SiliconGraphics
Computer Systems

MACE I/O ASIC Specification

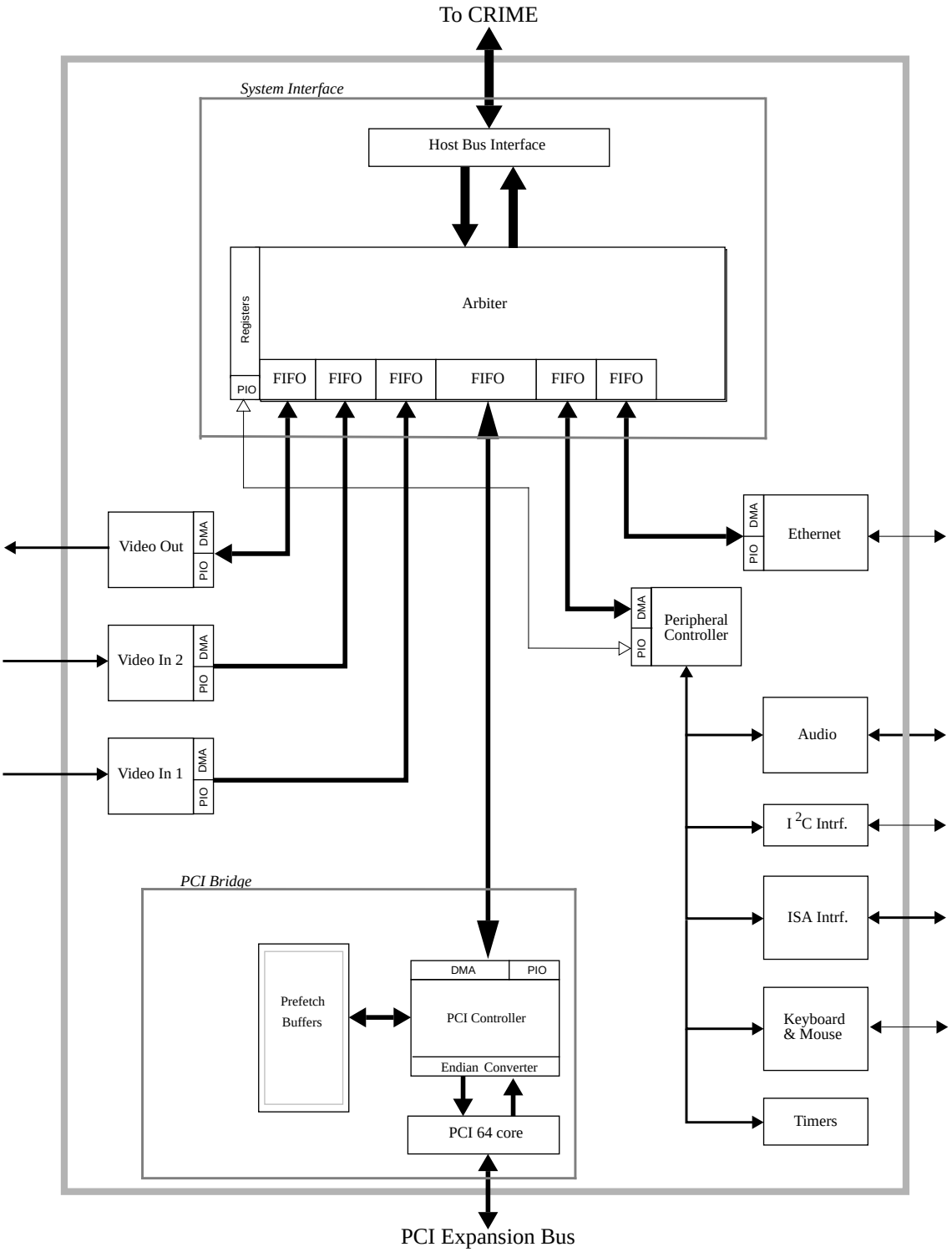
Chip Revision 2.0

May 31, 1996

Bill Dunham, Peter Baran, Tim Tucker
George Apostol, Rick Parada, Leslie Neft, Madhsudan Hans
David Jarosh, Kamram Izadi
John Maneatis, and Mike Travis

1 Introduction

The *Moosehead* system I/O asic forms the heart of the I/O subsystem. It contains all of the basic I/O interfaces including: keyboard & mouse, interval timers, serial, parallel, i^2c, audio, video in & out, and fast ethernet. The I/O asic also contains an interface to an external 64-bit PCI expansion bus that supports five masters (two SCSI controllers and three expansion slots). A block diagram of the entire I/O asic is shown below:



1.1 Main Features:

- A 64-bit PCI expansion bus with support for five external masters (cards or motherboard devices)
  - Two programmable arbitration levels, round-robin or fixed priority for external bus masters
  - Eight PCI external interrupts for connection to slots and motherboard devices
  - Two kilobytes of read ahead buffering for read cache line and read multiple PCI commands
  - Selectable byte swapper for bus master traffic to and from CRIME memory
- Two independent video input channels with variable resize, filtering, and color space conversion
  - Connects to NTSC/PAL video decoder, D1 digital video, or camera input sources
- One video output channel with color space conversion
  - Connects to NTSC/PAL video encoder or D1 digital video output
  - Internal loopback switch to either of the two video input channels
- Stereo audio TDM interface that supports one external multi-media codec
  - One stereo input channel and two stereo output channel
  - Independent rate DMA ring buffers for each stereo channel
  - Each stereo pairs sample rate clock can be locked to any one of three external sources
- Switchable 100Mbit/10Mbit Fast-Ethernet interface
  - MII interface for connection to external T2/T4/TX 110Mbit transceivers
  - CSMA-CD or packet switched full-duplex operation
- An 8-bit ISA bus with parallel, serial, RTC, and Flash-ROM
  - EPP/ECP-1284 Parallel interface
  - Dual 16C550C Serial ports with 16 byte FIFOs
    - ✔ DMA support for serial data rates up to 460.8 kilobaud
    - ✔ Hardware support for CTS/RTS hardware flow control in T.I. ACE 16550
    - ✔ HP IR link support integrated into each serial port with independant serial in
    - ✔ Per serial port clock prescaler selects 7.33Mhz or 4.00Mhz baud rate clock base
  - Five dma channels for the parallel, and serial interfaces
  - Dallas DS1687 Battery backed up Calendar clock with system serial number
  - Atmel 512Kx8 Flash-ROM
- Integrated PS/2 keyboard & mouse serial ports
- Integrated I^2C bus interface
  - Supports both 100K baud and 400K baud data rates
- Integrated 32-bit time base and four interval timers
  - PCI bus referenced 960 nanosecond resolution for time base
  - Three 32-bit compare registers for interval timer interrupt generation

1.2 DMA Ring Buffers

All of the devices on the peripheral controller inside the I/O asic use a common DMA interface that is based on ring buffers. Each device or data stream has a private ring buffer associated with it. DMA for a device is done by reading or writing data from the ring buffer data stored in CRIME main memory.

The memory based ring buffers each have a base address, a read pointer, a write pointer, and a size. The ring buffer base and size remain constant once the buffer is setup while the read and write pointers move around the ring as data is added or removed. The read and write pointers are indexes (or offsets) into the ring buffer relative to the start address. A picture of a ring buffer is shown below:

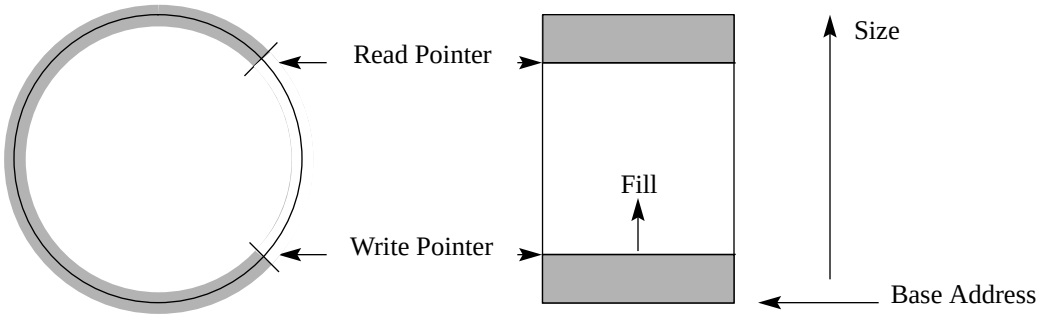


Figure: Ring Buffer Layout, Partially Filled in Wrapped Condition

The memory based ring buffers all use a common control register format with slight variations. Each ring buffer has four registers: a base & size register, a read pointer register, a write pointer register, and a configuration & status register. The base & size register point to the start address in CRIME memory where the ring buffer is stored. The read and write pointers are indexes (or offsets) into the ring buffer relative to the start address.

The read and write pointers automatically wrap around the ring because the ring size is really a mask that is applied to the pointers each time they are used. To construct an address in the ring, the following equation is used:

Address[31:0] = BaseAddress[31:N] | (PointerOffset[M:O] & (SizeMask[P:Q] | 1[Q-1:O]))

Figure: Ring Buffer Address Calculation

The constants N, M, O, P, and Q vary from device to device to some degree, but the basic principle is the same for all of the ring buffer DMA channels. The read and write pointers are masked and then logic or'd with the ring buffer base address to generate an address in main memory. Since the ring size is controlled by a mask, the rings are restricted to power of two sizes (i.e. 2, 4, 8, 16, 32, ...).

The last register of the four control registers, the configuration and status register, contains flags to enable the DMA channel and set the interrupt threshold and check the status of the ring. It also contains a field which represents the number of elements in the ring buffer from the hardwares point of view. This field is updated by the hardware DMA engine as data is added to or removed from the ring buffer by moving the read and write pointers.

The normal mode of operation for the ring buffer read and write pointers is for the DMA engine to control one and for software to control the other. As an example, for a DMA input channel the hardware would control updates to the write pointer and software would control updates to the read pointer. As data is placed into the ring by the hardware, the write pointer would be advanced and the count of items in the ring would increment. When the software reads items from the ring it updates the read pointer to tell the hardware how many it read (up to all of them). This implies that the hardware quickly recalculates the count of items in the ring after software updates it's read pointer.

1.2.1 Number of Elements

The number of elements in the ring buffer at any given time is based on the relative distance between the write pointer and the read pointer. When the write pointer offset and the read pointer offset are equal the ring buffer is empty and the number of elements in the ring is zero. A ring buffer by definition has one ambiguous point, that is, what does it

mean to say that the ring buffer is full? This is caused by the fact that the write pointer offset and the read pointer offset being equal could also mean that the ring buffer is full. To avoid this problem, the rule for the ring buffers used here is that the ring is full when the write pointer offset is one less than the read pointer offset. This makes calculating the number of elements in the ring trivial since no context is involved:

Number = (WritePointer - ReadPointer) & (SizeMask[P:Q] | 1[Q-1:0])

Figure: Number of Elements in Ring Buffer

1.2.2 Interrupts

The ring buffers provide a single FIFO threshold like interrupt. The interrupt is based on the number of items currently in the ring buffer and the interrupt condition selected. The ring buffer DMA engines provide four basic ring interrupt selections: none, empty, full, and levels. The first interrupt option, none, just disables the interrupt output and keeps the signal in the inactive state. Note that the interrupt selections all observe the ring size mask value.

The second and third interrupt options are generated using simple all zero and all ones detectors on the current ring count of items. The empty condition is true if all bits of the count are zero, while the full condition is true if all bits of the item count are ones (note that the full condition is true when the ring item count is N - 1 which observes the rule set above in section 1.2.1). The complement of these two conditions are also available: not empty and not full. Both of these tests use the size mask to force unused item count bits to zero or one.

The fourth option allows the system software to select among three possible levels 25%, 50%, and 75%. The 50% level condition is generated by looking at the most significant bit, based on the size mask, of the item count. When the MSB is true the ring buffer is >= 50% full, while when the MSB is false the ring buffer is < 50% full. The 25% and 75% level conditions are generated by looking at the two most significant bits, based on the size mask, of the current item count. When the two MSBs are zero the ring buffer is < 25% full, while when the two MSBs are true the ring buffer is >= 75% full. The complements of these two conditions generate the >= 25% and < 75% selections.

1.3 Memory Utilization

The memory interface provided by the CRIME asic gives the best performance when accesses are multiples of 64-bit quantities. Because of this, all of the DMA ring buffer engines within the I/O asic perform read and write operations using blocks of 64-bit words. For those devices where this is not a natural block size, such as serial and parallel for example, packet formats have been introduced so that the DMA engine can perform full 64-bit memory operations.

2.0 Video Input & Output

2.1 Overview

2.1.1 Introduction

This document describes the video portion of the Moosehead MACE chip.This includes the video input and output channels, filtering and colorspace conversion and the DMA engine.

2.1.2 Features

Video Input Channels

- 4 video input D1 sources (2 general purpose D1 sources, analog decoder, digital camera)
- 2 independent input channels supporting 4:2:2 YUV (from one or two input ports) or 1 channel supporting 4:2:2 YUV and 1 channel supporting Alpha.
- Clipping

Video Input Filtering and Scaling

- Conversion from non-square pixel format to square PAL or NTSC
- Down-scaling of image to any size
- Mipmap generation
- Color space conversion to RGB with optional dithering

Video Output Channels

- 3 video output D1 ports (2 general purpose D1 and an analog encoder)
- 1 output channel supporting either 4:2:2 YUV (to one output port) or 4:2:2:4 YUVA (to two combined output ports)
- Padding

Video Output Filtering

- Color space conversion
- Conversion from square PAL or NTSC pixel formats to non-square
- Notch filter

Video DMA

- Linear or tiled (CRIME Compatible) buffer formats
- Field or interleaved frame modes

Pixel Formats (big endian only)

- 32-bit RGBA
- 16-bit RGBA
- 4:2:2 YUV (8-bit or 10 bit)

Other features

- Time Stamp with Field/Frame Counter
- Genlock output to video input port

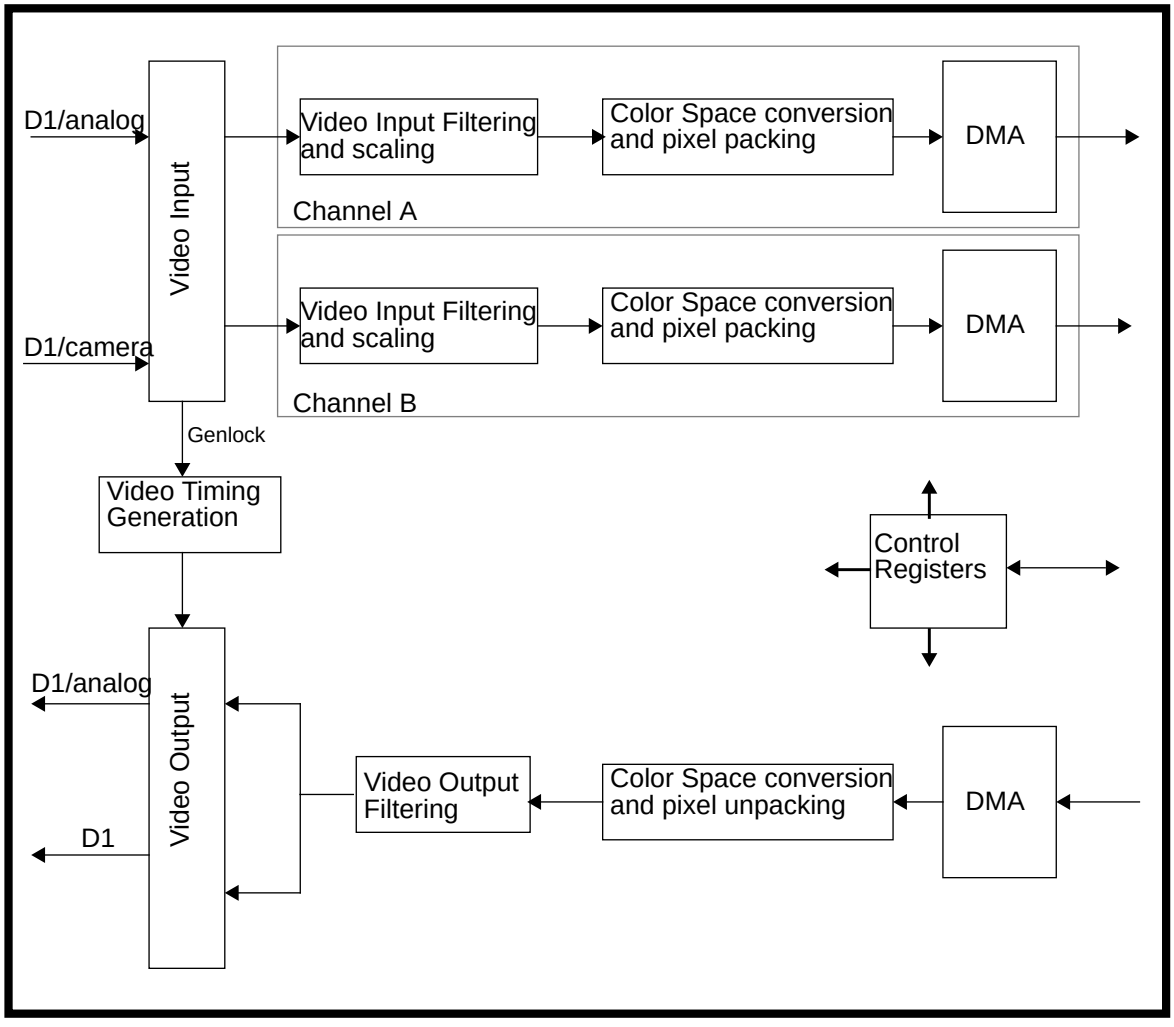
- Diagnostic Loop-back modes

2.1.3 Issues

1. How to handle VITC/Closed Captioned data?
2. Detecting loss of video sync?
3. Detecting loss of video clock?
4. Synchronizing changes to video filter parameters on field boundaries?
5. Pause/resume of video stream?
6. Handling VINO/Galileo compatible AGBR pixels?
7. Input field size for mipmapped restricted to be at least 512 pixels wide?

