the product-specific documentation for details.

Protocol B does not require an Init Data Stream I/O command because the host supplies new buffer pointers continuously.

4.2.1 Protocol B Data Stream Loading

When using Protocol B for loading data to the processor, the host uses the Protocol B Load Data Stream Command (Table 4-6) to give buffers to the processor. When the processor uses the buffers, it informs the host of its buffer usage via the Protocol B Store Data Stream message (Table 4-7). Using this feedback mechanism, the host determines how many new buffers to send during the next command cycle.

Table 4-6 Protocol B Load Data Stream Command Format

Command Field	Length	Value
Reserved	32 bits	0
Command Type	16 bits	0x05
TimeCode ID	4 bits	0
Stream ID	4 bits	Indicates data stream
Command Size "K"	8 bits	Number of address/size pairs times two.
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
:	:	:

Table 4-7 Protocol B Load Data Stream Message Format

Message Field	Size	Value
Type	8 bits	5
Stream ID	8 bits	
Size K	16 bits	2
Buffers Used	32 bits	Number of buffers used in last field time
Address	32 bits	Address of next byte to be read by the processor. Host can use the address to detect error and recover from error.
Total Buffers	32 bits	Total number of buffers used since the last time the buffers were flushed.
Reserved	32 bits	Undefined

The Load Command allows the host to send many buffers to the processor. Each buffer is specified by its address and its size. The number of buffers in a single Load Command is limited only by the maximum value of the K field[4] and the number of empty slots left in the processor's internal buffer queue associated with the stream ID in question. The size of the internal buffer queue for each stream ID is covered in the product-specific documentation.

The Buffers Used field in the load message counts only those buffers completely used since the host sent the last load message. If the last buffer used by the processor was only partially consumed, it is not included in the Buffers Used field count.[5]

The host reads the Address field of the load message to detect a partially consumed buffer. This field points to the next byte that the processor is about to read. If this pointer is in the middle of a host buffer, it means the buffer is partially consumed.

The Total Buffers field provides the total number of buffers consumed by the microcode since the last time the buffer queue was flushed. This field is useful for keeping track of buffer usage when for some reason the host misses buffer usage messages from the microcode. In these

4. The K field is equal to twice the number of buffers.
5. Depending on the size of the buffers and the amount of data transferred, the BuffersUsed count may be zero.

cases, simply adding up the Buffers Used fields from all messages received would not be reliable.

The size of the processor's internal buffer queues along with the buffer usage information returned in the load message is enough for the host to determine the status of the internal buffer queue and to determine how many more buffers it can send.

4.2.2 Protocol B Data Stream Storing

When using Protocol B, the host uses a similar method to request that the processor *store* data into host memory as it does to request that the processor *load* data from host memory.

That is, during both operations the host provide buffers to the processor as required and the processor reports its consumptions of those buffers. The only differences between the two operations are the commands and message types used. Table 4-8 shows the Protocol B Store Data Stream Command and Table 4-9 shows the Protocol B Store Data Stream Message.

Table 4-8 Protocol B Store Data Stream Command Format

Field	Length	Value
Reserved	32 bits	0
Command Type	16 bits	0x06
Reserved	4 bits	0
Stream ID	4 bits	Indicates data stream
Command Size "K"	8 bits	(Number of buffer pairs times 2)
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
Buffer Address	32 bits	Stream buffer address
Buffer Size	32 bits	Stream buffer size in bytes
:		

Table 4-9 Protocol B Store Data Stream Message Format

Message Field	Size	Value
Type	8 bits	6
Stream ID	8 bits	
Size K	16 bits	2
BufferUsed	32 bits	Number of buffers used in the last field time.
Address	32 bits	Address of next byte to be written by the processor. Host can use this address to detect error and recover from error.
Total Buffers	32 bits	Total number of buffers used since the last time the buffers were flushed.
Reserved	32 bits	Undefined

5
Other Data Interfaces

This chapter describes other C-Cube Processor-supported interfaces, including Picture I/O, context switch commands and the Serial Interface. This latter interface can be used to configure devices that support the I^2C interface protocol.

5.1 Picture Input/ Output Interface

In addition to video I/O via its video pins, the C-Cube Processor[1] also supports picture I/O between the host and processor memory over the PCI bus. This is useful for features that require the host processor to directly manipulate the video images either produced or consumed by C-Cube Processor-based products.

Like most of the general-purpose processor interfaces, the picture I/O interface is common across all processor products. However, the different products vary in what type of picture I/O they support and what trade-offs are necessary for the I/O processes. For example, some types of picture I/O require restrictions on the picture sizes supported or require non-real-time-operation, depending on the performance constraints of the product in question. Refer to the documentation for each particular product for information on the type of picture I/O supported by that product.

In general, a host program wanting to use picture I/O first sends a Start Picture I/O Command.[2] It will then use either protocol A or protocol B data I/O (Chapter 4) to send and receive data with the processor.[3] The stream ID used in these transactions depends on the product and the type of picture I/O done. When the host program is done with picture I/O, it then sends the Stop Picture I/O Command.[4] Throughout this process, any anomalous status are reported via the processor error-reporting mechanism ("Error Messages" on page 43). See the documentation for the specific product for details.

Section 5.1.1 details the format of the Start and Stop commands, while Section 5.1.2 describes the format of the data expected by or sent from the processor.

1. For brevity, the term "processor" or "the processor" is used from now on. When referring to the picture I/O format, the term "C-Cube YUV format" is used.
2. Particular products may require additional preparation prior to picture I/O.
3. Some products support only one protocol.
4. Additional commands might be required.

5.1.1 Command Format

The Start Picture I/O Command instructs the processor to begin picture I/O with the given parameters at a specified timecode (Table 5-1).