2.1.4 Video Overview

FIGURE 1. MACE Video Block Diagram



2.1.4.1 Video Input

There are 2 input channels each of which can be programmed to receive video data from one of 4 possible sources: an analog video decoder input, an 8-bit D1 input from a digital camera, or 1 of 2 general purpose D1 input ports (8 or 10 bit). These sources are multiplexed outside the chip so that there are effectively only 2 input ports on the chip. The primary port can be either the analog decoder or a general purpose D1, and the secondary port can be either the digital camera or a general purpose D1. The input channels can be configured as 2 independent channels of YUV 4:2:2 or can be used together to receive 1 channel of 4:2:2 YUV and 1 channel of Alpha. The merging of the Alpha with the data stream is done by software.

Clipping registers for both horizontal and vertical for each channel determine which portion of the video input field will be used as video data. This allows data to be sent with or without the ancillary data period, and also allows arbitrary clipping of the video input to any size field.

2.1.4.2 Input Filtering and Scaling

Each of the following functions may be performed in the order in which they are described below. All functions are optional. All filtering is done in YUV color space with sub-sampled UV.

2.1.4.2.1 Non-square to square pixel conversion

This filter will convert from non-square CCIR 601 pixels to square NTSC or PAL. There are 2 fixed scaling factors which can be selected. For NTSC the conversion will scale down by 10/11 (or 654/720). For PAL the conversion will scale up by 12/11 (or 786/720). This filter can also scale down by 1/2 horizontally. The scaling is implemented with a Mitchell filter, which provides the highest quality conversion that is feasible for this chip.

2.1.4.2.2 Arbitrary Down-scaling

The incoming fields can be scaled in both the vertical and horizontal by any ratio less than 1. The horizontal and vertical scaling are independent and can use different scaling factors. The down-scaler uses a block filter which is considered to be somewhat low quality. This essentially calculates output pixels based on the weighted average of the input pixels.

2.1.4.2.3 Mipmap Generation

The chip can generate mipmaps real-time from incoming fields of video. The mipmap function operates on a starting field size of either 512x256 or 512x128, which will be the size of the first mipmap. Subsequent mipmaps will be generated by repeatedly reducing the field in both directions by powers of 2. Thus, the following mipmaps will be generated:

- Starting from 512x256: 512x256, 256x128, 128x64, 64x32, 32x16, 16x8, 8x4, 4x2
- Starting from 512x128: 512x128, 256x64, 128x32, 64x16, 32x8, 16x4, 8x2, 4x1

The reduction will be done with a simple averaging method. Before mipmapping, the image must first be scaled and/or clipped to the appropriate starting size using the previous clipping/filtering/scaling functions. There is an option in the clipping block which may be used to pad the incoming image with extra blank lines in the case that the incoming field contains less lines than the desired starting mipmap size. This will add a border of black lines at the bottom of the image. This black border will be averaged in with the rest of the image as it is reduced, which may result in some loss of image quality at the edges.

Mipmapping can only be used with the 16-bit RGBA pixel format.

2.1.4.3 Input Color Space Conversion and Pixel Packing

Depending on the pixel format selected, the following operations may be performed:

- Upsampling of YUV (using simple interpolation of the missing U and V values)
- Color space conversion to RGB
- Dithering
- Packing into the appropriate format for DMA

2.1.4.3.1 Pixel Formats

The following pixel formats are supported:

- 32-bit RGBA and ABGR
- 16-bit RGBA (5551 with dither)
- 4:2:2 YUV, both 8-bit and 10-bit

2.1.4.4 Video Output

There is 1 video output DMA stream which can be either 4:2:2 YUV going to one output port or 4:2:2:4 YUV going to 2 combined output ports. The 2 D1 output ports supply 3 destinations: an analog encoder and 2 general purpose D1 interfaces (8 or 10 bit). The analog encoder interface will share the data portion of one of the D1 interfaces but uses additional video timing signals from the Mace chip.

2.1.4.5 Output Color Space Conversion and Pixel Unpacking

The video output path supports the same pixel formats as the input path. Depending on the pixel format selected, the following operations may be performed:

- Unpacking data
- Notch filter
- Color space conversion to YUV
- Down sampling of UV (using a 1/4 1/2 1/4 FIR filter)

The notch filter is a simple 1/2 0 1/2 FIR filter. It is opitional and can be useful when the video data has previously been dithered. If the DMA data is an RGBA format, the notch filter will be applied to the Y,U and V components after color space conversion, but before subsampling. If the DMA data is YUV format, then the notch filter will only be applied to the Y component. The notch filter is not applied to alpha.

2.1.4.5.1 Output Filtering - Square to Non-square Conversion Pixel Conversion

This filter will convert from square NTSC or PAL pixels to non-square CCIR 601. There are 2 fixed scaling factors which can be selected. For NTSC the conversion will scale up by 11/10 (or 720/654). For PAL the conversion will scale up by 11/12 (or 720/786). This filter can also scale up by 2 horizontally.The scaling is implemented with a Mitchell filter, which provides the highest quality conversion that is feasible for this chip.

All functions are optional. All filtering is done in YUV color space with sub-sampled UV. If output data is 10-bit D1, any filtering will result in an immediate truncation of the lower 2 bits of precision. Filtering will also be done on the alpha component.

2.2 Architecture

2.2.1 Video Input

In this document, video input port refers to the physical ports into the MACE chip. Video input processing channel refers to clipping, filtering and color space conversion internal to the chip (as differentiated from the DMA channel).

2.2.1.1 Video Input Ports

There are 2 video input ports to the video section of the MACE chip. Each of these ports may come from 2 sources which are multiplexed outside the chip:

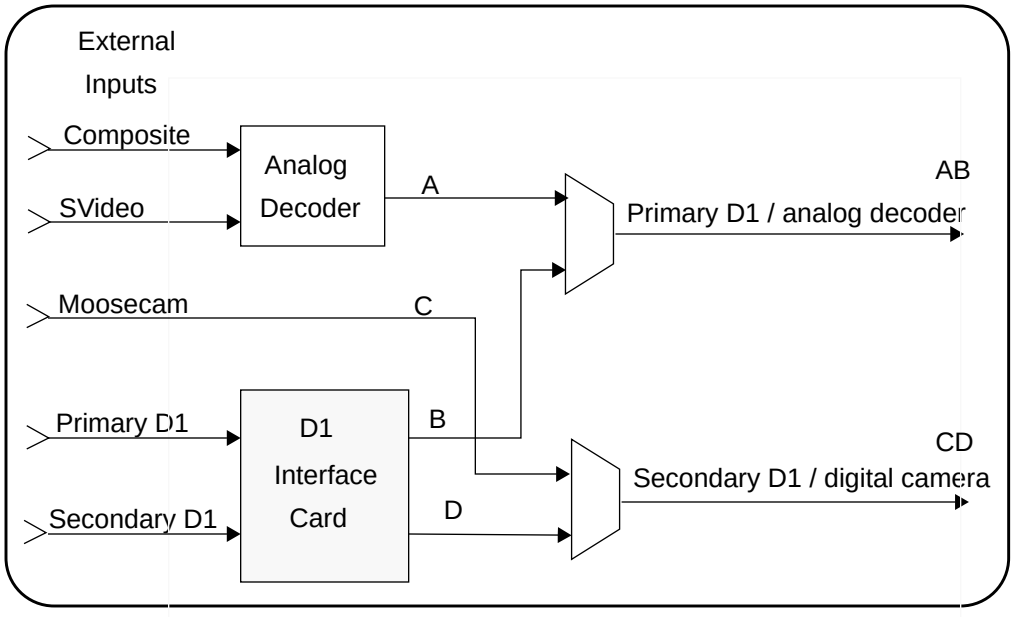
- Primary D1 from an analog video decoder or general purpose D1
- Secondary D1 from a digital camera (moosecam) or general purpose D1

The following notations are used to refer to the video input ports:

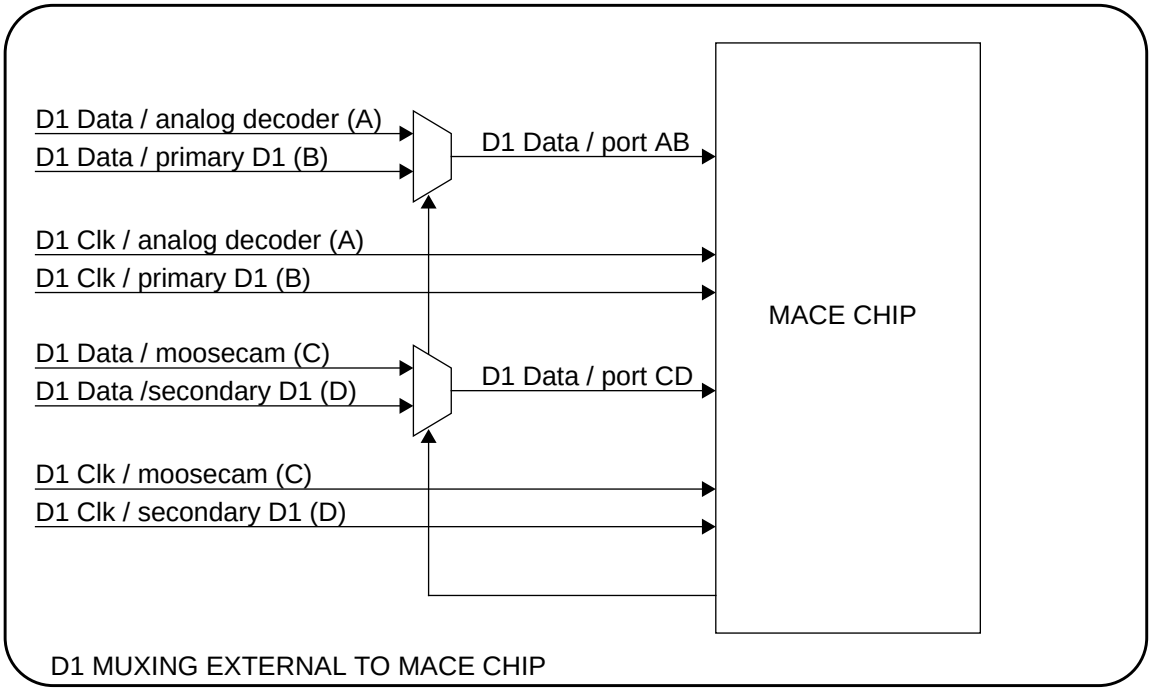
Port A = Analog Decoder
Port B = Primary D1
Port C = Moosecam
Port D = Secondary D1

Thus, internal to the Mace chip there are effectively 2 input sources refered to as AB and CD.

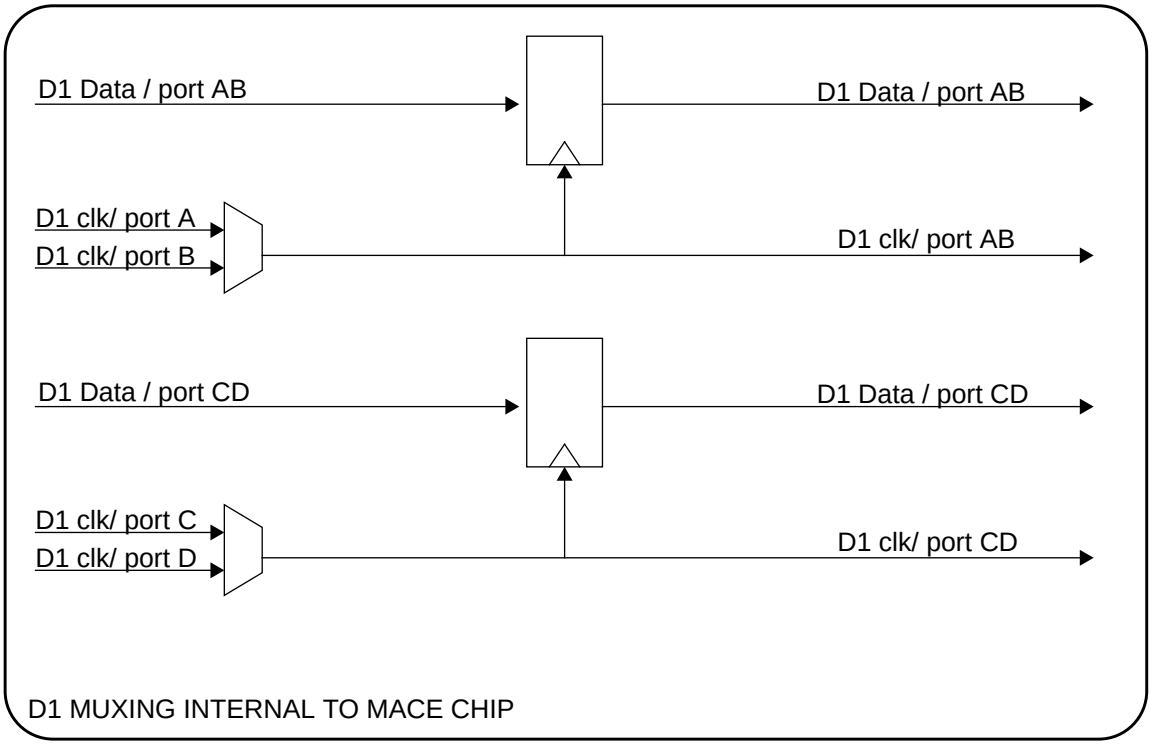
FIGURE 2. External Video Input Selectors



The analog inputs are selected via I^2C commands, and the input multiplexor is selected with the Video Hardware Configuration Register.



D1 MUXING EXTERNAL TO MACE CHIP

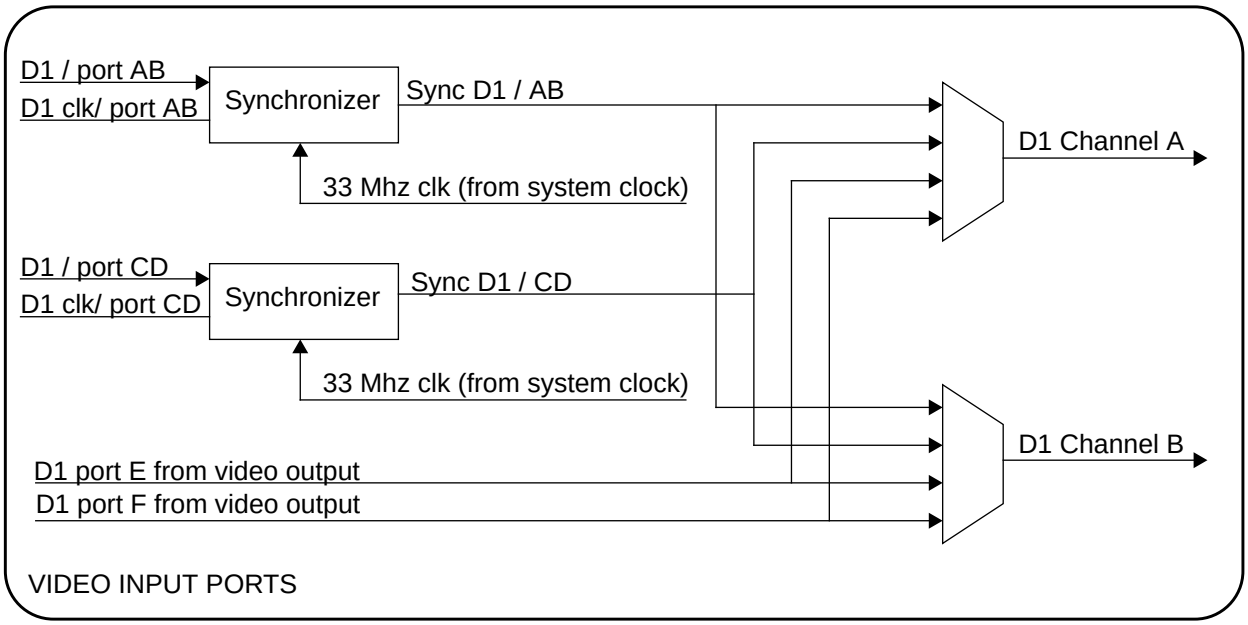


D1 MUXING INTERNAL TO MACE CHIP

Additionally, the chip's D1 output stream may be fed back into the video input path as shown in the following figure.

For input in YUV 4:2:2, either of the 2 chip input ports, or either of the output streams, may be selected as the input for each video input processing/DMA channels. If desired, the same input port may be selected for both channels. This would be useful to capture full size video while displaying a reduced size image on the monitor to view the image as it is being captured. The user may also wish to receive YUV on one channel and Alpha data on the other channel, however the data from these 2 streams will be DMA'ed separately and must be merged by software.

FIGURE 3. Video Input Ports



2.2.2 Video Output

There is only one video output DMA channel which can process one YUV data stream plus Alpha (if an RGBA DMA pixel format is used).

2.2.2.1 Video Output Ports

There are 2 video output ports from the video section of the MACE chip:

- Port E: D1 port which can go to both a general purpose D1 and to an analog encoder
- Port F: General purpose D1

Each ouput port can be programmed to drive data from 1 of 3 sources: the YUV data stream, the Alpha data stream or the selected D1 input port. The latter case, referred to as passthru mode, allows the user to monitor data on an external monitor as it is being captured via one of the input ports. This mode may also be useful for diagnostics. When passthru mode is selected, the D1 input port (AB or CD) which will be used is that which has been selected as the genlock source.

Both output ports will generate standard 10-bit D1 (compatible with 8-bit D1) with embedded control information. In order to accomodate the analog encoder, seperate control signals for blanking and field will be driven on Port E.

2.2.2.2 Video Output Timing Generation and Genlock

Video output timing can be genlocked to either D1 input port or an external sync source which provides the requried timing signals (h,v,f and clock) to the Mace chip. In order to be genlocked, the video output timing must be programmed to match the timing of the genlock source. A programmable register allows the output to be delayed (in the hardware) from the input by as much as 1 line. Any additional adjustment between the genlock source and the output must be handled in software via the video timing (Hpad/Vpad) registers.

Output may also be generated with no genlock source. In this case, the output will use a fixed 27 Mhz clock (corresponding to a 13.5 Mhz pixel clock) as it's timing source.

NOTE: THE MAXIMUM FREQUENCY AT WHICH THE VIDEO OUTPUT CAN RUN IS 30 MHZ.

Although the diagram below shows a certain amount of flexibility as to what video is driven to the 2 output ports, it is important to note that both video output ports must run with the same video timing.

VIDEO OUTPUT TIMING GENERATION AND GENLOCK

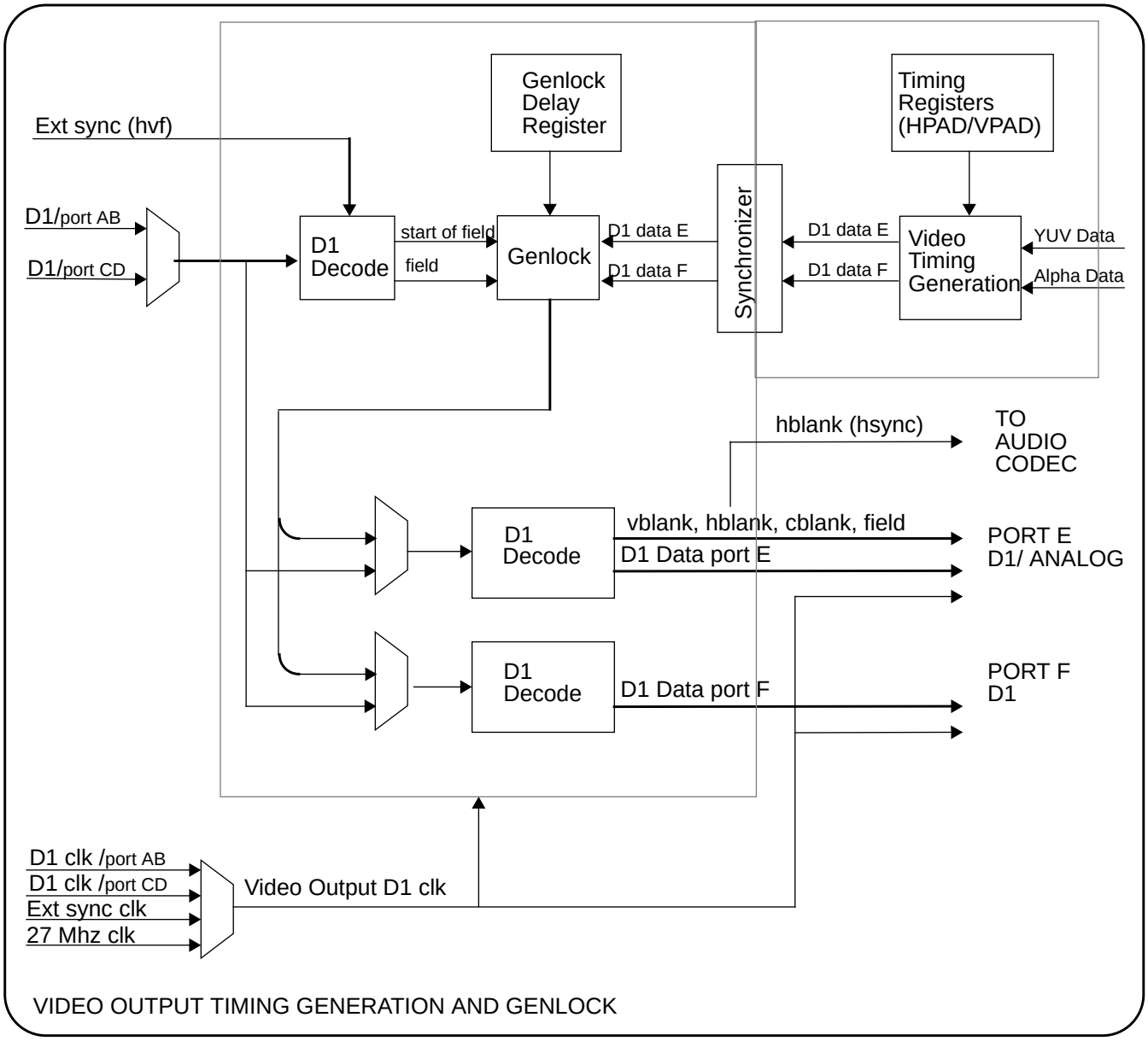
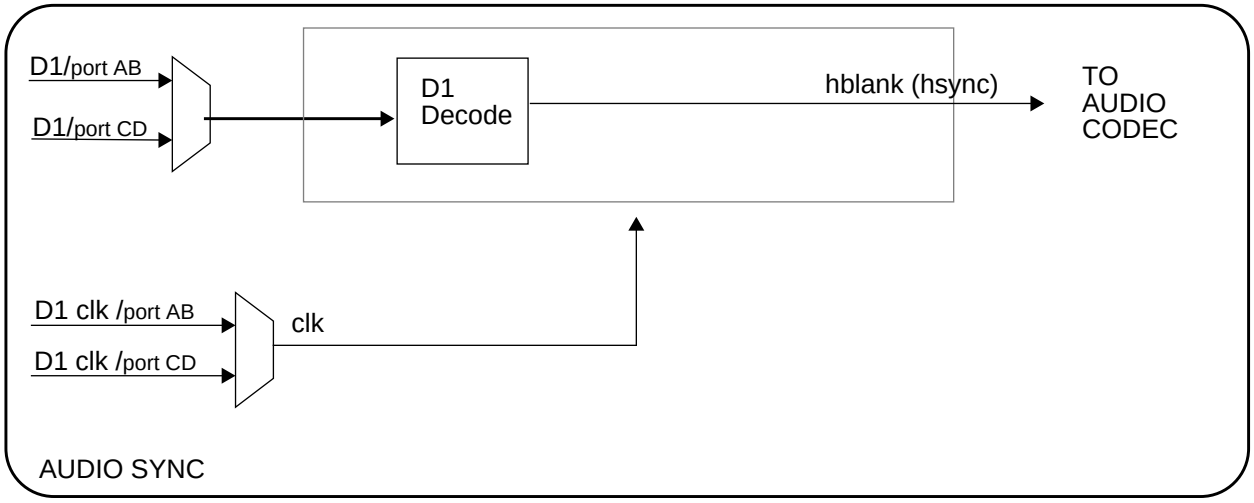


FIGURE 4.

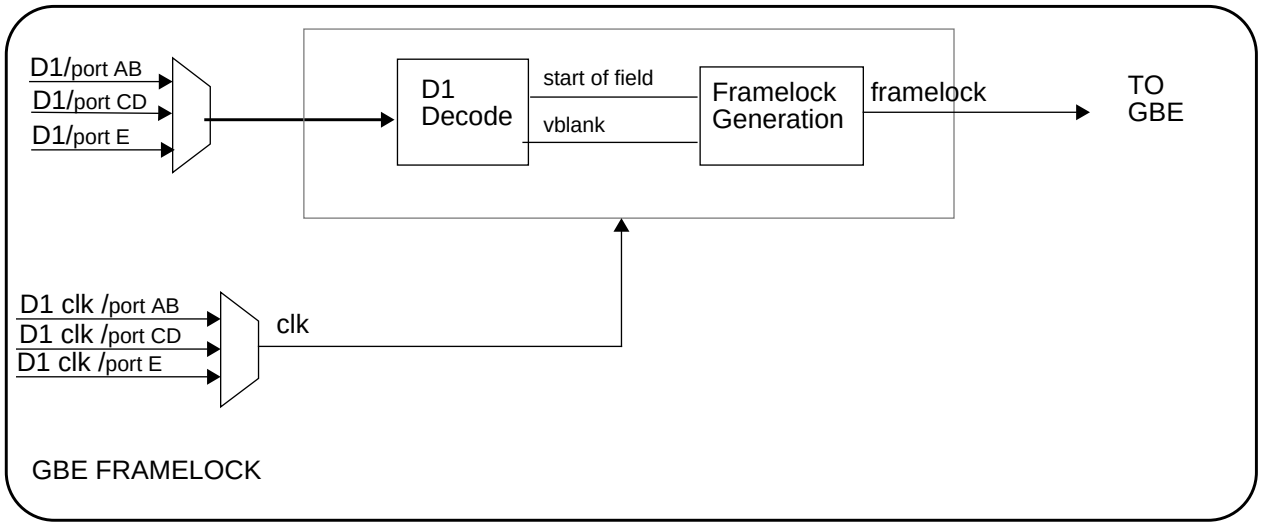
2.2.2.2.1 Audio Sync

The MACE chip provides 2 sync signals to the audio codec. The first sync, shown below, is programmatically selected from either of the 2 D1 inputs (AB or CD). The other sync (shown above) will always come from the video output. The frequency of the sync signal is the same as the H bit encoded in the corresponding D1 stream.



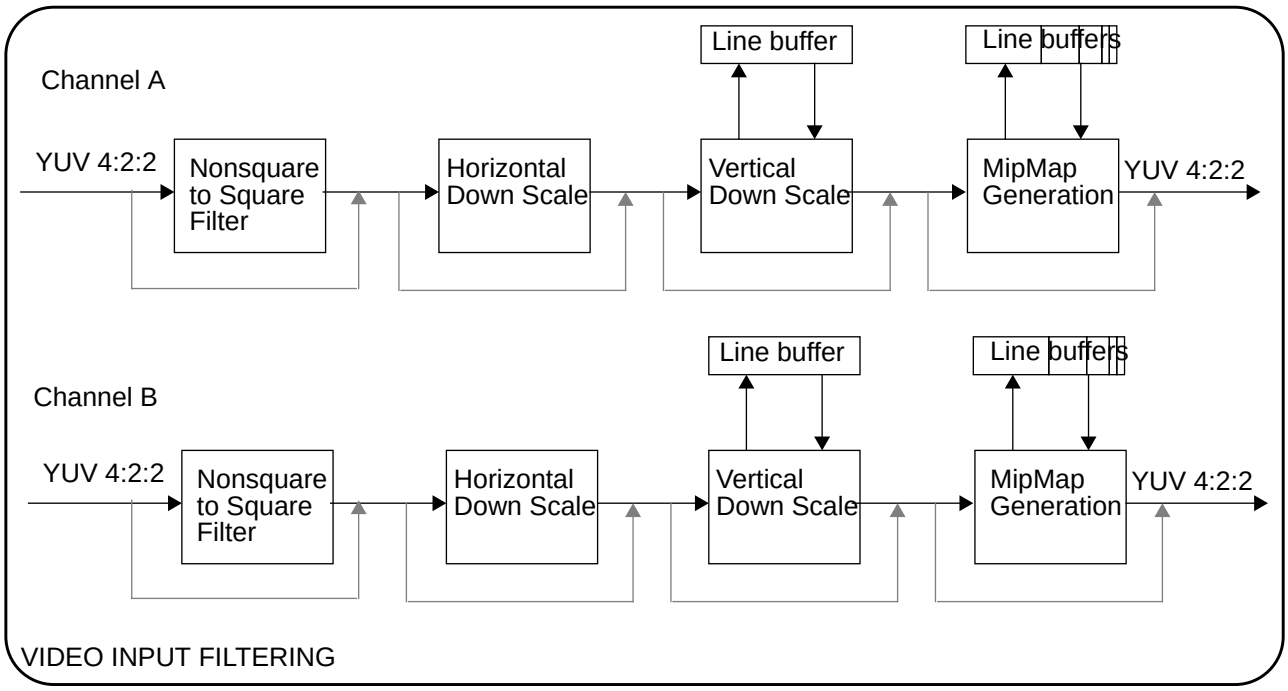
2.2.2.2.2 GBE Framelock

A framelock signal is provided to the GBE chip. It can be programatically selected from either D1 input or the D1 output. The frequency of the framelock signal will be the same as the frequency of start-of-field in the corresponding D1 stream.



2.2.2.3 Video Input Filtering

FIGURE 5. VIDEO INPUT FILTERING



2.2.2.3.1 Non-square <=> Square Pixel Conversion

The native mode for the MACE chip video input and output is non-square pixel which is 720 pixels horizontally for either PAL or NTSC. For applications which will use square pixels internally, a high quality Mitchell filter is used to convert from non-square to square on input and from square to non-square on output. These conversions are done with fixed ratios which are determined by the CCIR 601 and PAL and NTSC standards. The following table shows the conversions which can be performed.

TABLE 1. Non-square <=> Square Conversions

Video Format	Input Non-square to square	Output Square to non-square
NTSC	720 to 654 10/11	654 to 720 11/10
PAL	720 to 786 12/11	786 to 720 11/12

The Mitchell filter can also be used to scale horizontally on input by 1/2 and on output by 2.

The same scaling ratios will be applied, regardless of the size to which the incoming image is clipped.

2.2.2.3.2 Horizontal and Vertical Down Scaling

Arbitrary horizontal and vertical down scaling is done with a simple block filter which effectively calculates the output as a weighted average of the input. An offset at the start of the line will be used, such that the first coefficient used will be 1/2 of the specified scaling ratio. This will slightly improve the quality of the filter

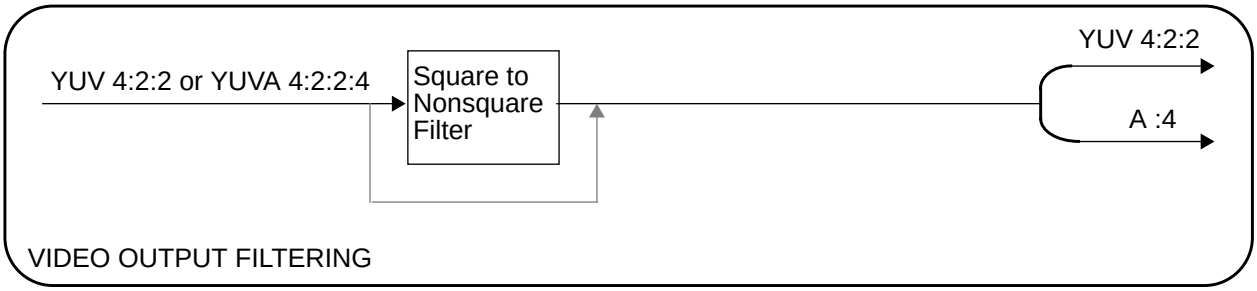
for some ratios (ie, 1/2, 1/3, 1/4, etc). An example of the coefficients applied when scaling by 1/2 is shown below:

TABLE 2. Scale by 1/2

input pixel/line => **output pixel/line ||**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
0	1/4	1/2	1/4						
1			1/4	1/2	1/4				
2					1/4	1/2	1/4		
3							1/4	1/2	1/4

2.2.2.4 Video Output Filtering

FIGURE 6. VIDEO OUTPUT FILTERING



2.2.2.5 Video Input /Output Color Space Conversion

FIGURE 7. VIDEO INPUT COLOR SPACE CONVERSION

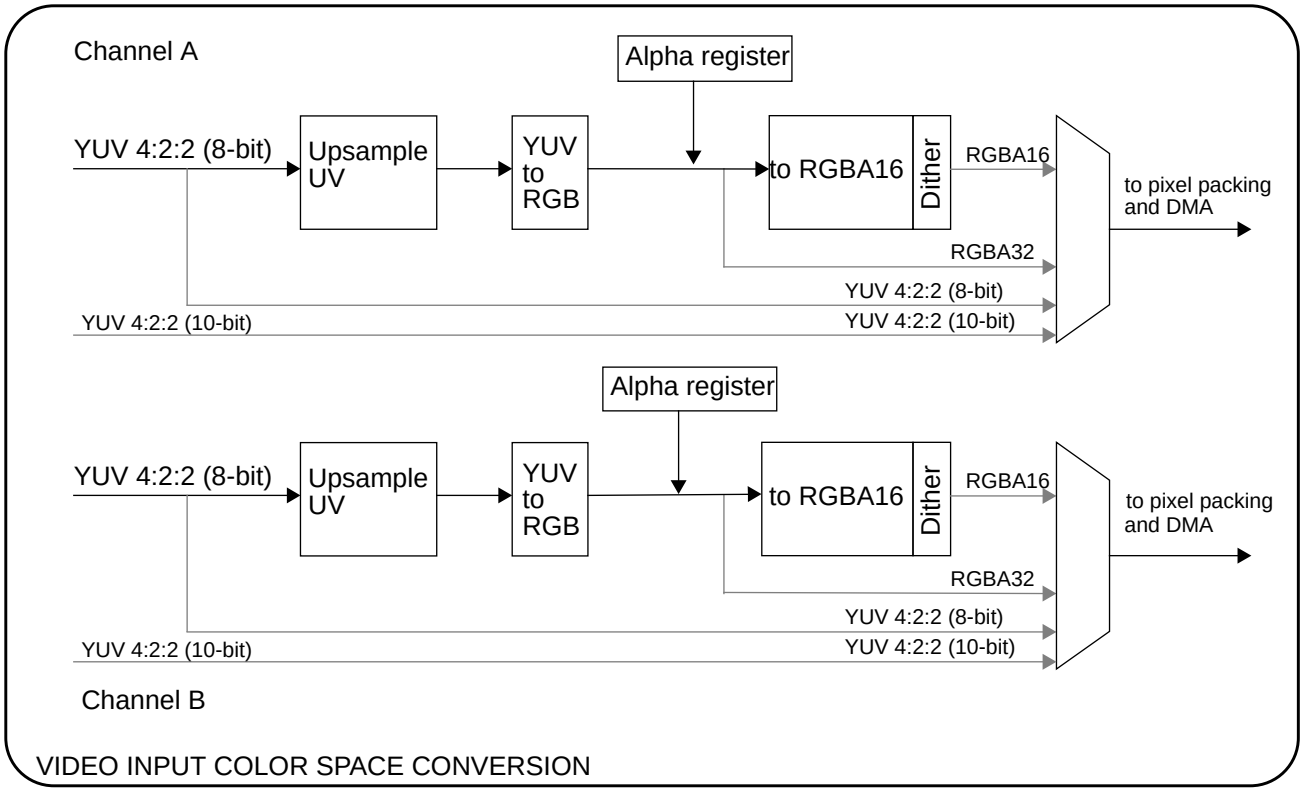
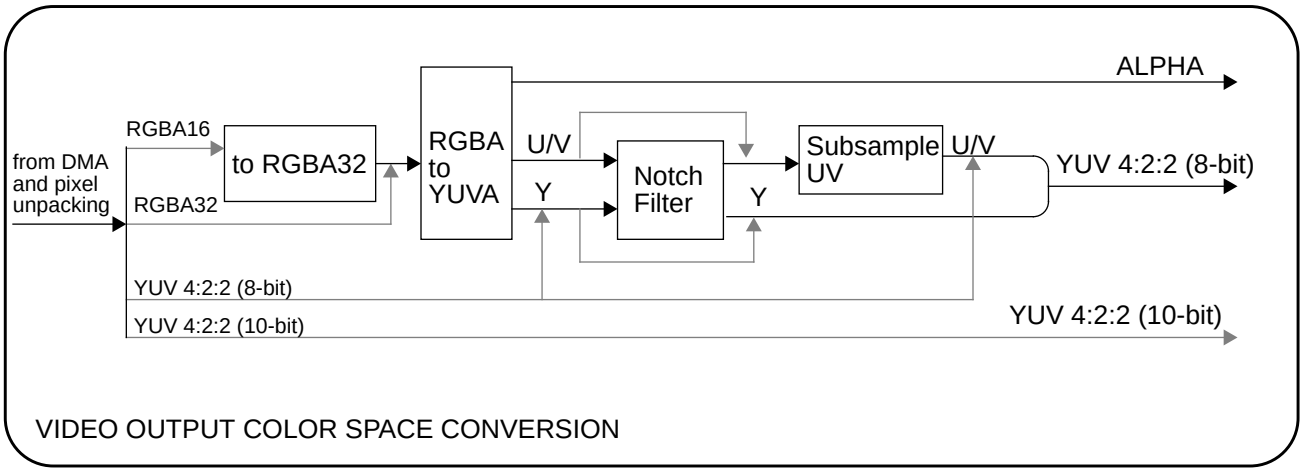


FIGURE 8. VIDEO OUTPUT COLOR SPACE CONVERSION



2.2.2.5.1 Sub-sampling and Up-sampling U and V

If the input pixels are converted to RGB space, the U and V components will be up-sampled prior to the conversion. The missing U and V samples are interpolated by a simple filter that calculates the simple average of the adjacent horizontal samples. The following formula is used:

$U_n = (U_{n-1} + U_{n+1}) / 2$
$V_n = (V_{n-1} + V_{n+1}) / 2$

This formula is implemented using shifts and adds as follows:

$U_n = (U_{n-1} + U_{n+1}) >> 1$
$V_n = (V_{n-1} + V_{n+1}) >> 1$

If the output pixels are being converted from RGB space, the U and V components will be sub-sampled after the conversion. The U and V samples are horizontally pre-filtered with a 1/4 1/2 1/4 FIR filter prior to decimation to suppress aliasing:

For even values of n:
$U_n = 1/4 * U_{n-1} + 1/2\ U_n + 1/4\ U_{n+1}$
$V_n = 1/4 * V_{n-1} + 1/2\ V_n + 1/4\ V_{n+1}$

This formula is implemented using shifts and adds (with rounding) as follows:

$U_n = (U_{n-1} + (U_n<<1) + U_{n+1} + 0x2) >> 2$
$V_n = (V_{n-1} + (V_n<<1) + V_{n+1} + 0x2) >> 2$

Subsampling and upsampling are not selected by the programmer, but will be performed automatically whenever required.