Table 5-1 Start Picture I/O Command Format

Field	Length	Value
SMPTE Timecode	32 bits	
Command Type	16 bits	11
TimeCode ID	4 bits	See product-specific documentation
Stream ID	4 bits	See product-specific documentation
Command Size “K”	8 bits	2
Width	16 bits	Picture width in terms of luma pels
Height	16 bits	Picture height in terms of luma pels
Reserved	8 bits	Should be set to 0
Video Format	4 bits	0 - RGB 1 - RGB-Alpha 2 - UYVY 3 - YUYV 4- C-Cube YUV format (Separate luma and chroma; U and V interleaved) 5- C-Cube YUV format, luma only 6- C-Cube YUV format, chroma only 7-15 - Reserved
Chroma Format	4 bits	0 - 4:2:0 1 - 4:2:2 2 - 4:4:4 3 - 4:1:1 4-15 - Reserved
Flags	8 bits	Bit 0, header flag: 0 - Picture data will have no header 1 - Picture data will be preceded by a header Bit 1, software re-order flag 0 - Do not perform byte re-ordering in software 1 - Perform byte re-ordering in software Bits 2-7, reserved
Video Rate Divisor	8 bits	Allows for low-frame-rate picture I/O

The Stream ID Field specifies what type of picture I/O to select if different legal types are supported by different products. The Width and

Height fields specify the picture dimensions of the picture in each picture data package (Section 5.1.2). These parameters allow scaling of the video if such a feature is supported by the product in use.

The Video Format Field specifies which color space to use and how luma and chroma are combined.[5] The C-Cube YUV format consists of the Y data in one block and the UV data in another block, with the UV data interleaved in an 8U/8V repeating pattern. The Chroma Format Field gives the relationship between the chroma resolution and the luma resolution.[6]

Currently, only two bits in the Flags Field are used. Bit zero is used to signal the processor whether or not the picture data header defined in Section 5.1.2 should precede the actual picture data. Though the header is useful for error checking and recovery, it is not absolutely necessary and in some applications it is wasteful.

Bit one of the Flags field is used to turn software byte re-ordering on or off. When set, the order of bytes in each four-byte word of output or input will be reversed by the microcode. This is useful for applications where the host needs to manipulate the data, uses a different byte-ordering paradigm (endian) than the microcode, and the revision of processor being used does not support hardware re-ordering. (Early versions of the processor do not support hardware re-ordering, but later versions will.) However, support for software re-ordering is limited to only certain situations; see the product-specific documentation for details. On later revisions of the processor, hardware re-ordering should be used instead of software re-ordering, unless the application needs to maintain compatibility between drivers used for early revisions and drivers used for later revisions.

If picture data headers are used in the picture stream, they will *not* be re-ordered, even if the re-ordering flag is set. This is so that the multi-byte data items that are present in the header can be easily read by the host processor without re-arranging the byte order. Though the single-byte data items in the header will appear in a different order than that used by

5. The RGB, RGB-Alpha, UYVY, and YUYV formats consist of the indicated pattern repeated to achieve the desired resolution.
6. This field is used to specify standards 4:2:0, 4:2:2, 4:4:4 and 4:1:1.

the microcode, this is more easily dealt with than re-arranging bytes within a data item.

The Video Rate Divisor parameter allows the host to slow down the rate of picture I/O. This is useful if the host lacks the performance to keep up with I/O tasks at full-frame-rate. The Divisor parameter is the number of pictures that should be skipped between pictures sent or received. For example, a value of zero is full-frame-rate I/O, while a value of 1 divides the frame rate of the picture I/O process by two.

Table 5-1 above, lists the full range of picture I/O parameters which are supported by the general picture I/O interface format. Some microcode products may only support a small subset of the parameter values listed by the table. Each type of picture I/O supported places constraints on the allowable values for Width, Height, Video Format, Chroma Format, and Video Rate Divisor. The Start Picture I/O Command is general enough to handle all the different types of picture I/O that are done by processor products. Consult the specific product documentation for limitations placed by the product on the Start Picture I/O parameters.

While the Start Command starts the picture I/O operation, the Stop Picture I/O Command causes the processor to abort it. Table 5-2 shows the Stop Command format.

Table 5-2 Stop Picture I/O Command Format

Field	Length	Value
SMPTE Timecode	32 bits	
Command Type	16 bits	12
TimeCode ID	4 bits	
Stream ID	4 bits	Must match a previous Start command
Command Size “K”	8 bits	0

Some C-Cube-processor products support multiple timecodes that can be used to schedule commands. Typically in these cases, one timecode is *free-running* according to some external signal such as VSYNC, while the others are associated with the processor frame processing loop. This can be an issue when the picture I/O being done requires non-real-time processing because of the following:

- A free-running timecode does not provide an accurate measure of the number of frames processed and thus should not be used to schedule Start Picture I/O and Stop Picture I/O commands if the host wishes to have precise control over the number of pictures processed.
- In single-timecode products, the timecode accurately tracks the non-real-time processing loop, but for multiple-timecode products, the host needs to be careful to supply the appropriate timecode ID in the Start Picture I/O and Stop Picture I/O commands in order to use a timecode reference that accurately tracks the processing loop. The product-specific documentation details the timecode IDs available for a particular product.

Along with the timecode rate, the host interrupt rate may also be slowed in accordance with non-real-time behavior, since the host interrupt may be gated by the field processing loop, depending on the microcode being used and whether or not video-sync mode is used (“Microcode Start Mode Flag” on page 11). In any case, the host drivers that communicate with and control the processor should not need to do anything special to handle non-real-time processing.

5.1.2 Data Format

During the interval of time between the execution of a Start Picture I/O Command and a Stop Picture I/O Command, picture data is transferred over the PCI bus using either Protocol A or Protocol B data I/O. (See Chapter 4 for more details). A host program using picture I/O should use either the Load Data Stream Command or the Store Data Stream Command (Chapter 4) to send or receive data.

In the case of picture output, the processor ensures that the output data conforms to the standard described in this section, but for picture input, the host needs to present the picture data in the format described here.

The packaging of the picture data (e.g. in fields or frames) and the interval between data transfers varies from product to product, but all products share a basic data format. In this basic format, there is a small header for the data package followed by a data payload for the package, where the data payload is typically either a field or a frame depending on the type of picture I/O done.[7]

Table 5-3 shows the data format. Note the parallelism between the header fields in the data format and the picture parameters in the Start Picture I/O Command.

Table 5-3 Picture I/O Data Format

Field	Length	Value
Magic Number	32 bits	0x00FF00FF
Timecode	32 bits	For picture output: SMPTE timecode or frame counter For picture input: Ignored
Width	16 bits	Picture width in terms of luma pels
Height	16 bits	Picture height in terms of luma pels
Reserved	20 bits	0
Video Format	4 bits	0 - RGB 1 - RGB-Alpha 2 - UYVY 3 - YUYV 4- C-Cube YUV format 5- C-Cube YUV format, luma only 6- C-Cube YUV format, chroma only 7-15 - Reserved

7. It will depend on the product and the stream ID.

Table 5-3 Picture I/O Data Format

Field	Length	Value
Chroma format	4 bits	0 - 4:2:0 1 - 4:2:2 2 - 4:4:4 3 - 4:1:1 4-15 - Reserved
Stream ID	4 bits	Whatever was given in the Start Picture I/O command
Reserved	31 bits	0
Top Field Flag	1 bit	If a field data package: 0 - Top field of a frame 1 - Bottom field of a frame If not a field data package: Always 0
Payload Size	32 bits	Size of picture data payload in bytes, not including header
Picture Data Payload	As given in Payload Size	Picture data

The value of the Magic Number Field is extremely unlikely to occur naturally in video data and thus can be used as a value to search for errors and do error recovery. The Timecode Field is valid for picture output only.[8] It provides a SMPTE timecode or frame counter value, which is synchronized to the picture's output rate.[9] Exactly how the timecode corresponds to the picture varies from product to product. For instance, an encoder might tag a picture with a timecode indicating when it was captured, while a decoder might tag it with a timecode indicating when it was decoded; refer to the product-specific documentation for details. In any case, the Timecode Field allows the host to determine the order in which pictures should be displayed, because in some special cases the *Picture I/O order* and the *display order* are not the same.[10]

8. It is ignored during picture input.
9. The actual timecode type, i.e. SMPTE or frame counter, is configured by the processor.
10. Display order is the order in which pictures are presented to the viewer.

The Width and Height fields give the pixel dimensions for the data package, whether that package corresponds to a field, a frame, or some other video data set. These should be the same as the dimensions given in the Start Picture I/O Command that started the data transfer. Note that different products and different picture I/O types require different data packaging and thus will imply slightly different interpretations of the picture dimensions.

The Video Format Field is the same as the corresponding field in the Start Picture I/O Command. When the C-Cube YUV format is used, luma and chroma are not interleaved but are transferred in a single package with one header, luma and chroma in that order.

The Chroma Format Field indicates the relative resolution of the chroma data. When chroma is transferred by itself (e.g. when the video format is C-Cube YUV format, chroma only), the picture dimensions in the header are ***not*** different for different chroma formats. That is, the picture size is expressed as luma pixels. Video format, chroma format, and the picture dimensions must be considered when deriving the actual size of the data payload.

The Top Field Flag is used when the picture data is packaged field by field; otherwise it is zero. It indicates whether the input or output field is the top field of a frame or the bottom field. For picture input, the host is responsible for setting this flag correctly in the header, while for picture output, the processor is responsible.

Because the parameters in the Start Picture I/O Command are constrained differently for different products and different types of picture I/O, the fields in the picture data header are constrained in a corresponding manner. The header is general enough to handle all the possible formats that are supported by processor products, even if a single product is not using the full range of values allowed in the header fields.

If the header bit in the Flags Field of the Start Picture I/O Command was clear, the picture data package has no header. In this case, only the bare data payload is sent in each processing interval. The host derives the size of the picture data payload from the type of picture data requested.

5.2 Serial Interface

The processor has microcode-controlled I/O pins that are used to connect to a Serial Interface[11] bus to control peripheral devices such as video/audio encoder/decoder chips. All processor microcode products provide two ways for the host to program the Serial Interface devices: either through initialization memory or through the command interface as the next two sections describe.

5.2.1 Initialization Memory Format

To program Serial Interface devices through initialization memory, the host loads the Serial Interface data according to Table 5-4 before starting the processor. The processor initializes Serial Interface devices by loading their programming sequences from the Serial Interface initialization memory before starting communication with the host. This method is used when the host needs to program Serial Interface devices in order to generate video clock for the processor because the processor serial interface relies on the video clock to start the communication with

11. This interface is used to configure devices that support the I^2C protocol.

the host. The SDRAM address and size for Serial Interface initialization memory is specified in the documentation for each product.

Table 5-4 Serial Interface Initialization Memory Format

Field	Length	Value
Number of devices to be initialized (N)	32 bits	
Serial interface clock frequency	16 bits	Serial interface clock frequency in terms of number of cycles in processor clock. e.g. for 100 Khz Serial interface clock frequency in 100Mhz processor, the value should be 1000.
Serial interface flag	8 bits	Bit 0, Serial interface format flag: 0 - Standard Serial interface format with start/ stop condition. 1- No stop condition; used for special Serial interface devices (e.g. MSP 3410P) Bits 1-7: Reserved; should be 0.
Device #1 Serial interface address	8 bits	The peripheral device Serial interface bus address. The least significant bit will be ignored.
Bytes for transfer (K_1)	16 bits	Number of bytes for transfer
Data	$8*K_1$ bits	
Serial interface flag	8 bits	(same as above)
Device #2 Serial interface address	8 bits	The peripheral device Serial interface bus address. The least significant bit will be ignored.
Bytes for transfer (K_2)	16 bits	Number of bytes for transfer
Data	$8*K_2$bits	
:	:	:
:	:	:
Serial interface flag	8 bits	(same as above)
Device #N Serial interface address	8 bits	The peripheral device Serial interface bus address. The least significant bit will be ignored.
Bytes for transfer (K_N)	16 bits	Number of bytes for transfer
Data	$8*K_N$bits	

5.2.2 Serial Interface Command Format

One can also program Serial Interface devices at any time after initialization by using the Serial Interface Command that Table 5-5 shows. When examining the table, note the following:

- Bytes for Transfer Field. The processor transfers data with the host in 8-byte multiples. This means that for read or write requests, the host needs to allocate enough memory for the processor to round up to the next 8-byte boundary.
- Serial Interface Sub-Address Length Field.This field is used to support addressing of more than one byte. For a read operation, this field indicates the number of bytes in the location pointed to by the Data/Address Field. The data at this location is combined with the Device Serial Interface Address Field to form the full device address. The result of the read operation overwrites the sub-address data.