2.2.2.5.2 Color Space Conversion

Graphics pixels are RGB, while video pixel are YUV based, requiring conversion to allow the system to manipulate and display the video stream. For all RGB pixel formats, the YUV data is first upsampled and then converted to RGB.

Although the conversion process is theoretically perfect, in practice, errors are introduced by approximations, truncation and rounding.

2.2.2.5.2.1 RGB to YUV

The following equations are from p. 288 of Video Demystified:

Y = 0.257 * R + 0.504 * G + 0.098 * B + 16
U = -0.148 * R - 0.291 * G + 0.439 * B + 128
V = 0.439 * R - 0.368 * G - 0.071 * B + 128

This can be approximated with fixed coefficient multipliers as follow:

Y = 263/1024 * R + 516/1024 * G + 100/1024 * B + 16
U = -152/1024 * R - 298/1024 * G + 450/1024 * B + 128
V = 450/1024 * R - 377/1024 * G - 73/1024 * B + 128

The equations can be manipulated further to produce the following hardware implementation:

Y = (263 * R + 516 * G + 100 * B + 16384)/1024
U = (-152 * R - 298* G + 450 * B + 131072)/1024
V = (450 * R - 377 * G - 73 * B + 131072)/1024

Before the divide, 512 is added for rounding:

Y = (263 * R + 516 * G + 100 * B + 16896)/1024
U = (-152 * R - 298* G + 450 * B + 131584)/1024
V = (450 * R - 377 * G - 73 * B + 131584)/1024

Given RGB values in the range 0-255, these equations will produce Y values in the range 16-235 and U/V values in the range 16-240.

2.2.2.5.2.2 YUV to RGB

The following equations are from p. ??? of Video Demystified:

R = 1.164 * (Y - 16) + 1.596 * (V - 128)
G = 1.164 * (Y -16) - 0.813 * (V -128) - 0.391 * (U -128)
B = 1.164 * (Y -16) + 2.018 * (U - 128)

This can be approximated with fixed coefficient multipliers as follow:

R = 1192/1024 * (Y - 16) + 1634/1024 * (V - 128)
G = 1192/1024 * (Y -16) - 832/1024 * (V -128) - 401/1024 * (U -128)
B = 1192/1024 * (Y -16) + 2066/1024 * (U - 128)

The equations can be manipulated further to produce the following hardware implementation:

R = (1192 * Y + 1634 * V - 228224)/1024
G = (1192 * Y - 832 * V- 401 * U + 138752)/1024
B = (1192 *Y + 2066 * U - 283520)/1024

Before the divide, 512 is added for rounding:

R = (1192 * Y + 1634 * V - 227712)/1024
G = (1192 * Y - 832 * V- 401 * U + 139264)/1024
B = (1192 *Y + 2066 * U - 283008)/1024

The conversion from YUV to RGB can produce values out of range. These values are clamped to the appropriate minimum (0) or maximum (255) for the range.

2.2.2.5.3 Dithering

When video data is to be input in 16-bit RGBA (5551), dithering may be performed. An ordered dither with a matrix which minimizes introduced texture and beating, is performed on a field basis. Initially the intensity level of the pixel is scaled to evenly repartition existing values into the fewer number of bits. These bits are then modulated according to a comparison of the remainder of the scaling and the value returned by indexing into the dither matrix by the pixel position. This method has the nice feature that overflow will never occur on the resultant pixel value.

TABLE 3. Dither Matrix

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

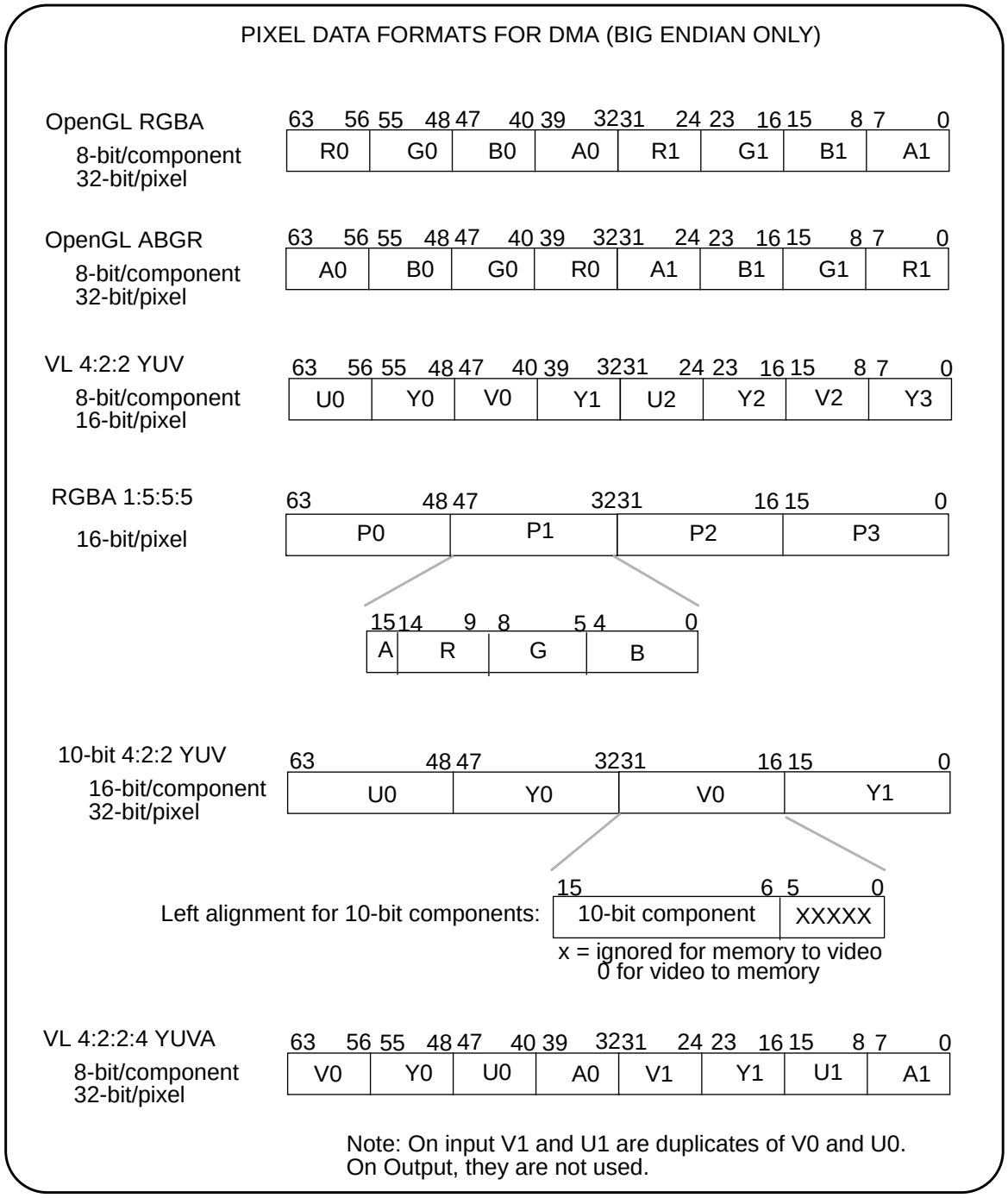
Dithering is turned on or off via a bit in a programmable register.

2.3 Programmer's Interface

2.3.1 Pixel Data Formats

All data passed into and out of the MACE video ports is expected to be in YUV 4:2:2 and can be either 8 or 10-bit resolution. The supported pixel formats are listed below. Note: The 10-bit YUV format is supported only for 10-bit D1 and does not allow any processing of the data

FIGURE 9. .PIXEL DATA FORMATS FOR DMA



2.3.2 Buffer, Capture and Page Formats

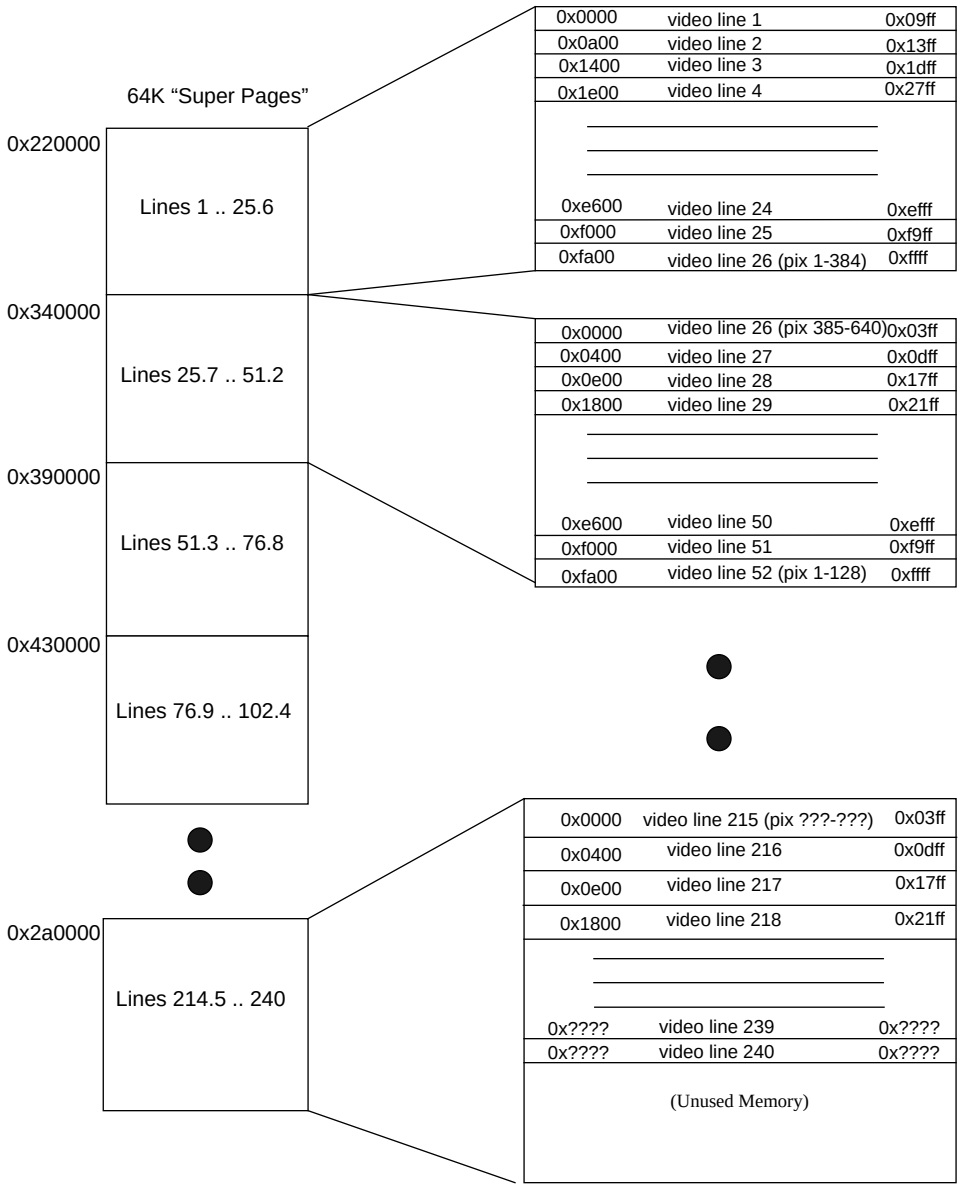
There are three forms of buffer formats: linear, tiled and mipmapped; two capture formats: fields and frames; and two forms of buffer pages: 4k pages and 64k super-pages. Here are the differences between these formats:

- Linear Video - Linear video has the characteristic that the pixels of a video lines are placed in memory in a linear fashion with possible breaks to accommodate page layouts.
- Tiled Video - Tiled video is placed in memory such that the pixels from a video line are spread across one or more “tiles” with only a certain number of pixels in each line. The tiles supported in MOOSEHEAD are 512 bytes wide thereby making the tiles 512, 256, or 128 pixels wide depending on whether the pixels are 1, 2, or 4 bytes in size. This mode is compatible with the pixel layout used by the CRIME rendering engine.
- MipMapped Video - This is a further extension of Tiled Video where in addition to the 3 sized tiles produced by Subsampled Video a 4th Tile is generated which is a combination of the remaining decimated sizes: 1/64th, 1/256th, 1/1024th, etc. Software then reassembles these pixels into the appropriate format for input to the CRIME rendering engine as a Texture Map. This mode is applicable to Video Input only.
- Field Capture Mode - In field capture, an entire field is captured in consecutive memory locations and a new buffer is used for each subsequent field.
- Frame Capture Mode - In frame capture, two fields are assembled into memory in an “interleaved” fashion so that the first field is placed in lines 1, 3, 5 and so on, and the second field is placed in lines 2, 4, 6, on up. Which field is placed first is the so-called “Dominant” field.
- 4K Pages - Traditional UNIX systems have allocated real memory in 4096 byte blocks and assigned these to addresses that span 4096 bytes of virtual address space. The actual location of these 4K pages could be totally random within the memory system of the host computer as the “contiguous” nature of the memory was achieved by assigning contiguous virtual memory addresses. In the Redwood release of IRIX (6.0), pages were increased in size to 16Kb to accommodate the larger address space of the Challenge class machines. [Note that 4K Page mode is not supported in the MACE Video DMA.]
- 64K Super Pages - To accommodate the requirement for high bandwidth from memory to graphics, the concept of 64K Super Pages were introduced. This involves allocating physically contiguous memory in 65,536 byte chunks, which is still mapped into user memory using 4K addresses (needing then 16 page translation table entries). Because this access requires fewer bits to “locate” the physical page (only the upper 16 bits), it is being designed into the hardware for other components of the Moosehead system. One usage is in the VICE memory translation table. It is designed to contain pointers to 4Mb of system memory simultaneously, therefore it only needs 64 16-bit translation table entries. Another usage of these “Super Pages” is in the GBE to Main Memory video capture path. Since it’s largest size of capture involves a PAL sized field (768x288 field x 4 byte RGBA pixel), the frame can be described by 13.5 (14) 16 bit “Super Page” descriptors.

2.3.2.1 Linear Field Video

Linear field video uses 64k “super pages” where the beginning of each video line immediately follows the previous one and the video line is split at page boundaries. The diagram below shows 32 bit RGBA pixels in a full sized NTSC (640x240) field.