  For a write operation, the sub-address information is obtained either from the host buffer or the Data/Address Field (see below) depending on bit 1 of the Serial Interface Flag Field. In either case, the data to be written will be expected to follow the sub-address information.
- Data /Address Field. For a read request or a write request where bit 1 of the Serial Interface Field is set, this Data /Address Field contains the host memory address of the buffer allocated for the data to be read or written. In this case, the last 3 bits have to be zero (i.e. 8-byte aligned), and K should be equal to 4.

  If the Serial Interface Sub-Address length is not 0, the sub-address is stored at the beginning of the host buffer.

.

Table 5-5 Serial Interface Command Format

Field	Length	Value
SMPTE Timecode	32 bits	
Command Type	16 bits	9
TimeCode ID	4 bits	0
Stream ID	4 bits	0
Command Size “K”	8 bits	At least 4. Could be more if bit 1 of the Serial interface Flag field (see below) is clear.
Read/Write	8 bits	0 - Read request 1 - Write request
Request ID	8 bits	ID for identifying the command
Bytes For Transfer	16 bits	Number of bytes for transfer. (Note: The processor can only transfer data from the host in 8-byte multiples. This means that for a read request or for a write request where the data comes from host memory, the host has to allocate enough memory for the processor to round up to the next 8-byte boundary.)

Table 5-5 Serial Interface Command Format

Field	Length	Value
Serial interface Flag	8 bits	Bit 0, Serial interface Format flag: 0 - Standard Serial interface format with start/stop condition. 1- No stop condition; used for special Serial interface devices (e.g. MSP 3410P). Bit 1, Address/Data flag: 0 - Data/Address field (below) contains actual data for write operation. Not valid for a read operation. 1 - Data/Address field contains pointer to data buffer in host memory. Bit 2, Address/Data flag: 0 - Data/Address field (below) contains Data or Address depending on bit 1. 1 - Data/Address field contains pointer to data buffer in host memory. The data in host memory is in the same format as Init Serial interface string. This bit overrides the bit1 value. Ignored if chip is not on Standby. Bits 3-7: Reserved; should be 0.
Device Serial interface Address	8 bits	The peripheral device Serial interface bus address.
Serial interface Clock Frequency	16 bits	Serial interface clock frequency in terms of number of the cycles of the processor clock; e.g. for a 100 Khz Serial interface clock frequency and a 100Mhz processor, the value should be 1000.
Reserved	24 bits	0
Serial interface Sub-Address Length	8 bits	0 to 255. This field is used to support addressing of more than one byte. For a read operation, this field indicates the number of bytes in the location pointed to by the Data/Address field that should be used as a part of the Serial interface address in addition to Device Serial interface address field. The result of read will overwrite the sub-address data. For a write operation, the sub-address information will be obtained either from the host buffer or the Data/Address field (see below) depending on bit 1 of the Serial interface flag field. In either case, the data to be written will be expected to follow the sub-address information.

Table 5-5 Serial Interface Command Format

Field	Length	Value
Data /Address	32 bits * (K-3)	For a write request where bit 1 of the Serial interface Flag field is clear, Data/Address contains the actual data to be written. For a read request or a write request where bit 1 of the Serial interface field is set, Data/Address contains the host memory address of the buffer allocated for the data to be read or written. In this case, the last 3 bits have to be zero (i.e. 8-byte aligned), and K should be equal to 4. If the Serial interface Sub-Address length is not 0, the sub-address is stored at the beginning of the host buffer. If the data to be transferred is less than a multiple of 4 bytes in length, the data will occupy the lower byte addresses of a 4-byte word.

To keep track of Serial Interface requests, the host associates a request ID with each Serial Interface Command. The processor inserts this request ID in the Serial Interface status message it returns so that the host can identify the request to which it corresponds. Table 5-6 shows the Serial Interface message format.

Table 5-6 Serial Interface Message Format

Message Field	Size	Value
Type	8 bits	9
Reserved	8 bits	Undefined
Size K	16 bits	1
Reserved	8 bits	Undefined
Request ID	8 bits	ID corresponding to a previous command
Reserved	8 bits	
Status	8 bits	0 - Success 1 - Fail

In response to a serial interface command, the processor firmware simply copies the data from host memory or from the command itself (depending on bit 1 of the serial interface Flag field) to the serial interface bus or reads the data from serial interface bus to the host memory. To ensure that the serial interface transaction is complete, the

host should not modify its serial interface buffer until a message with the associated request ID has been received. Also, to allow the programming of several serial interface devices using only one command, one can set the serial interface_Flag bit 2. In this case, the data copied from the host memory is assumed to be in the serial interface initialization memory format presented earlier. In this specific case (serial interface_Flag bit 2 set), the serial interface will be initialized if the processor is in standby (i.e. only performs host communication). If serial interface_Flag bit 2 is set, bit 1 is ignored and device serial interface address is ignored as well.

Although Serial Interface commands are allowed while the processor is in the active state, they will directly impact the processor performance. To ensure real-time performance, the host must limit Serial Interface data transfers to one byte of data at a 100 KHz Serial Interface clock frequency during each field time. It is recommended for the host to stop encoding or decoding before issuing Serial Interface commands.

5.3 Context Switch

In applications that require a context switch between decoder microcode and encoder microcode, it is necessary to save the current processor state before switching to the other state. This is done with the Context Switch Command described in this section.[12]

The processor supports context switches only at closed GOP boundaries. Context switches allow the host application to decode one GOP of pictures using decoder microcode, perform its own graphics processing on the resultant pictures, and then re-encode those pictures using encoder microcode. Typically, context switches are used with picture I/O (Section 5.1), which allows the pictures to be input and output through the PCI bus instead of the video pins, thus making it easier for the host to capture and manipulate the picture data.

Table 5-7 shows the syntax of the Context Switch Command.

Table 5-7 Context Switch Command Format

Field	Length	Value
SMPTE Timecode	32 bits	
Command Type	16 bits	0xE
TimeCode ID	4 bits	
Reserved	4 bits	0
Command Size “K”	8 bits	2
Address	32 bits	Address of the host buffer for storing the state of the program. For the buffer size requirement, refer to the microcode-specific manual.
Reserved	32 bits	0

The host allocates a buffer for the processor to store its state information. The size for the state buffer varies for each product; see the product-specific documentation for details. The host does not need to know the actual contents of the state buffer.

12. As “Microcode Start Mode Flag” on page 11 described, the current state is loaded back into the processor the next time that microcode is used by setting bit 1 of the Start Mode shared memory location to one and providing a pointer to the state buffer in the State Address shared memory location.

A Context Switch Command follows the encoder's or decoder's start[13] command. When the processor receives a Context Switch Command, it waits until the current encoding or decoding process is complete before executing the context switch.[14] It then stores the state of the program into the host-specified memory address after the last frame to be encoded or decoded has traversed the entire processing pipeline.

Once the processor is done writing its state information, it issues a Context Switch message to the host. This message indicates to the host:

- That all microcode operations are complete.
- It is safe to halt the current microcode.
- To load the other microcode for the next batch of processing.

Table 5-8 outlines the Context Switch message.

Table 5-8 Context Switch Message Format

Message Field	Size	Value
Type	8 bits	0xE
Reserved	4 bits	Undefined
Reserved	4 bits	Undefined
Size K	16 bits	1
Reserved	24 bits	0
Status	8 bits	0 - Success 1 - Fail

In context switch applications, it is typically important for the context switch time to be as small as possible.To facilitate this, the host should do the following:

- If the application does not require normal video output (i.e. it uses the picture I/O interface), the host should set bit 0 of the Microcode Start Mode Flag shared memory location (Section 1.2.3) so that the microcode does not attempt to synchronize with the VSYNC signal. Otherwise, extra time is wasted to perform the synchronization.

13. Refer to the specific microcode product's documentation for command names.
14. Thus, the host does not have to schedule the command with perfect accuracy.

- Hardware initializations via the serial interface (Section5.2) typically only need to be performed once when a system is powered up or reset. Thus, when microcode is reloaded for a context switch, the serial interface initializations can usually be skipped. This can be done by setting the first 32-bit word of the serial interface initialization memory to zero before starting the microcode. Since serial interface initialization is very time-consuming, this will reduce the context switch time considerably. For this to work, however, both of the microcode sets involved in the context switch need to be able to run using the same set of serial interface initializations. Also, the host must be careful not to reset any of the hardware initialized via the serial interface when re-loading the microcode sets.
- After the host receives a context switch message, it is possible for it to load the next set of microcode *without* resetting the chip. This is useful when a serial ROM is connected to the processor, since resetting the chip will automatically trigger the transfer of data from the ROM to the processor, which can be very time-consuming (on the order of 500 milliseconds). To allow code to be loaded without resetting, the microcode will run a small stub in IMEM that will wait for new code to be loaded into SDRAM and then jump to location zero to start the new code. On the host side, the loader should skip the first three sections of the UX file, which reset the chip and the SDRAM controller. It should also skip the last two sections, which set the chip to the run state and wait for the Loader State shared memory location to show a value of 5. All the other sections, which load the microcode and data tables, should be executed. These also include the section that sets the Loader Command shared memory location to one, which is also important as this is the location checked by the context switch stub to determine when the host has completed loading the new code. Within a millisecond of the loader writing this location, the stub will transfer control to the new microcode and normal execution will begin.

6
System Timing

This chapter discusses the real-time operations that video field interrupts generate. The timing of buffer transfers and host interrupts is critical in guaranteeing communication between the C-Cube Processor[1] and the host. The critical timing events are:

- Video Field Synchronization
- Processor Command/Message Buffer Transfers
- Processor-To-Host Interrupt
- Processor Bitstream Transfers.

1. Unless inappropriate, the term “processor” or “the processor” without “C-Cube” is used from now on.

6.1 Overview

The video input for encoders or video output for decoders generates a synchronization interrupt to the processor once per field. This prompts the processor to begin command and message buffer transfers over the PCI interface. These buffers are small and can transfer relatively quickly. When these transfers are complete, the processor interrupts the host to signal that the next command and message buffers are ready for processing.

The time interval between the video sync interrupt and the host interrupt varies with microcode products, but is such that the host sees a consistent 1/60th second between interrupts for NTSC and 1/50th for PAL. The processor-to-host bitstream transfer for encoders and/or the host-to-processor transfers for decoders occurs after the host interrupt and before the next video sync interrupt. These transfers can be large (up to 250Kbytes) and are allocated a relatively large amount of time. Figure 6-1 shows the relative timing of the interrupts and the data transfers between the host and processor.

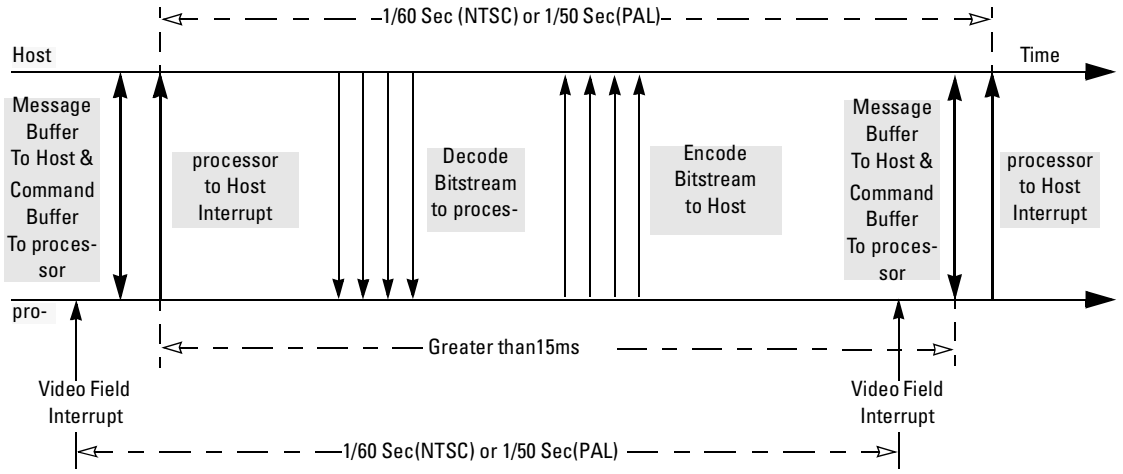

Figure 6-1 Host Interface Timing

For non-real-time applications, such as the input or output of a frame over the PCI bus, the processor communicates with the host right after it processes each field of data. The processor also ensures that there is no more than one interrupt per processor per field time.