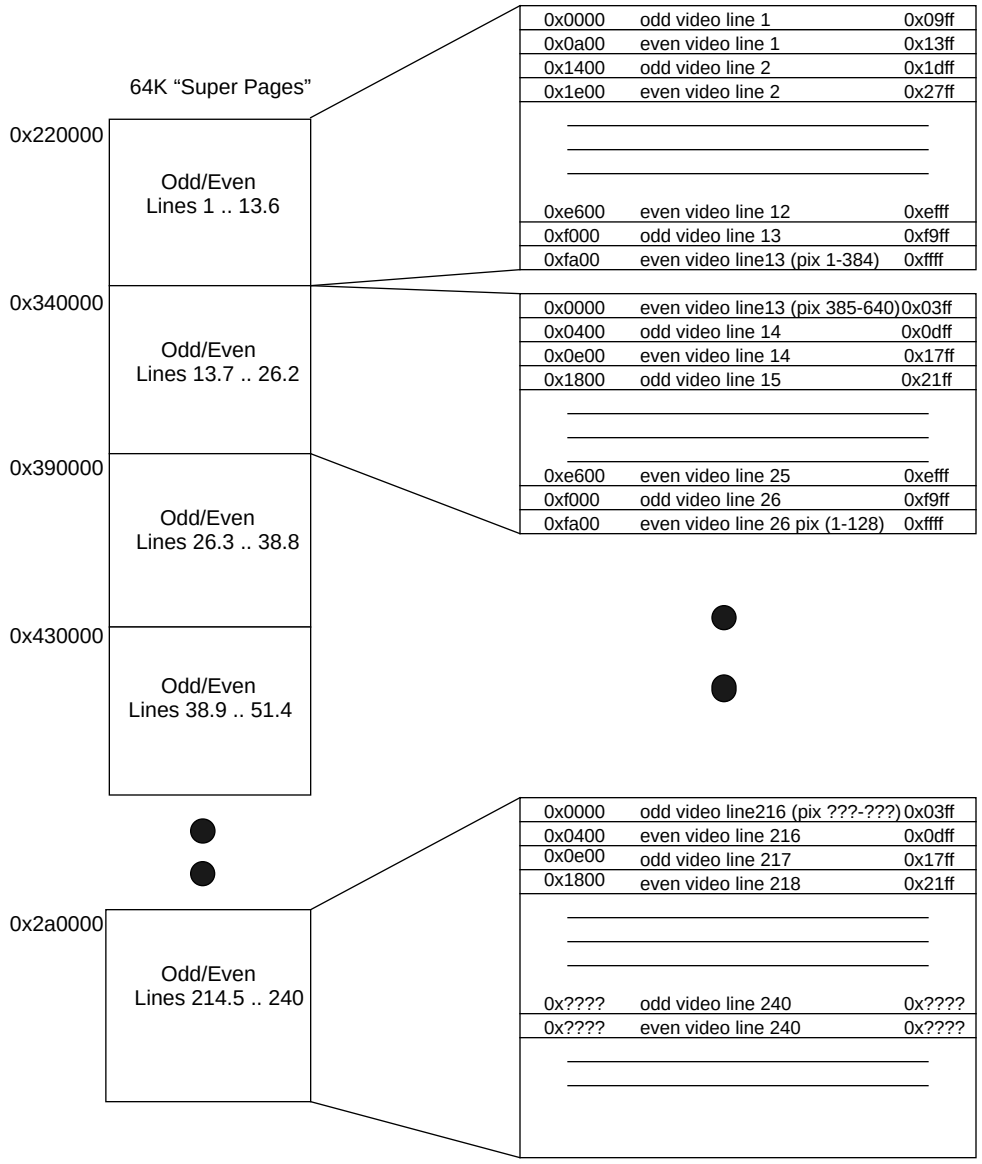
FIGURE 10. Linear Field Video



2.3.2.2 Linear Frame (Interleaved) Video

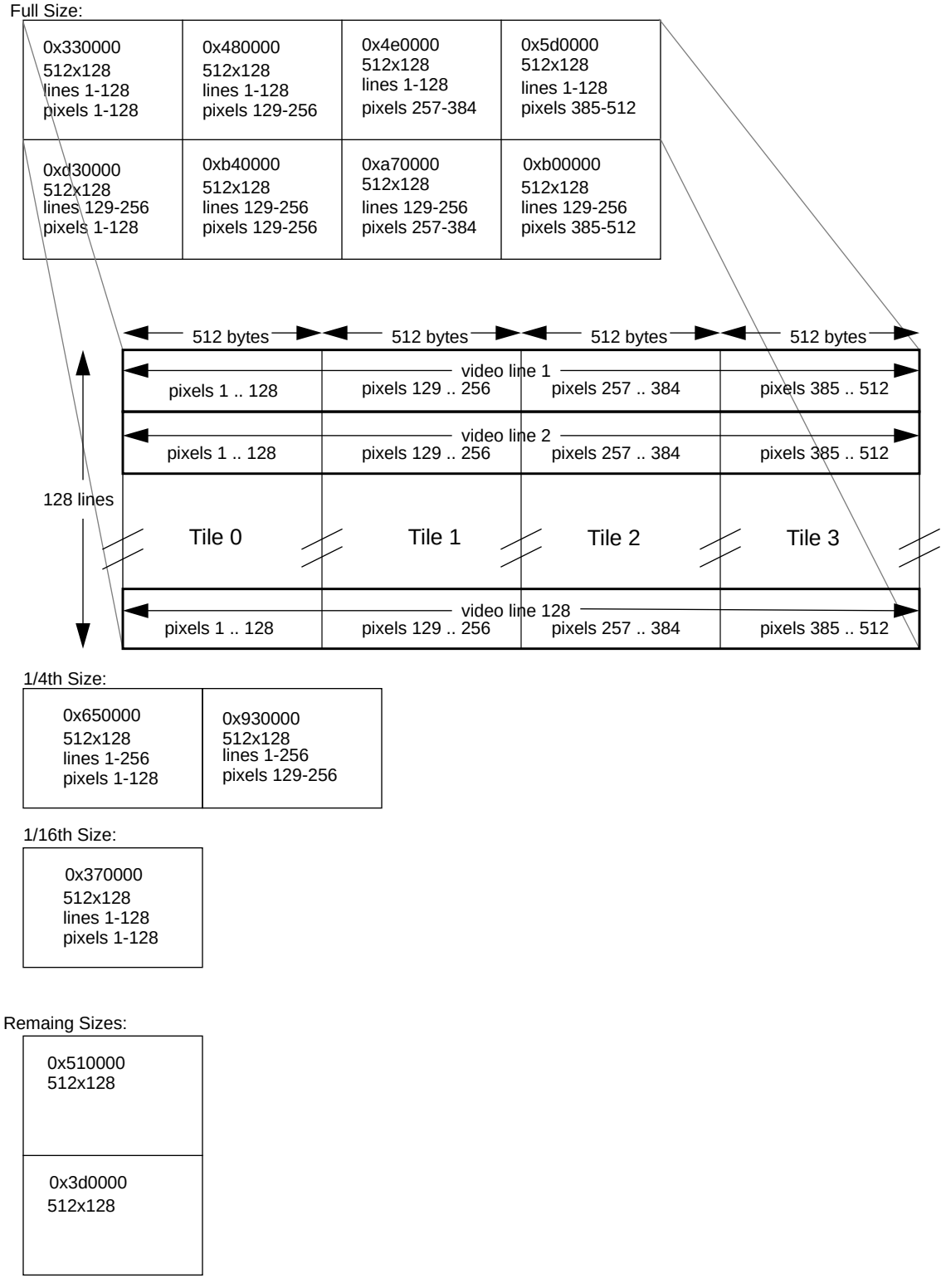
This is an example of NTSC 640x480 video frames (interleaved fields) packed as 32 bit RGBA pixels in a 64K Super Pages of memory.

FIGURE 11. Linear Frame (Interleaved) Video



2.3.2.3 Mipmapped Field Video

This example shows capturing 16 bit RGBA Pixels in fields into 64K Tiles of memory.



2.3.3 Register Descriptions

In the following register descriptions, all bits not explicitly defined are read back as 0 and all registers are defined on 64-bit aligned boundaries and are read or written using 64-bit programmed i/o operations.

Note that an approved “Name” has been suggested for each register. This name appears as part of each registers paragraph title (so that it will show up in the table of contents) and in the Table Title. Where a register is representative of a set of registers (one per channel) all register names are listed in the section paragraph while the first channel is listed in the Paragraph and Table titles. This will help both hardware and software refer to the register throughout the life of this hardware by using the same moniker.

2.3.4 Register Address Map Summary

The video registers are divided into 4 groups, one that is global for the whole device and one each for each of the video channels, two for input, one for output.

TABLE 4. Video Register Set Description

Offset	Register Set Description
0x100000	Video Input Channel (#1) Control and Status Registers
0x180000	Video Input Channel (#2) Control and Status Registers
0x200000	Video Output Channel (#3) Control and Status Registers

2.3.5 Video Channel Registers

Each video channel (2 input and 1 output) has a set of registers for control and status of that channel. The Input Channel Offset and Output Channel Offset indicates the offset from the beginning of that register set. A blank entry indicates that that particular register is not present for that channel.

TABLE 5. Video Input and Output Channel Registers

Input Channel Offset	Output Channel Offset	Type	Bits	Register Name	Register Function
0x0	0x0	RW	9:0	CONTROL	DMA Control
0x8	0x8	RW	10:0	STATUS	DMA Status
0x10	-	RW	16:0	CONFIG	Input Channel configuration options
-	0x10	RW	21:0	CONFIG	Output Channel configuration options
0x18	0x18	RW	31:6 2:0	NEXT_DESC*	Pointer to next descriptor block
0x20	0x20	RW	15:3	FLD_OFFSET*	Offset into first page for first pixel
0x28	-	RW	11:3	LINE_WIDTH*	Width of each video line in bytes
	0x28	RW	21:0	FIELD_SIZE*	Width and height of output field
0x30	-	RW	40:0	HCLIP_ODD	Specifies the pixel boundaries for odd video input line
0x38	-	RW	38:0	VCLIP_ODD	Specifies the line boundaries for odd video input line
0x40	-	RW	7:0	ALPHA_ODD	Alpha value for RGBA input pixels for odd video input line
0x48	-	RW	40:0	HCLIP_EVEN	Specifies the pixel boundaries for even video inpu line
0x50	-	RW	38:0	VCLIP_EVEN	Specifies the line boundaries for even video input line
0x58	-	RW	7:0	ALPHA_EVEN	Alpha value for RGBA input pixels for even video inpu line
-	0x30	RW	41:0	HPAD_ODD	Specifies the pixel boundaries for odd video output line
-	0x38	RW	39:0	VPAD_ODD	Specifies the line boundaries for odd video output line
-	0x40	RW	41:0	HPAD_EVEN	Specifies the pixel boundaries for even video output line
-	0x48	RW	39:0	VPAD_EVEN	Specifies the line boundaries for even video output line
-	0x50	RW	10:0	GEN_DLY	Genlock Delay Value
-	0x58	RW	18:0	VHW_CFG	Hardware Configuration
0x80 - 0xB8	0x80 - 0xB8	RW	63:0	DMA_DESC	Dma Descriptors

Notes:

1. the “*” indicates this register is double-buffered, that is, the contents are not used by the hardware until the beginning of the next field. These double-buffered registers are updated simultaneously when the next descriptor address (NDA) is read for the next field. A “-” indicates that this register is not present for this type of channel (input or output).

2.3.5.1 Control Register

This register controls the individual channel’s operation and enables various interrupt. All bits initialize to 0 on reset..

TABLE 6. VCHAN-CONTROL - Control Register Bit Fields - Video Input

Bits	Type	Description
0	RW	Enable DMA (0 - resets DMA)
1	RW	Enable Vertical Sync Interrupt
2	RW	Enable DMA Complete Interrupt
3	RW	Enable Lost Sync Interrupt (external pin)
4	RW	Enable Buffer Field Overflow Interrupt
5	RW	Enable Horizontal Overflow Interrupt
6	RW	Enable Vertical Overflow Interrupt
7	RW	Enable FIFO Overflow Interrupt
8	RW	Enable CRIME memory error Interrupt
9	RW	Enable GPIB Interrupt

TABLE 7. VCHAN-CONTROL - Control Register Bit Fields - Video Output

Bits	Type	Description
0	RW	Enable DMA (0 - resets DMA)
1	RW	Enable Vertical Sync Interrupt
2	RW	Enable DMA Complete Interrupt
3	RW	Enable Genlock Lost Interrupt
4	RW	Enable Buffer Field Overflow Interrupt
5	RW	Not used
6	RW	Not used
7	RW	Enable FIFO Underflow Interrupt
8	RW	Enable CRIME memory error Interrupt

2.3.5.1.1 Enable DMA

A zero in this bit stops any ongoing DMA operation and resets the DMA engine to a quiescent state. A one in this bit enables the DMA engine to begin processing input and output pixels.

2.3.5.1.2 Vertical Sync Interrupt

Vertical sync interrupt occurs when the video input detects Start of Field or the video output generates Start of Field (internal timing) or detects Start of Field (genlocked). This interrupt, when enabled, occurs as long as the video signal is active.

2.3.5.1.3 DMA Complete Interrupt

DMA Complete interrupt occurs when the last byte of the field buffer has been transferred either to memory (video input) or has been read from FIFO (video output).

2.3.5.1.4 Lost Sync / Genlock Lost Interrupt

Lost sync interrupt occurs when the HLOCK input pin becomes inactive. This is normally caused by either removing the video connector or removing power from the external video equipment.The Sync Present status bit indicates the dynamic state of the sync of this channel.

For video out, this interrupt is generated when synchronization with the genlock signal is lost.

2.3.5.1.5 Buffer Field Overflow

Buffer field overflow occurs when the DMA engine encounters a DMA descriptor with a value of zero. This indicates that the expected video field is larger than the allocated buffer. The current transfer (if any) is terminated and the FIFO is reset.

2.3.5.1.6 Horizontal Field Overflow

For video input only, this interrupt occurs when the video interface is programmed to capture data past the end of the current line.

2.3.5.1.7 Vertical Field Overflow

For video input only, this interrupt occurs when the video interface is programmed to capture data beyond the last line of the current field.

2.3.5.1.8 FIFO Underflow/Overflow

FIFO underflow or overflow occurs when the DMA engine is not filling or emptying the FIFO at a rate sufficient to maintain the video rate. Underflow occurs on video output, while overflow occurs on video input. The current transfer (if any) is stopped and no further memory transfers occur for this channel.

2.3.5.1.9 CRIME Memory Error

The CRIME memory error enables the detection of those errors in dealing with CRIME memory transfers.

2.3.5.1.10 GPIB Interrupt

This interrupt occurs when the GPIB input pin is active (logic ‘1’).

2.3.5.2 DMA Status Register

The Status Register indicates which interrupts are active from the device. Reading this register clears any bits that are active. Rereading the register a second time will indicate interrupts that occurred since the last time this register was read. All bits initialize to 0 on reset..

TABLE 8. VCHAN_STAT -DMA Status Register Bit Fields - Video Input

Bits	Type	Description
0	RO	DMA Active Status (*)
1	RO	Vertical Sync Interrupt
2	RO	DMA Complete Interrupt - Last line of video has been transferred.
3	RO	Lost Sync Interrupt - Input Device detected lost sync.
4	RO	Buffer Field Overflow - incoming video field exceeded buffer size, (encountered zero memory descriptor)
5	RO	Horizontal Overflow Interrupt
6	RO	Vertical Overflow Interrupt
7	RO	FIFO Overflow - CRIME memory interface failed to keep up with FIFO requirements.
8	RO	CRIME Memory Error Interrupt
9	RO	GPIB Interrupt
10	RO	Sync Present Status (*)

TABLE 9. VCHAN_STAT -DMA Status Register Bit Fields - Video Output

Bits	Type	Description
0	RO	DMA Active Status (*)
1	RO	Vertical Sync Interrupt
2	RO	DMA Complete Interrupt - Last line of video has been transferred.
3	RO	Lost Sync Interrupt - Input Device detected lost sync.
4	RO	Buffer Field Overflow - outgoing video field exceeded buffer size, (encountered zero memory descriptor)
5	RO	Not Used
6	RO	Not Used
7	RO	FIFO Underflow - CRIME memory interface failed to keep up with FIFO requirements.
8	RO	CRIME Memory Error Interrupt

Note: “*” indicates that this register does not reset to zero when read but reflects the active state of this status signal.

2.3.5.2.1 DMA Active Status

The DMA Active Status reflects the “active” status of the DMA transfer and does not reset to zero when read.

2.3.5.2.2 Sync Present Status

The Sync Present Status reflects the “active” state of the HLOCK Pin and does not reset to zero when read.

2.3.5.3 Configuration Registers

The Configuration register is specific for each of the input channels and the output channel.

TABLE 10. Input Channel Configuration Register

Bits	Field Name	Description
0	CHANNEL RESET	0 = reset processing this channel 1 = enable processing this channel
1	D1 RESET	0= reset D1 circuitry 1= enable D1 circuitry
3:2	VIN_SOURCE	Video Input Source 00 = Use primary D1/analog port (AB) as source 01 = Use secondary D1 port/moosecam (CD) as source 10 = Loopback mode: use video output port E as source 11 = Loopback mode: use video output port F as source
4	D1 PRECISION	External Pixel Precision of input D1 stream 0 = 8-bit D1 1 = 10-bit D1
5	D1 ECC	Error Correction of input D1 stream 0 = disable D1 error correction 1 = enable D1 error correction
9:6	D1 SOF_COUNT	Start of field count for input D1 stream Number of lines to wait after start of vertical blank (following active video) before checking field id bit
12:10	DMA Pixel FORMAT	000 = Open GL RGBA 32-bit (8-bit per component) 001= 16-bit RGBA (5551) 010= VL 4:2:2 YUV (8-bit per component) 011 = VL 4:2:2 YUV (10-bit per component) 100 = Open GL ABGR 32-bit (8-bit per component) 101 = VL 4:2:2:4 YUVA (8-bit per component)
13	DITHER	Only used if 16-bit RGB format is selected 0 = disable dither 1= enable dither
15:14	MEM_MODE	00 = Linear 64K buffers 01 = Tiled 64K buffers 10 = Mip-mapped 64K buffers starting with 512x128 image 11 = Mip-mapped 64K buffers starting with 512x256 image
16	INTERLEAVED	Interleaved Mode 0 = field mode 1 = frame (interleaved) mode

2.3.5.3.1 Channel Reset

The configuration register contains control bits which are changed infrequently and cannot be changed ‘on the fly’. The channel will power up in reset mode and should be left in reset mode until all relevant registers have been programmed. The channel should be in reset mode whenever any fields in this configuration register are to be changed. It does not affect the DMA section.

2.3.5.3.2 D1 Reset

The D1 should be in reset mode whenever any changes are made affecting the stability of the D1 input source (ie, changing the select in the Video Hardware configuration register). This reset basically resets all the circuitry in the seciton of the video input channel which decodes the D1 stream.

2.3.5.3.3 VIN_SOURCE - Video Input Source

There are 4 possible sources for each video input channel: the 2 D1 input sources or the 2 output ports. When one of the D1 input sources is selected, the Video Hardware Configuration register (described later) must be also be programmed to select which of the 2 external D1 ports is selected for the desired D1 input source.

Loopback mode allows the programmer to send data from memory out through the video output channel and back in through the video input channel. This allows the programmer to take advantage of the hardware processing capabilities of the chip. To use Loopback mode, select either the E or F port to be passed back into video input. The Video Ouput Configuration register (described later) must be programmed to select which D1 stream (YUV or alpha) will driven to ports E and F. When using Loopback mode, you will still get a video output stream. Note that for Loopback mode to work properly, the video timing for the input and output channels must be programmed consistently.

Selection of the input source for each of the 2 video input channels is independent. Each channel may be programmed to use either of the 4 sources regardless of the source used by the other channel. Both channels may alos use the same source.