6.2 Timing

This section describes the process of ensuring proper system timing.

6.2.1 Command Timing

The processor handles command timing as follows:

- First, it fetches the command buffer from the host no earlier than 15 ms after it interrupts the host.
- Next, it waits for the command and message transfers to complete, then it interrupts the host.
- After the host interrupt, the processor processes the command buffer.[2]
- Last, the host updates the next command buffer between the host interrupt and the next video field interrupt.

6.2.2 Message Timing

The processor handles message timing as follows:

- First, the processor reads the command block, then transfers the message buffer to the host.
- Next, it interrupts the host to indicate that the command and message transfers are complete.
- After the host interrupt, the message buffer is ready for the host to process.
- Last, the processor updates the next message buffer no earlier than 15 ms after it interrupts the host.

2. The details of the command processing are unique to each program track.

6.2.3 Encode Bitstream Timing

When encoding is active, every command buffer contains a Store Data Stream Command that indicates the number of bits for the next encode bitstream transfer.

The processor handles encode bitstream timing as follows:

- The command indicates the number of bits to transfer in Protocol A. For Protocol B, the number of bits are specified in the buffer list.
- Before the next video field interrupt, the processor transfers the maximum number of specified bits to the designated bitstream buffer. In the message buffer, it includes a Store Data Stream message. In Protocol A, this message indicates the number of bits written or, in Protocol B, the buffers consumed.

When running *internal* bit rate (open loop) reference,[3] the host is responsible for removing the average number of bits per field from the bitstream buffer between host interrupts and putting this number of bits in the Store Data Stream command.

In *external* bit rate (closed loop) control, the host should monitor the number of bits that the channel requires between host interrupts and put this number of bits in the Store Data Stream Command.

6.2.4 Decode Bitstream Timing

If Protocol A is used and decoding is active, every command buffer contains a Load Data Stream Command to indicate the updated host buffer write pointer.

When using Protocol B, a Load Data Stream command may not be necessary in every command cycle if there are enough buffers left in the processor's buffer queue. The processor is responsible for transferring the average number of bits per field from the designated bitstream buffer(s) before the next video field interrupt and then including a Load Data Stream message in the message buffer to update the number of bits consumed.

---

3. This is where the host determines the bitrate.

6.3 System Clock Reference

The processor maintains a hardware register called “PTS counter” as a System Clock Reference (SCR). This register acts as a 12-bit 90 kHz counter. The processor constructs the 33-bit PTS counter by detecting the wraparound of the 12-bit PTS counter and maintaining the most significant 21 bits in SDRAM.

If the host does not need to directly set the PTS counter, the host can use the PTS offset command to change the PTS counter. Consult the product-specific documentation for details.

In some applications, the host might need to set the PTS counter directly at the instant that it receives an SCR value. In those cases, the host needs to follow the procedure described below to set the PTS counter. Table 6-1 shows the processor SDRAM addresses for setting the PTS counter.

Table 6-1 Variables for Setting the PTS Counter

Address	Data	Description
0xFC0ABC	Hardware Video PTS counter	bit 12.. 23 (i.e. 12 bits)
0x40018	Microcode PTS upper word	PTS upper 21 bits maintained by processor
0x4001C	Microcode PTS lower word	PTS lower 12 bits maintained by processor
0x40020	Host_Semaphore	Semaphore flag set by the host
0x40024	Processor_Semaphore	Semaphore flag set by the processor

Whenever the host needs to set the PTS counter, it also needs to update the Software PTS value. To avoid race conditions, the host should follow the procedure below with the processor to handle the PTS counter:

1. Set the Host_Semaphore value to 1.
2. Poll Processor_Semaphore until the value is zero.[4]
3. Program the Hardware Video PTS counter bits 12..23 with the desired lower 12 bits of the SCR.
4. Set the Software PTS upper and lower values to reflect the new SCR.[5]

4. The processor releases the Processor_Semaphore in less than 1 tick of a 90 KHz clock.

5. Set the Host_Semaphore value back to 0. It is required that the host completes the procedure in less than 3 ticks of a 90KHz clock (i.e. ~3000 100Mhz cycles).[6]

5. The host should adjust the PTS counter no more than once per field. If the host adjusts the PTS counter more than once per field, the processor real-time performance might be impaired.
6. This method of PTS update may not be supported on all products. Consult the product-specific API for details.

7
Audio I/O Configuration

This chapter describes the configuration and capabilities of the audio I/O channels on the C-Cube Processor.[1] Refer to the C-Cube Processor Hardware User's Manual for more details.

1. Unless inappropriate, the term "processor" or "the processor" without "C-Cube" is used from now on.

7.1 Audio Processing Overview

The processor has two audio ports; one input, one output. The input port receives audio in the range of 16 bits to 32 bits. The processor captures a maximum of eight channels of digital audio, in the same range, from external audio A/D, D/A, or AES/SPDIF interface chips for editing and off-line compression. The processor also playbacks uncompressed data for editing.

The C-Cube Processor supports simultaneous capture and playback of a maximum of eight channels of audio data. There are four input pins and four output pins, each which can handle a pair of channels. All of the channels are synchronized to a single audio clock (ACLK). There are STATUS and FSYNC pins for both the input pins and the output pins (Figure 7-1). This interface support a number of popular ICs available in the market today. When communicating with the host, the processor uses this interface as follows:

- Provides an interleaving buffer for transferring audio data with the host. This buffer handles channel data in powers of two (x^2).
- When a stereo pair is captured, every even sample comes from channel zero and every odd sample from channel one.
- When capturing extra channels,[2] samples from un-used channels are also used to maintain the required x^2 number of channels.[3]
- Uses DMA to transfer blocks of sample data with SDRAM. The processor latches its PTS counter[4] at the first bit of the first sample that goes through the audio pins. For applications that capture audio, this timestamp is inserted in an optional header block pre-appended to each block of interleaved samples transferred to the host (See Section 7.4).
- In general, the sampling rate and number of channels is the same for both capture and playback.

2. As in the case of an stereo pair and a linear timecode (LTC)
3. For example, with six-channel captures, the un-used two channels have null data.
4. Refer to "System Clock Reference" on page 79 for more details

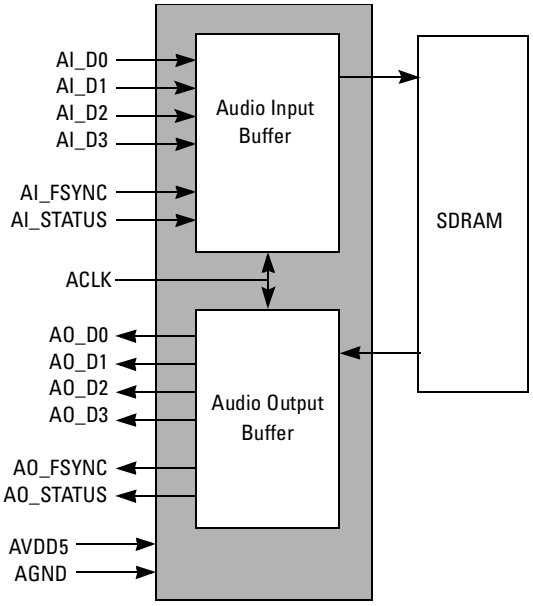

Figure 7-1 C-Cube Processor Audio Interface Block Diagram

7.2 Audio Frame Formats

The processor supports seven audio frame formats. Table 7-1 lists these formats with the product that they support.

Table 7-1 Supported Audio Formats

Format	Product Compatibility
FRFORM 0	DSP chips
FRFORM 1	Crystal 4216, 4218 Codecs
FRFORM 2	Not Supported
FRFORM 3	Crystal ADCs and DACs
FRFORM 4	I^2S devices
FRFORM 5	Typical multibit DACs and interpolation filters
FRFORM 6	Typical multibit DACs and interpolation filters
FRFORM 7	Typical multibit DACs and interpolation filters

Table 7-2 specifies the external characteristics of the frame formats.[5]

Table 7-2 Audio Frame Format

FrForm Value	Samples/ Frame	Pulse Sync Type	Frame Sync Transition
0	1	Pulse	First bit of each sample
1	2 + 2 status	Pulse	First bit of first sample in frame
2	Reserved	Reserved	Reserved (n/a)
3	2	Left High/Right Low	First bit of each sample
4	2	Left Low/Right High	One clock before first bit of each sample
5	2	Left High/Right Low	One clock after last bit of each 16-bit external sample
6	2	Left High/Right Low	One clock after last bit of each 18-bit external sample
7	2	Left High/Right Low	One clock after last bit of each 20-bit external sample

While examining Table 7-2 above, note that the following information applies to *all supported* frame formats:

- *Samples/Frame* specifies the number of audio samples transmitted per frame sync cycle on each audio pin.
- *Pulse Sync Type* is a low-to-high transition on inFsync or outFsync during the first bit or the clock, but before the first bit of the first sample in each audio frame. On output, pulse frame sync is high for one clock, and then goes low until the next frame. On input, the high to low transition is ignored.
- *Left High/Right Low Sync* is high during left channel sample bits and low during right channel bits.
- *Left Low/Right High Sync* is inverted. The transition of frame sync from low to high and high to low defines the beginning or end of each sample as defined in the frame sync transition column of the above table.

5. See the C-Cube Processor Processor Hardware User's Manual for detailed descriptions and timing diagrams of each format.

- *Frame Sync Transition* is input from the AI_FSYNC pin for audio capture and output on AO_FSYNC for audio output. Frames need not be back-to-back on input. The number of clocks in each sample is at least 16.

The processor outputs sample data and status data from the most significant to the least significant bit. Table 7-3 shows the input and output timing between the processor's Control Bus and its I/O registers (Section 7.3.2 and Section 7.3.3).

Table 7-3 Processor I/O Operations

Memory Size	Input or Output	Clocks/ Sample	Processor Operation
16 bits	Input	16 ~ 32	Input samples are truncated to the most significant 16 bits.
	Output	16	All bits in sample are used.
		32	For frame formats 0, 1, 3, 4-- Left-justified with 16 trailing zeros For frame formats: 5, 6, 7-- Right-justified with 16 leading zeros
32 bits	Input	16 ~ 32	Input samples are left-justified with complementary trailing zeros.
	Output	16	Only the most significant 16 bits are used.
		32	All bits in sample are used.

Note the following from the above table.

- For *input* samples:

  When the memory size is 16 bits, the processor truncates input samples to the most significant 16 bits.

  When the memory size is 32 bits, the processor left-justifies input samples and adds complementary trailing zeros.

- For *output* samples:

  When the memory size is 16 bits, the processor's output actions depend upon both the clock-sample rate and the format. These dependencies are as follows:

  If the sample rate is 16 clocks per sample, all bits in the sample are used.

If the sample rate is 32 clocks per sample, and the frame formats are 0, 1, 3 or 4, the sample is left-justified with 16 trailing zeros.

If the sample rate is 32 clocks per sample, and the frame formats are 5, 6 or 7, the sample is right-justified with 16 leading zeros.

- Additionally for *output* samples, when the memory size is 32 bits, the processor's output actions depend upon the sample rate. These dependencies are as follows:

  If the sample rate is 16 clocks per sample, only the most significant 16 bits of the sample are used.

  If the sample rate is 32 clocks per sample, all bits in the sample are used.

Table 7-4 shows information specific to FRFORM 1.

Table 7-4 FRFORM 1 Timing References

Output Frames	Input Frames	Reference Document
The first word is the left[1] channel sample.	The first word is the left sample.	C-Cube Processor Hardware User's Manual
The second word is the most significant 16 bits of the status out register.	The second word is captured in the *status registers*.	Section 7.3.2 and Section 7.3.3
The third word is the right output sample	The third word is the right channel sample; the fourth is not used.	C-Cube Processor Hardware User's Manual
The fourth word is the least significant 16-bits of the status out register.	n/a	C-Cube Processor Hardware User's Manual

1. Lowest channel number

7.3 Audio Configuration Interfaces

The programmer has two interfaces available for accessing audio I/O. The first interface is the Audio Configuration Command, which is used for configuring audio. The second is the I/O registers, which are used for getting status.