2.3.5.3.4 D1 PRECISION

This field indicates whether the input D1 stream should be interpreted as 8-bit D1 or 10-bit D1. This is used solely in detection of marker codes. In 8-bit mode, only the upper 8 bits of data will be used to detect the 0xFF and 0x00 marker codes. In 10-bit mode, all 10-bits will be used to detect the 0x3FF and 0x000 marker codes. This bit does not affect the precision of the D1 data which is captured and is independent of the DMA data format used. That is, you can set 8-bit mode and still capture 10-bit data, or vice versa.

2.3.5.3.5 D1 ECC - Error Correction

This bit determines whether error correction will be applied to the control codes D1 source. Note, that when capturing imbedded control codes as data, the non-corrected codes will be captured.

2.3.5.3.6 D1 SOF_COUNT - Start of field count

Due to variation in interpretation of the D1 standard, there is some ambiguity as to when the “start of field” (SOF) occurs. Since there is no specific control code indicating SOF, this event it is normally inferred. This count allows for a (hopefully) foolproof method of dealing with this situation by making SOF programmable. The SOF can thus occur at a somewhat arbitrary point, although the programmer would normally want to be consistent with the spec. The Field ID bit in the control codes MUST be valid for the current field at SOF. The SOF count is the number of lines between the start of the D1 vertical blanking period which follows active video and the start of the next field. Thus, it would normally equal the number of blank lines at the bottom of the field. Note that the VSYNC interrupt occurs at SOF.

2.3.5.3.7 FORMAT - DMA Pixel Format

The DMA Pixel Formats are described in the subsection Pixel Data Formats. The video input channel will perform upsampling and color space conversion (YUV to RGB) as required by the specified format. Note that all data is processed as 8-bit components. When the YUV 10-bit format is selected all processing (except for clipping) is bypassed.

2.3.5.3.8 DITHER

The dither bit enables dithering when RGB16 is selected as the video input pixel format. When a pixel format other than RGB16 is selected, this bit will be ignored.

2.3.5.3.9 MEM_MODE - Memory Mode

This selects which type of buffers will be used in memory.

2.3.5.3.10 INTERLEAVED - Interleaved Mode

A one will enable interleaved (or frame) mode.

TABLE 11. Output Channel Configuration Register

Bits	Field Name	Description
0	CHANNEL RESET	Channel Reset 0 = reset processing this channel 1 = enable processing this channel
2:1	PORT_E_SOURCE	Data source for video output port E 00 = disble port 01 = pass thru D1 input (genlock source) 10 = data from YUV stream 11 = data from alpha stream
4:3	PORT_F_SOURCE	Data source for video output port F 00 = disble port 01 = pass thru D1 input (genlock source) 10 = data from YUV stream 11 = data from alpha stream
5	Genlock Reset	Reset Genlock and video output circuitry 0= reset 1= enable
7:6	Genlock Source	Output Genlock Source 00 = no genlock (use 13.5 Mhz pixel clock and internal timing) 01 = external sync (hvf) input 10 = Port AB D1 input 11 = Port CD D1 input
8	Genlock Precision	External Pixel Precision for genlock source 0 = pixels are 8-bit D1 1 = pixels are 10-bit D1
9	Genlock Ecc	Error Correction for genlock source 0 = disable D1 error correction 1 = enable D1 error correction
13:10	Genlock SOF_COUNT for Odd field	Number of lines to wait after start of vertical blank (following active video) before checking field id bit for odd field
16:14	Pixel FORMAT	DMA Pixel Format 000 = Open GL RGBA 32-bit (8-bit per component) 001= 16-bit RGBA (5551) 010= VL 4:2:2 YUV (8-bit per component) 011 = VL 4:2:2 YUV (10-bit per component) 100 = Open GL ABGR 32-bit (8-bit per component) 101 = VL 4:2:2:4 YUVA (8-bit per component)
17	NOTCH_FILTER	This is generally used if RGB data to be output has been previously dithered 0 = disable filter 1 = enable filter
19:18	CLAMPING/ EXPANSION	00 = no clamping 01 = clamp to extended data range 10 = clamp to legal YCrCb range 11 = special expansion
20	INTERLEAVED	Interleaved Mode 0 = field mode 1 = frame (interleaved) mode
21	MEM_MODE	Memory Mode 0 = Linear 64k Buffers 1 = Tiled 64k Buffers

TABLE 11. Output Channel Configuration Register

Bits	Field Name	Description
25:22	Genlock SOF_COUNT for Even field	Number of lines to wait after start of vertical blank (following active video) before checking field id bit for even field

2.3.5.3.11 Channel Reset

The configuration register contains control bits which are changed infrequently and cannot be changed ‘on the fly’. The channel will power up in reset mode and should be left in reset mode until all relevant registers have been programmed. The channel should be in reset mode whenever any fields in this configuration register are to be changed. It does not affect the DMA section.

2.3.5.3.12 PORT_E_SOURCE/PORT_F_SOURCE - D1 Video Output Source

The video output section always produces 2 data streams internally, YUV and alpha, regardless of the DMA data format. (If the DMA data format is YUV, then the alpha stream defaults to the equivalent of black). The chip itself has 2 D1 data ports: E and F. The programmer can route either data stream, YUV or alpha, independently to either output port. A third option, passthru, will route data from the D1 genlock source to either output port. Passthru mode can only be used if a D1 input has been selected as the genlock source.

Note that the analog decoder hangs transparently off the E port and so will get whatever data stream goes to the E-port.

2.3.5.3.13 Genlock Reset

The genlock should be in reset mode whenever any genlock parameter is changed or when the genlock source is not expected to be stable or match the video output channel timing. This reset basically resets all the circuitry in the video output channel which uses the genlock (or default 27 MHz) clock. This bit must be set to enable to produce video output (or loopback data to video input), even if no output genlock source is specified.

2.3.5.3.14 Genlock Source

The video output D1 streams can be locked to either one of the D1 input sources. It can also be lock to an external sync source which provides H/V/F signals. If no output genlock source is selected, the D1 output streams will free run using a 27 MHz clock. Note that there is only one genlock source which is used for both the E and F ports. For genlock to work properly, the programmed video timing for the video output (see VPAD and HPAD registers) must match that of the genlock source.

To run in loopback (to video input) mode with no video output, select no genlock source.

2.3.5.3.15 Genlock Precision

This bit is used only if the output is genlocked to one of the D1 input ports. It indicates whether the input D1 stream should be interpreted as 8-bit D1 or 10-bit D1. This is used solely in detection of marker codes. In 8-bit mode, only the upper 8 bits of data will be used to detect the 0xFF and 0x00 marker codes. In 10-bit mode, all 10-bits will be used to detect the 0x3FF and 0x000 marker codes. This bit does not affect the precision of the video output D1 streams.

2.3.5.3.16 Genlock ECC - Error Correction

This bit is used only if the output is genlocked to one of the D1 input ports. It determines whether error correction will be applied to the genlock source D1 data stream.This bit does not affect the video output D1 stream, which will always include error correction bits.

2.3.5.3.17 Genlock SOF Count

This field is used only if the output is genlocked to one of the D1 input ports or to the external sync (HVF) source. Please see the description of the D1 SOF Count field of the Video Input Configuration Register.

2.3.5.3.18 FORMAT - DMA Pixel Format

The DMA Pixel Formats are described in the subsection Pixel Data Formats. The video output channel will perform color space conversion (RGB to YUV), expansion and subsampling (to YUV 4:2:2) as required by the specified format. Note that all data is processed as 8-bit components. The YUV 10-bit format is preserved as 10-bits only if no filtering is performed.

2.3.5.3.19 NOTCH_FILTER

The notch filter is a 1/2 0 1/2 FIR filter. It is useful when the video data has previously been dithered. When DMA data is a YUV format, the notch filter will only be applied to the Y component. When the DMA data is an RGB format, the notch filter, is applied to the Y, U and V components after the data has been converted to YUV space, but before it is subsampled. The filter is never applied to alpha.

2.3.5.3.20 CLAMPING/EXPANSION

The video output section always attempts to generate a d1 output stream that will be valid as either 8-bit D1 or 10-bit D1. There is no configuration bit indicating which mode the destination device operates in. Marker codes and control codes will be generated as 10-bit values and thus will be compatible with either 8 or 10 bit D1 devices. If DMA data is sent in 10-bit YUV format and is not processed, the precision will not be altered. If DMA data is any other format or undergoes any processing in the video output path, the resultant final YUV data will have only 8-bit precision. The data will normally be expanded to 10 bits by left justifying and adding zeros to the 2 lsb's. This expansion will occur before the clamping function, which may subsequently alter the data. Note that both expansion and clamping will occur after all color space conversion and filtering.

There are 4 options for clamping/expanding data which is passed out of the MACE chip via the D1 ports.

No Clamping:
Data will not be altered, with the following exception for expansion:
If the DMA pixel format is NOT 10-bit YUV then the 8-bit value 0xFF will be expanded to 0X3FF.
This option only makes sense if no color space conversion is being done on the output. It is most likely to be used when sending the image along with imbedded ancillary data. Selecting this mode will preserve any embedded marker codes (0x000 and 0x3FF). The exception described above is used to preserve embedded 8-bit marker codes.

Extended Range Clamping:
Data is altered only to avoid sending marker codes in the data stream. Thus:
0x3FF is changed to 0x3FB
0x000 is changed to 0x004
This option assures that there will be no possible confusion of data with the control code sequence, without significantly modifying the image data. This may be used when the MACE chip is configured to drive the output back to the input, so that the programmer can maximize the range of data which is processed by the chip.

Legal Range Clamping:
Data is clamped to legal YCrCb ranges. For 8-bit data the range is 16-235 for Y and alpha, 16-240 for Cr/

C. For 10-bit data the range is 64-940 for Y and alpha, 64-960 for Cr/Cb. This option will be used in typical applications when image data is being color space converted or processed and output to a video device. The actual clamping will be performed after all processing and filtering.

Special Expansion:
This is a special expansion function which has been included for the sole purpose of allowing the D1 interface to be used as a general purpose interface sending non-video data. This effectively modifies the way expansion from 8-bit to 10-bit is done. The 8-bit data will be right-justified and the 2 msbs will be set to “01”. No clamping is done.

INTERLEAVED - Interleaved Mode

A one will enable interleaved (or frame) mode of input or output.

2.3.5.3.21 MEM_MODE - Memory Mode

This selects which type of buffers will be used in memory.

2.3.5.4 Next Descriptor Address - Video Input

TABLE 12. Next Descriptor Address

Bits	Description
31:6	Next Descriptor Address (aligned on 64 byte boundary)
2	Valid Bit
1:0	Field Capture Bits 0x - Capture next field (either type) 10 - Capture next odd field 11 - Capture next even field

TABLE 13. Next Descriptor Address - Video Output

Bits	Description
31:6	Next Descriptor Address (aligned on 64 byte boundary)
2	Valid Bit
1	Not used
0	Field to send 0 - send as next odd field 1 - send as next even field

The address in the NDA register is restricted to be on a 64 byte boundary.The Next Descriptor Address (NDA) register points to the DMA Descriptor table in memory and is loaded with the address of the first descriptor by software. During vertical blanking, the DMA descriptor page table is read into the translation cache. If the Valid bit is set to 0, then the next field is skipped (not transferred to/from memory). This register is double buffered and the buffer is loaded into the NDA register at the end of the field.

The Valid Bit is set to one by software to indicate that a valid address has been written to this register. The hardware will zero this bit after it is copied. The Field Capture Bits indicate which field to begin processing for this descriptor. This allows the software to implement an odd or even “dominance”.

For video output, bit 1 of this register is ignored, and bit 0 defines the output field ID.

2.3.5.5 Field Offset

The Field Offset register points to the first pixel of the first page of the buffer. In linear mode, it can span the entire 64K page but is restricted to be on an 8 byte boundary. It is normally written by software at the beginning of each field when interleaving fields into frames, or for cases where the input field will be put into the 2nd part of a 64k page that is being shared between fields or frames. This register is double buffered and the buffer is loaded into the Field Offset register at the end of a field.

In Tiled mode, field offsets are restricted to 512 byte boundaries, i.e. the image offset must be left-justified within the tile.

TABLE 14. FLD_OFFS - Field Offset Register

Bits	Description
15:3	Byte Offset of first pixel in first page

2.3.5.6 Line Width (Video input only)

The Line Width register is programmed with the number of bytes in each video line. In Linear mode with Interleaving enabled, it is added to the current pixel address at the end of a line so that an empty video line is left for input, or on the second field, the previous odd video lines are skipped.

TABLE 15. LINE_WIDTH - Line Width Register

Bits	Description
11:3	Line Width in bytes (multiple of 8 bytes)

2.3.5.7 Field Size (Video output only)

This register defines the output field size.

TABLE 16. Field Size

Bits	Description
21:12	Number of lines in field.
11:3	Line Width in bytes (multiple of 8 bytes)

2.3.5.8 Input Filtering, Clipping and Scaling Registers

The incoming video stream may contain a variable number of pixels per line and lines per field, both visible and blanked, depending on the input device. The programmer may choose to capture just active video, or may capture active video and some portion of the blanking period, which may contain ancillary data. The data window which is captured is fully programmable and thus can support a number of standard or non-standard devices. There are however, practical limitations to this. Mace Video is being designed to support standard non-square NTSC and PAL D1 formats, pseudo-standard NTSC and PAL square D1 formats, as well as all formats driven by the moosecam.

The registers described below will determined how data is captured from the incoming D1 stream and how it will be resized. There are 2 sets of registers: one for the even field and one for the odd field. This allows the programmer to dynamically and smoothly change the capture region (clipping) and re-sizing. To accomplish this, the programmer would change the odd registers after the start of the even field, and the even registers after the start of the odd field, such that the hardware will always be using a quiescent register. The burden for smooth transitions is entirely on the programmer. The hardware merely looks at the

appropriate register during each field and if the register changes at the wrong time, something undefined will happen.

In the horizontal direction, there are 3 sizing blocks executed in the following order:
input clipping => non-square to square conversion => horizontal down-scaling

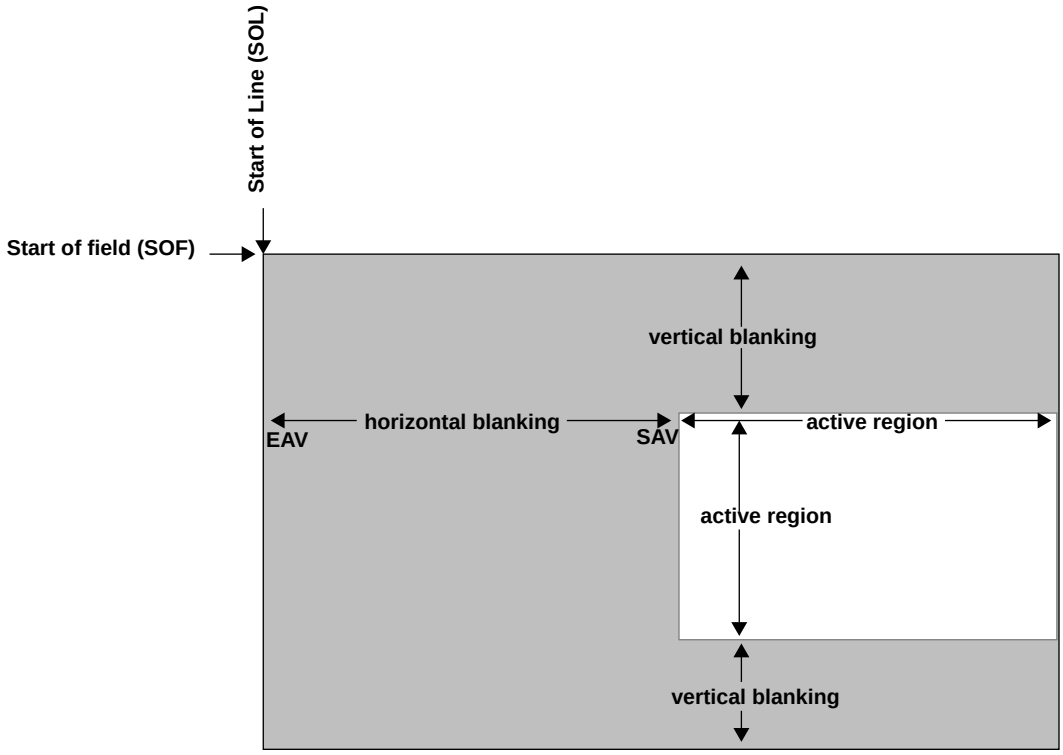
In the vertical direction there are 2 sizing blocks executed in the following order:
input clipping => vertical down-scaling

With the exception of the input clipping, the resizing blocks are all optional and independently programmed. NOTE: ALL CLIPPING REGISTERS, BOTH EVEN AND ODD, MUST BE PROGRAMMED WITH VALID VALUES BEFORE THE CHANNEL IS ENABLED.

TABLE 17. HCLIP (_ODD & _EVEN) Registers

Bits	Name	Description
9:0	Hstart	Starting pixel position of video to be captured. Relative to the start of horizontal blanking. = # pixels preceding region to be clipped - 2
19:10	Hend	Ending pixel position of video to be captured. Relative to the start of horizontal blanking. = HSTART + # pixels to be clipped - 1
29:20	Filt_cnt	Number of pixels per line at output of nonsquare to square conversion filter. This count is used only if the nonsqaure to square conversion filter is used. This must be an even number.
30:37	H Scaling ratio	This 8-bit value represents the ratio for Horizontal scaling (see explanation of scaling ratios below). This is essentialy the ratio by which the image will be scaled down in the horizontal direction multiplied by 256. Used only if horizontal scaling is enabled.
38	H Scaling on	0 = no scaling 1 = scale down horizontally
39:40	Filter mode	Mode for Nonsquare to square conversion filter: 00 = none 01 = pal nonsquare to square (scale up by 7868/720) 10 = ntsc nonsquare to square (scale down by 654/720) 11 = scale down by 1/2

Hstart and Hend define the region of active video which will be clipped from the D1 stream. This may be either the entire active video region, an area within the active video region, or any arbitrary area including the active video and/or a protion of the blanking regions. Both values are pixel counts relative the the “start of line” which is defined as the end of active video (EAV) control sequence. Pixel 0 is defined as the first pixel after the EAV control code, thus it is not possible to capture the first 4 bytes which comprise the EAV control sequence. Clipping can only be done on even pixel boundaries since D1 consist of u-y-v-y pixel pairs.