7.3.1 Audio Configuration Command

Audio I/O is configured with the Audio Configuration command, described in Table 7-5. This command should be issued at the time of the initial configuration of the processor. Audio control operates differently in the different products; please refer to the product specific manual for further information.

Table 7-5 Audio Configuration Command Parameters

Field	Length	Value
Reserved	32 bits	0 (i.e. priority only)
Command Type	16 bits	0xD
Reserved	4 bits	0
Stream ID	4 bits	0x7 (words)
Command Size "K"	8 bits	
Frame Format	8 bits	Specify the external audio frame format as described in preceding sections of this chapter. The valid values of the byte are as follows: ■ 0: Used for communicating with DSP chips ■ 1: Supports Crystal 4216, 4218 Codecs ■ 2: Reserved ■ 3: Supports Crystal A/D and D/A chips ■ 4: Supports I-squared-S parts ■ 5-7: Supports typical multibit DACs and interpolation filters.
Output Synchronization	8 bits	0: Output framing generated by processor. 1: Output framing synchronized to input framing
Clock Per Sample	8 bits	0: 16 clocks per sample 1: 32 clocks per sample Note: only used if Output synchronization is not set (value 0)
Bit Per Sample (See Table 7-3 for packing information.)	8 bits	0: 16 bits per audio sample. 1: More than 16 bits is captured as a 32-bit word ("memory size" is 32).

Table 7-5 Audio Configuration Command Parameters

Field	Length	Value
Sync Clock Edge	8 bits	0: Audio serial data and frame sync clocked on falling edge of s-clock input. 1: clocked on rising edge
Little Endian	8 bits	0: Big endian (msb at lowest address) 1: Little Endian (lsb at lowest address)
Channel Count	8 bits	0: 1 channel captured on audio pin 0 1: 2 channels captured on pin 0. 2: 4 channels captured on pins 0, 1. 3: 8 channels captured on pins 0, 1,2, and 3.
Video Coupling	8 bits	0: Loosely couple with video. Audio keeps capturing while video sync is lost. 1: Tightly couple with video. Halt audio capture while video sync is lost. Automatically restart capturing once video sync is restored.
Sample Rate	32 bits	Nominal Rate in samples per second per channel
Samples Per Audio Block (See Table 7-6 for Audio Block Format.)	32 bits	Number of samples per audio block transferred to the host. Consult product specific manual for limits on the block size.
Insert Header	8 bits	0: Timestamp and header information will not be inserted in the audio data stream. 1: Header information will be inserted. This will be done by the host in play mode, and by the processor in capture mode.
Capture	8 bits	0: Do not configure for audio capture 1: Configure for audio capture
Play	8 bits	0: do not configure for audio play 1: configure for audio play
Loopback	8 bits	0: Do not loopback input audio 1: loopback input audio. Note: play and loopback are mutually exclusive. If loopback is desired, set the play parameter to zero.
Reserved	64 bits	Must be 0

7.3.2 Audio Status In_0/1 Registers

The Audio Status In_0 (0xFC0B08) and the Audio Status In_1 (0xFC0B0C) are two 32-bit registers, memory-mapped to their respective Control Bus addresses. These registers contain the data clocked in from the StatusIn pin of the C-Cube Processor Processor. These registers operate as follows:

1. The Audio Status In _0 register contains data clocked in during the last input left sample time and the Audio Status In_1 register contains data clocked in during the right sample time except for frame formats Frform 0 and 1. The data's external format is identical to an input sample specified by the FrForm field of the audio control register (Table 7-5).
2. If the number of input sample data bits is less than 32, then low order zeros are appended to the value stored in the register. Frform 0 clocks the status data from each sample into both the Audio Status In_0 register and the Audio Status In _1 register. Frform 1 clocks in 16 bits of status data from each audio frame.
3. Status data from processor pin AudioInput[0] is loaded into the most significant 16-bits of the Audio Status In_0 register.
4. Status data from processor pin AudioInput[1] is loaded into the least significant 16-bits of the Audio Status In_0 register.
5. Status data from processor pin AudioInput[2] is loaded into the most significant 16-bits of the Audio Status In_1 register.
6. Status data from processor pin AudioInput[3] is loaded into the least significant 16-bits of the Audio Status In_1 register.
7. Data is clocked even if the input DMA is disabled, as long as InFsync and SClk are active. These registers are typically used to capture AES/SPDIF or Codec status with an external PAL, or directly from the CS4216 or AD1847.

7.3.3 Audio Status Out Register

The Audio Status Out (0xFC0B10) is a 32-bit register, memory-mapped to its Control Bus address. This register contains data that is clocked out on the processor pin StatusOut during each input sample time except for frame format Frform 1. This register operates as follows:

1. Frform 1 clocks out 16-bits of data to the processor pins AudioOut [0..3] during each audio frame. Data is clocked only on frames with active OutFsync. The data's external format is identical to an input sample specified by the FrForm field of the audio control register.
2. If the number of sample data bits is less than 32, the low order bits of the register are not transmitted. This register is typically used to control AES/SPDIF or Codec status with an external PAL or directly to the CS4216 and AD1847.

7.4 Audio Block Format

When an audio block is captured and transferred to the host, it may contain a header every block of N samples if the insert header bit is set in the audio configuration command. The block size limits vary with the sample rate. Table 7-6 shows the format of this header.

Table 7-6 Audio Block Format

Field	Length	Value
Magic Number	32 bits	0xDEADCAFE
PTS	64 bits	Left justified
Reserved	8 bits	0
Number of Channels	8 bits	
Sample Size	8 bits	Value is 16 or 32
IOvr	8 bits	■ 0: normal ■ 1: DMA setup overrun on input
Reserved	32 bits	0
Block Size(N)	32 bits	in bytes
Audio Data	8N bits	Pure sample data with channel samples interleaved.

Index