EAV = 4 byte control sequence for end of active video
SAV = 4 byte control sequence for start of active video

Some examples:

To capture all of active video:
hstart = #blank pixels - 2
hend = hstart + #active pixels - 1

To capture a smaller region within the active video region:
hstart = #blank pixels - 2 + #pixels in the active region preceding area to be clipped
hend = hstart + #pixels to be clipped - 1

To capture a region including active video and part of the blanking region:
hstart = #pixels preceding region to be clipped -2
hend = hstart + #pixels to be clipped - 1

If the nonsqaure to square conversion filter is to be used, the Filter Count field must be used. This value is required since it would be cumbersome and inexact to have the hardware calculate this. This should be set to the number of pixels -1 which will be output by the Filter. Again, since all hardware operates on pixel pairs, this must reflect an even number of pixels. The number of pixels which will be produced should be calculated by multiplying the selected scale factor by the number of pixels produced by the clipping logic. This number can then be rounded either up or down to the nearest even pixel count (but no further). Note that the scaling factors applied by the filter are the same regardless the number of pixels in the clipped region. The setting of the filter mode is independent of the actual D1 format being used on the input source. If the filter mode is set to none, no conversion will occur and the Filter Count will not be used.

Note that when dynamically changing the clipping region, the Filter Count (if the filter is used) must also be adjusted with the change in the clipping region size.

Note that there are restrictions imposed by the DMA requiring the total line size to be a multiple of 8 bytes. Thus, it is incumbent upon the progammer to produce a multiple of 8 bytes given the combination of clipping, filtering and scaling and selected pixel format. This requirement is explained in detail in the section on DMA.

TABLE 18. VCLIP (_ODD & _EVEN) Registers

Bits	Name	Description
9:0	VStart	Starting line position of video to be captured. Relative to the start of field. = # lines preceding region to be clipped
19:10	Vblkend	Line position of video to end padding with black. Relative to the start of field. =
29:20	VEnd	Ending line position of video to be captured. Relative to the start of field. = VSTART + # lines to be clipped - 1
37:30	V Scaling ratio	This 8-bit value represents the ratio for vertical scaling (see explanation of scaling ratios below). This is essentialy the ratio by which the image will be scaled down in the vertical direction multiplied by 256. Used only if vertical scaling is enabled.
38	V Scaling on	0 = no scaling 1 = scale down vertically

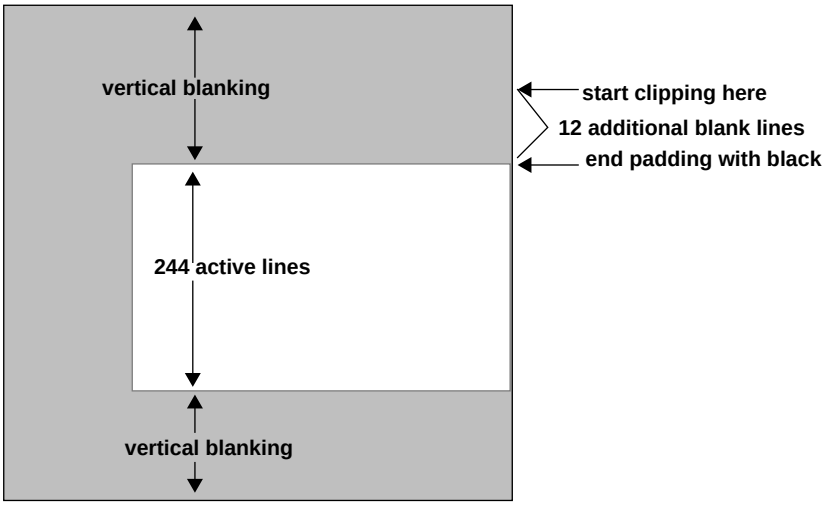
Vstart and Vend define the region of active video which will be clipped from the D1 stream. This may be either the entire active video region, an area within the active video region, or any arbitrary area include either the active video or blanking regions. Both values are line counts relative to the “start of field” (SOF). (See Video Input Configuration regsiter for an expalnation of SOF). Line 0 is defined as the first line after the SOF. Some examples:

To capture all of active video:
vstart = #blank lines preceding image
vend = vstart + #active lines - 1

To capture a smaller region within the active video region:
vstart = #blank lines preceding image + #lines within active region preceding area to be clipped
vend = vstart + #lines to be clipped - 1

To capture a region including active video and part of the blanking region:
vstart = #lines preceding region to be clipped
vend = vstart + #lines to be clipped - 1

The Vblkend field allows for the use of a special function which allows the programmer to pad a small number of black lines at the beginning of the field. This feature is intended to be used in conjunction with mipmapping, if the actual number of visible lines in a field is less than the desired starting mipmap. Using the mipmap function requires that the number of lines (after clipping and vertical scaling) be either 128 or 256. If the actual number of lines in the active portion of video is less than the desired mipmap size, the programmer can program the clipping register to capture additional lines from the vertical blanking region which precedes the active area. This will put extra blank lines at the top of the image. Using the Vblkend field will force these extra lines to black. When the Vblkend field is set to a value greater than Vstart, the captured lines at the start of the field from the Vstart line to the Vblkend line will be forced to black. To use this function, there must be sufficient blank lines in the actual D1 field, so that the desired total number of lines can be captured. If no additional black lines are to be added to the image, set Vblkend to 0.



Example:

To capture all of active video with additional black lines preceding the image:
vstart = #blank lines preceding image - #additional black lines
vblkend = vstart + #additional black lines
vend = vstart + #additional black lines + #active lines - 1

2.3.5.8.1 Limitations on Clipping Regions

Theoretically, it is possible to program almost any clipping region, constrained only by the bit width of the various clipping register fields. In reality, however, there are a number of limitations on what can be clipped. The limitations are somewhat difficult to enumerate concisely, and at this point, not all the potential limitations may be known. It is expected that there will be no restrictions when clipping a ‘realistic’ image from an actual D1 stream, however it is possible to hit a corner case that produces and error when attempting to capture an ‘unrealistic’ artificial video scenario. The restrictions are:

- It is not possible to capture the first or last line of a field.
- There must be at least 8 blank or uncaptured pixels in a line. (possibly less)
- When the Pal non-square to square conversion filter is enabled, the chip generates more pixels than it receives. It uses a portion of the blanking period to generate these extra pixels. Therefore the horizontal blanking period must be at least the difference between the number of active (captured) pixels going into the filter and the number going out of the filter (as indicated by the filter count field), or a minimum of 16.
- When Mipmapping is used, there must be at least 138 blank (or uncaptured)pixels per line. And at least 2 bl;ank (uncaptured) lines at the bottom of the field.
- It may not be possible to capture 720 pixels, perform Pal non-square to square conversion to produce 768 pixels, then scale down with the horizontal scaler to produce 512 pixels and then mipmap the results. (Don’t ask!).

- When using vertical scaling, there can be no more thatn 768 pixels per line (after non-square to square conversion and horizontal scaling).
- The number of active (captured) pixels per line must be a multiple of 2 pixels at any point in the video processing channel. The final number of bytes per line (a function of the pixel format) must always be a multiple of 8 bytes.

2.3.5.8.2 Scaling Ratio

The following c-code shows an example of how to calculate the scaling ratio for either the vertical or horizontal scaling. Using this algorithm will ensure that the hardware generates the number of pixels or lines desired. Since there are only 255 levels of scaling, there are some output sizes which cannot be produced from the given input size. For horizontal scaling, both the input size and the output size must the number of U/V samples, that is, the size divided by 2.

```
int insize, outsize; /* number of input pixels or lines, desired number of output pixels or lines
float scalefactor;
int r; /* scaling factor to enter into register field

scalefactor = (((float) outsize) / ((float) insize)) * 256.0 + 0.5;
r = (int) scalefactor

while ( (r/2 + (insize-2)*r + 256) < (outsize*256) )
 r++;

while ((r/2 + (insize-2)*r + 1) > (outsize*256))
 r--;

if ((r < 1 || r > 255) ||  (r/2 + (insize-2)*r + 256) < (outsize*256) )
 /* it is not possible to produce the desired number of output pixels/lines */
 ;
```

2.3.5.9 ALPHAIN(_ODD & _EVN) - Alpha Registers

The Alpha register is used in the video input channels to supply alpha when the DMA pixel format is an RGBA format. There are 2 copies of this register: one for the even field and one for the odd field, so that the alpha values can be programmed to change smoothly on field boundaries.

TABLE 19. ALPHA(_ODD,_IN) Registers

Bits	Description
7:0	8 Bit value placed in the Alpha component of the RGBA pixel

2.3.5.10 Output Filtering and Padding Registers

The outgoing video stream may contain a variable number of pixels per line and lines per field, both visible and blanked, depending on the output device. The programmer may choose to output just active video, or may output active video and some portion of the blanking period, which may contain ancillary data. The data window which is output is fully programmable and thus can support a number of standard or non-standard devices. There are however, practical limitations to this. Mace Video is being designed to support standard non-square NTSC and PAL D1 formats, pseudo-standard NTSC and PAL square D1 formats.

The output pad and timing registers determine the horizontal and vertical blanking periods for the ouput D1 stream, as well as where the data is placed within the D1 field. This the programmer may also specify padding of the outgoing image so that it can be displayed in a larger window. This will allow the DMA stream to have fewer pixels than the required visible picture region. The hardware will pad with the correct level of black to allow compatibility with files that may not fill a selected Television output standard. This will allow playback of smaller than full size video pictures through the video output port with the active picture framed in black while still meeting the various fixed timing formats of PAL and NTSC.There are 2 sets of registers: one for the even field and one for the odd field. This allows the programmer to dynamically and smoothly change the padding region and re-sizing. To accomplish this, the programmer would change the odd registers after the start of the even field, and the even registers after the start of the odd field, such that the hardware will always be using a quiescent register. The burden for smooth transitions is entirely on the programmer. The hardware merely looks at the appropriate register during each field and if the register changes at the wrong time, something undefined will happen.

For horizontal padding and timing, all values are expressed in pixel pairs (or 4-byte quantities in the D1 stream). The horizontal counters are all 10-bits which allows for 1024 pixel pairs, or 2048 pixels or 4096 bytes. In the normal case where only image data is sent, the hardware will insert the correct EAV and SAV control codes in the D1 stream. If the programmer is output image data as well as a portion of the blanking region, then the DMA data must include the SAV control code in the correct location. The EAV control code is always generated by the hardware.

TABLE 20. HPAD (_ODD & _EVEN) Registers

Bits	Name	Description
9:0	EAV to SAV	EAV control code to the SAV control code: #blank pixels/2 - 2
19:10	EAV to EAV	EAV control code to EAV control code: #total pixels/2 - 1
29:20	EAV to image	EAV control code to start of data: #pixels preceding image/2 -1
39:30	Filter Count	Total number of pixels at output of square to non-square filter
41:40	Filter Mode	00 = no filtering 01= PAL square to non-square conversion 11/12 10 = NTSC square to non-square conversion 11/10 11 = zoom up by 2 horizontally

Some examples:

To output active video with no additional padding:
EAV to SAV = #blank pixels/2 - 2
EAV to image = #blank pixels/2 - 1
EAV to EAV = #total pixels/2 - 1

To output active video with padding:
EAV to SAV = #blank pixels/2 - 2
EAV to image = #blank pixels/2 + #pixels of padding/2 - 1
EAV to EAV = #total pixels/2 - 1

To output active video and part of the blanking region:
EAV to SAV = not used, set to all 1’s
EAV to image = #pixels preceding region to be output/2 -1
EAV to EAV = #total pixels/2 - 1

Note that the hardware will automatically pad to black after the end of the image until the EAV.

In addition to the input padding, there are registers provided to specify the number of pixels output from the Mitchell filter stage, since it would be cumbersome and inexact to have the hardware determine this. Thus in addition to specifying the total number of pixels and lines being output by DMA, the programmer specifies the number of pixels to be output by the Mitchell filter. This number should be determined from the actual scaling factor, however, it may be rounded up or down. If the output of any filter stage is programmed to be less than the actual size produced, clipping will occur. It is not recommended to use a number greater than that produced by the relevant filter stage, other than rounding up to the nearest whole. Note that there are restrictions imposed by the DMA requiring the total line size to be a multiple of 8 byte.

TABLE 21. VPAD(_EVEN, _ODD) Registers

Bits	Name	Description
9:0	SOF to end Vblank	Start of field to the end of the vblank region which precedes active video. #blank lines preceding image - 1
19:10	SOF to start of vblank	Start of field to the start of the vertical blanking region which follows active video. #lines preceding final vertcal blanking region -1
29:20	SOF to EOF	Start of field to end of field #total lines - 1
39:30	SOF to start image	Start of field to start of active data. #lines preceding image - 1

Some examples:EAV

To output active video with no additional padding:
SOF to end vblank = #blank lines preceding active video -1
SOF to start image = #blank lines preceding active video -1
SOF to start vblank = #blank lines before active video + lines of active video - 1
SOF to EOF = # total lines -1

To output active video with padding:
SOF to end vblank = #blank lines preceding active video -1
SOF to start image = #blank lines preceding active video + #lines padding preceding image -1
SOF to start vblank = #blank lines before active video + lines of active video - 1
SOF to EOF = # total lines -1

To output active video and part of the blanking region:
SOF to end vblank = #blank lines preceding active video - 1
SOF to start image = #lines preceding image -1
SOF to start vblank = #blank lines before active video + lines of active video - 1
SOF to EOF = # total lines -1

Note that the hardware will automatically pad to black after the end of the image until the start of vblank.

2.3.5.11 Genlock Delay Register

When the video output is genlocked, there will be a small amount of implicit delay between the genlock source and the video output. This register can be used to add a fixed amount of additional delay, up to one line. With clever programming of the video timing parameters, it is possible to have the output lead the

input. This register is only used when an external genlock source is selected in the Genlock Source field of the Output Channel Configuration register.

TABLE 22. Genlock Delay Register

Bits	Description
10:0	Number of clock cycles (bytes) of additional Genlock delay (must be > 0)

NOTE THAT WHEN A GENLOCK SOURCE IS USED, THIS REGISTER MUST NEVER BE SET TO ZERO.

When the genlock source is one of the chip's D1 ports, the delay between the output and the source is 15 clock cycles plus the Genlock Delay Register value , a minimum of 16 cycles. When the genlock source is the external hvf sync input, the delay between the output and the source is 10 clock cycles plus the Genlock Delay Register value, a minimum of 11 cycles. (This is measured from the assertion of the H bit from the sync source to the corresponding 0xff marker code of the chip's d1 output stream).

2.3.5.12 Video Hardware Configuration Register

The hardware configuration register controls the setup of hardware related to video, but which falls outside the realm of the individual input or output channels.

TABLE 23. VHW_CFG Register

Bits	Field Name	Description
0	PORT AB RESET	Reset circuitry for port AB 0 = reset 1 = enable
1	PORT AB SELECT	0= select the analog encoder (A) 1= select the primary D1 (B)
2	Port CD RESET	Reset circuitry for port CD 0 = reset 1 = enable
3	PORT CD SELECT	0= select the moosecam (C) 1= select the secondary D1 (D)
4	Audio Sync Reset	Reset circuitry for audio sync from AB/CD 0 = reset 1 = enable
6:5	Audio Sync D1 Source	Select input D1 source to use for Audio Sync 00 = none 01 = external sync source 10 = Input Port AB 11 = Input Port CD
7	Audio Sync D1 Precision	External Pixel Precision for D1 input 0 = pixels are 8-bit D1 1 = pixels are 10-bit D1
8	Audio Sync D1 Ecc	Error Correction for input D1 0 = no D1 error correction 1 = enable D1 error correction

TABLE 23. VHW_CFG Register

Bits	Field Name	Description
9	GBE Framelock Reset	Reset circuitry for GBE Framelock 0 = reset 1 = enable
11:10	GBE Framelock D1 Source	GBE Framelock Source 00 = Output Port E/F 01 = external sync source 10 = Input Port AB 11= Input Port CD
12	GBE Framelock D1 Precision	External Pixel Precision 0 = pixels are 8-bit D1 1 = pixels are 10-bit D1
13	GBE Framelock D1 Ecc	Error Correction 0 = no D1 error correction 1 = enable D1 error correction
17:14	GBE Framelock SOF Count Odd	Number of lines to wait after start of vertical blank (following active video) before checking field id bit for odd field
18	GPIBO	Controls value of GPIB output pin.
22:19	GBE Framelock SOF Count Even	Number of lines to wait after start of vertical blank (following active video) before checking field id bit for even field
31:23	RESERVED	RESERVED
35:32	Revision Code	Revision code for MACE

2.3.5.12.1 Port AB Reset / Port AB Select

Although there may be up to 4 video sources in the system (A,B,C,D), there are only 2 D1 input ports to the MACE chip (AB and CD). The muxing for this is done external to the chip. This field selects which source, A or B will be driven into the AB port. This will effect the video input channels if the input video source for the channel is AB. It also effects the video output channel if it is genlocked to the AB port. Whenever this bit is modified, it is a good idea to put the port into reset mode.

2.3.5.12.2 Port CD Reset / Port CD Select

Although there may be up to 4 video sources in the system (A,B,C,D), there are only 2 D1 input ports to the MACE chip (AB and CD). The muxing for this is done external to the chip. This field selects which source, C or D will be driven into the CD port. This will effect the video input channels if the input video source for the channel is CD. It also effects the video output channel if it is genlocked to the CD port. Whenever this bit is modified, it is a good idea to put the port into reset mode.

2.3.5.12.3 Audio Sync Reset/ Source/ D1 precision / D1 ECC

There are 2 outputs of the chip which can be used for audio sync, HSYNC1 and HSYNC2. Both of these signals will generate a signal which has the same frequency as the hsync (or hblank) for the corresponding D1 stream. HSYNC2 is generated from the D1 stream from video output Ports E/F. The timing of this signal will match that of the output video timing as programmed by the video output registers. None of the bits in this register will affect HSYNC2. HSYNC1 can be generated from 3 possible sources: either of the 2 D1 input ports (AB or CD) or the external sync (HVF) source. The Source field determines which source is used. The definition of D1 precision and D1 ecc as the same as in the VIdeo input Configuration register. If the external HVF sync source is selected then the D1 precision and D1 ecc bits are not used. Whenever any of these bits are modified, the audio reset circuitry should be in reset mode.

2.3.5.12.4 GBE Framelock Reset/ Source/ D1 precision / D1 ECC/ SOF Count

The chip can produce a GBE Framelock signal for the GBE chip which has the same frequency as the vsync (or sof) for the corresponding D1 stream. GBE Framelock can be generated from 4 possible sources: the D1 output port (EF) , either of the 2 D1 input ports (AB or CD) or the external sync (HVF) source. The Source field determines which source is used. The definition of D1 precision, D1 ecc, and SOF count as the same as in the VIdeo input Configuration register. If the external HVF sync source is selected then the D1 precision and D1 ecc bits are not used. Whenever any of these bits are modified, the GBE Framelock reset circuitry should be in reset mode.

2.3.5.12.5 GPIBO

This bit directly drives the GPIB output pin of the chip.

2.3.6 DMA Descriptors

2.3.6.1 Linear or Tiled DMA Descriptor

The DMA Descriptor contains a list of addresses specifying the addresses of the 32 pages making up the video pixel buffer. Each page table entry contains the upper 16 bits (31:16) of a 64K page real address. Note that the DMA descriptor must start on a 64 byte boundary.

Note that any descriptor that's a zero (0) indicates there is no buffer associated with these pixels. In the linear mode it constitutes the "end of field" and a video field overflow error will be posted and further pixel processing of this field will be terminated. For tiled format, then a zero (0) indicates that this size is not wanted.

TABLE 24. Linear or Tiled DMA Descriptor

Offset	63:48	47:32	31:16	15:0
0x80	Page 0	Page 1	Page 2	Page 3
0x88	Page 4	Page 5	Page 6	Page 7
0x90	Page 8	Page 9	Page 10	Page 11
0x98	Page 12	Page 13	Page 14	Page 15
0xA0	Page 16	Page 17	Page 18	Page 19
0xA8	Page 20	Page 21	Page 22	Page 23
0xB0	Page 24	Page 25	Page 26	Page 27
0xB8	Page 28	Page 29	Page 30	Page 31

2.3.6.2 UST/MSC- Unadjusted System Time/Media Stream Counter Register

[This register is actually in the Timer section of MACE. It is described here only for reference.]

The field counter is a free running counter that is updated at the end of each field. Note that the least significant bit of this register is actually a field identification bit that is a logical 0 for the odd field and a logical 1 for even field. To reset, the programmer writes the desired reset value into the register. The UST portion is a snapshot of the MACE uptime (UST) counter when Start of Field occurred. This register increments as long as there is an active video signal.

Both registers can be read with one 64 bit read operation to guarantee an autonomous values.

TABLE 25. UST/MSC- Unadjusted System Time/Media Stream Counter Register

Bits	Name	Description
0	Field ID	0= odd field 1= even field
31:1	Frame Count	
63:32	Universal System Time	Snapshot of MACE uptime (UST) counter

2.4 Revision History

2.4.1 12/01/94

1. Dropped the following pixel formats:

   VL 4:2:2:4/4:4:4:4 YUVA 8-bit per component

   RGB 3:3:2

   8-bit Y-only

   RGB 4:4:4:1

   RGBA 16-bit per component

   4:2:2:4/4:4:4:4 YUVA 16-bit per component

2. Dropped field/frame rate control.
3. Dropped gamma/de-gamma
4. Although Alpha can still be input on of the input channels, the alpha stream will not be merged with the data stream from the other channel. It will have to be done in software.
5. Eliminate programmable first coefficient for horizontal and vertical down-scaling. The first coefficient is now fixed at 1/2 the scaling ratio.
6. Changes were made to the clipping registers which does not effect chip functionality. Clipping registers now specify start and end line and pixel numbers.
7. Eliminate vertical scale output line count. The hardware does not need to have thus supplied.
8. Clamping requirements on input and output have been defined. Output will have 3 clamping options that can be set by the programmer: 1) don't clamp, 2) clamp 0 to 1 and 255 to 254, and 3) clamp data to correct YCrCb levels. Input will only be clamped so that data does not underflow or overflow the 0 to 255 range.
9. Mipmapping hardware only generates mipmaps down to 2x1. It cannot do 1x1 mipmap.
10. Allow programmer to disable D1 error correction on input, in case the Phillips chip does not correctly generate the error checking bits.

2.4.2 12/13/94

11. Merged in register descriptions from the video DMA specification.
12. Added video device interrupt mask and status registers.
13. Combined channel configurations into one register with parts applicable to input, output or both.
14. Modified field counter to be UST/MSC (this register is in the Timers section now).

2.4.3 12/19/94

15. Modified addresses to conform to MACE address map.
16. Merged HSCALE and VSCALE into FILT_IN register.
17. Swapped input descriptions of D1/digital camera and D1/analog decoder to match output D1 port.
18. Removed common device interrupt and control registers.
19. Added linear/tiled/mipmapped buffer format (with corresponding changes in the text.)
20. Merged in programming examples.

21. Rearrange specification to more readable format:
    - Overview
    - Architecture
    - Programming Interface
        - Pixel formats
        - DMA Descriptors
        - Buffer formats
        - Register Descriptions
    - Examples
    - Revision History
22. Added “figure caption” to most of the diagrams.

2.4.4 2/13/95

23. Changed LINE_SIZE to LINE_WIDTH
24. Removed (obsolete) examples
25. Updated config register
26. Merged Input and Output Filter/Scaling registers

2.4.5 2/21/95

27. Rearranged some register bits, eliminated somethings that aren’t needed, added somethings that were missing.

2.4.6 3/21/95

28. Updated register address map. Identified VHW_CFG register in Output channel.

2.4.7 3/22/95

29. Added “type” and “bits” to register summary list. Added bit fields to other registers that didn’t have them

2.4.8 3/27/95

30. Added ABGR 32 pixel format. (All ready existed, just somehow never got into spec).

2.4.9 3/29/95

31. Changed field assignment for Vend and Vblkin in VCLIP register so that it matches vhdl.
32. Reordered registers in regsiter description.

2.4.10 4/3/95

33. Slight changes to video out config register and vhw_vfg register.

2.4.11 4/6/95

34. Fixed bit fields for horizontal scaling ratio and scaling on bit.

2.4.12 5/12/95

35. Fixed bit posiitons and added missing signals to some of the registers.

2.4.13 6/13/95

36. Fixed bit positions for HPAD register.

2.4.14 9/28/95

37. Fixed bit positions for VPAD register.
38. added more descriptions for video out status and next descriptor registers

2.4.15 10/12/95

39. Clarified genlock delay register.
40. Updated video configuration register descriptions.

2.4.16 3/20/96

41. Fixed some errors in register definitions.

2.4.17 6/10/96

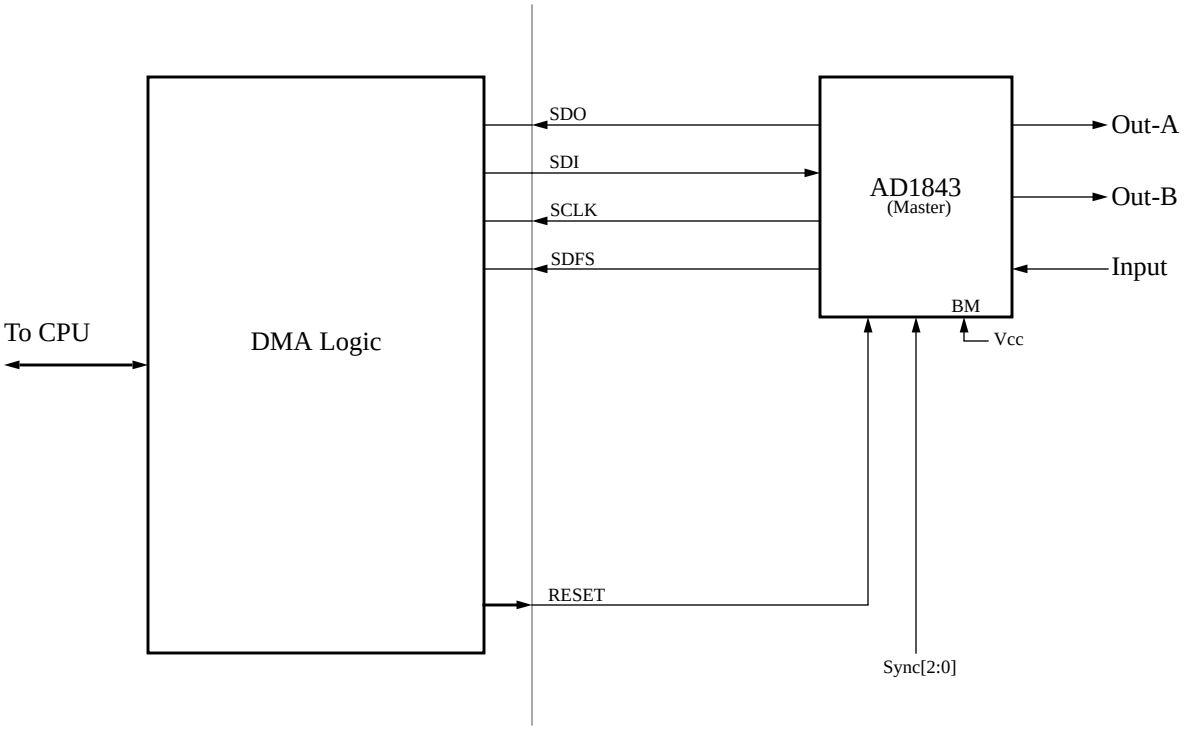
42. In VHW_CFG register: change GBE Framelock SOF count to 2 fields, one for even and one for odd.
43. InVideo Output Configuration register: change Genlock SOF count to 2 fields, one for even and one for odd.
44. Add pixel format YUVA8.
45. Change Mitchell Filter ratios.

2.4.18 7/15/96

46. Field size (video out) and line_width (video in) registers are now double buffered.
47. revision code added to hardware configuration register

3 Audio Codec Interface

The *Moosehead* system I/O asic contains a simple audio interface that supports an external stereo audio codec like the Analog Devices AD1843. The interface also contains dma ring buffers for each stereo channel. These ring buffers are stored in system main memory and system software must copy data into or out of the ring buffers if needed (i.e. no scatter/gather hardware is provided). A high level block diagram of the audio interface is shown below:



Main features:

- Support for one AD1843 Stereo Multi-media Audio Codec on a single TDM bus
- Independent rate DMA channels for each stereo input or output channel (3 total)
- Per-sample counting and 960ns resolution time stamping (MSC/UST)
- Continuous read of codecs “channel status” register over TDM bus

3.1 Audio

The audio interface provides a single TDM serial bus that is intended to support one Analog Devices AD1843 codec or similar device. The TDM bus can support other devices that use the same style of interface, but it is not the intention of this design to provide a completely flexible interface that could handle every device on the market.

The audio interface supports one stereo input channel and two stereo output channels in the maximum configuration of one codec. Each stereo channel operates independently using a time base provided by the codec that is adjustable in 1 hertz increments. This implies that the audio interface does not provide a sample rate time base of its own ala HAL2. Note that each time base can also be independently sync’d to any of the three video channels.

The audio interface provides some assistance for changing the external codec configuration through a simple register read and write interface. The hardware is given a codec register address and data value to write which is sent to the codec every TDM cycle. This also implies that the audio interface knows nothing about the internal register interface of the codecs it supports.

3.2 Register Programming Interface

The following table shows all of the audio interface registers. All bits not explicitly defined are read as zeros. All registers are defined on 64-bit aligned boundaries and can be read or written using 64-bit programmed i/o operations.

TABLE 26. Audio Interface Registers

Offset	Register Name	Type	Bits	Function
0x00	Control & Status	RW	24:0	Reset control & status register
0x08	Codec status cntl	RW	23:0	Stereo codec status address, control, write value
0x10	Codec status input mask	RW	15:0	Value to mask the read value with for interrupts
0x18	Codec status input	RO	15:0	Stereo codec last register read value
0x20	Ch1 in Ring Control	RW	10:5	Stereo input channel #1 ring buffer control
0x28	Ch1 in Read Pointer	RW	11:5	Stereo input channel #1 ring read pointer
0x30	Ch1 in Write Pointer	RW	11:5	Stereo input channel #1 ring write pointer
0x38	Ch1 in Ring Depth	RO	11:5	Stereo input channel #1 ring buffer depth
0x40	Ch2 out Ring Control	RW	10:5	Stereo output channel #2 ring buffer control
0x48	Ch2 out Read Pointer	RW	11:5	Stereo output channel #2 ring read pointer
0x50	Ch2 out Write Pointer	RW	11:5	Stereo output channel #2 ring write pointer
0x58	Ch2 out Ring Depth	RO	11:5	Stereo output channel #2 ring buffer depth
0x60	Ch3 out Ring Control	RW	10:5	Stereo output channel #3 ring buffer control
0x68	Ch3 out Read Pointer	RW	11:5	Stereo output channel #3 ring read pointer
0x70	Ch3 out Write Pointer	RW	11:5	Stereo output channel #3 ring write pointer
0x78	Ch3 out Ring Depth	RO	11:5	Stereo output channel #3 ring buffer depth

3.2.1 Interrupts

All interrupts for the audio subsystem are reported in the peripheral controller master interrupt status register (see the ISA interface chapter for details). The peripheral controller contains one master interrupt status and mask register for all of the subsystems contained within it. System software must read that interrupt status register and dispatch to the attached subsystems. Note that interrupts are still cleared within the individual subsystems.

3.2.2 Reset Control & Status Register

The following table shows the reset control and status register for the audio interface:

TABLE 27. Reset Control & Status Register Bit Fields

Bits	Reset Value	Type	Description
0	1	RW	Codec RESET control 0 - reset inactive 1 - reset signal to external codec active
1	0	RO	Codec present 0 - codec not present 1 - codec detected on the serial bus (clock active)
8:2	0	RO	Stereo input channel #1 ring write pointer alias
15:9	0	RO	Stereo output channel #2 ring read pointer alias
22:16	0	RO	Stereo output channel #3 ring read pointer alias
24-23	0	RW	Volume control positive edge latched inputs External volume push button inputs. Latched on rising edge, clear by writing zeros. Logic OR'd together to generate the level sensitive volume status change interrupt.

3.3 Codec internal register reading and writing

The codec interface includes a simple register read/write channel to be used for access to the codec's internal registers. To load a register, the system software should write the value it wants to write along with the address and write code into the address and control register. A register read operation can be performed the same way by writing the address and read code into the address and control register.

Any or all of the bits in the status word buffer can be used to generated an interrupt condition. This is controlled by a mask in the codec status input mask register.

3.3.1 Codec Read/Write Interface Registers

The following tables show the individual bits for the codec register read/write interface:

TABLE 28. Codec Status Address and Control Register

Bits	Reset Value	Type	Description
23:17	0	RW	Address of register to read or write in the codec Should be set to the channel status word when not used to reprogram the codec
16	1	RW	Read or Write operation 0 - register write 1 - register read
15:0	0	RW	Word to be sent to the codec on every TDM cycle (note: always repeats)

TABLE 29. Codec Channel Status Input Mask Register

Bits	Reset Value	Type	Description
15:0	0	RW	Mask to apply to channel status word input buffer before generating interrupt

TABLE 30. Codec Channel Status Input Word Buffer Register

Bits	Reset Value	Type	Description
15:0	0	RO	The last channel status word received from the codec

3.4 Stereo DMA MSC/UST Registers

The stereo DMA channels each have a single 64-bit register (see Timer chapter) that holds the current count of stereo samples sent or received and the time when that last sample was sent or received. The sample pair counter (MSC) is incremented every time the codec delivers or asks for a new sample pair (i.e. even when the ring buffers are full or empty), except when the DMA channel is in the RESET state. Each time one of the DMA channels processes a sample it increments the sample pair counter (MSC) and snaps the value of the 32-bit MACE uptime counter (UST). The two values can be read together using a single read operation. Note that the register is writable so that the device driver software can zero out the MSC counter if desired.

The picture below shows the format of the MSC/UST registers:

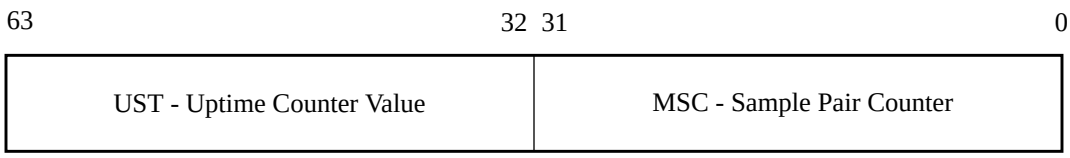


Figure: Stereo DMA MSC/UST Format

3.4.1 Stereo pair DMA registers

The following tables show the individual register bits for each stereo pair DMA channel:

TABLE 31. Channel Read Pointer Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RW	Ring buffer read pointer
4:0	0	RO	<reserved, read as zeros>

TABLE 32. Channel Write Pointer Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RW	Ring buffer write pointer
4:0	0	RO	<reserved, read as zeros>

TABLE 33. Channel Control Register

Bits	Reset Value	Type	Description
10	1	RW	RESET 0 - channel active 1 - reset channel (inactive), all registers reset, interrupt output inactive, fifos flushed, MSC does not increment.
9	0	RW	DMA enable 0 - channel disabled, but state not modified, just frozen 1 - channel enable and active (pointers must be setup)
8	0	RO	<reserved, read as zeros>
7:5	0	RW	Interrupt threshold 000 - interrupt disabled, interrupt output at inactive level 001 - interrupt on input channel ring buffer >= 25% full (< 25% for output channels) 010 - interrupt on input channel ring buffer >= 50% full (< 50% for output channels) 011 - interrupt on input channel ring buffer >= 75% full (< 75% for output channels) 100 - interrupt on ring buffer empty 101 - interrupt on ring buffer not empty 110 - interrupt on ring buffer full 111 - interrupt on ring buffer not full
4:0	0	RO	<reserved, read as zeros>

TABLE 34. Channel Current Ring Depth Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RO	Number of 32-byte blocks in the ring buffer (computed by DMA engine) Computed using a subtracter: WritePointer - Readpointer. All zeros is the empty condition, all ones is the full condition.
4:0	0	RO	<reserved, read as zeros>

3.5 Stereo Audio DMA

The audio DMA uses one ring buffer for each stereo audio input or output pair. The hardware assumes that all samples are stored in the ring buffers as 8 byte items. That way any item always starts on an 8 byte aligned boundary.

3.5.1 Stereo pair input data format

The stereo DMA input channels collect samples from the codecs and write that data out into the main memory ring buffers. The following general rules apply to the audio input DMA format:

- Each item written to main memory is a multiple of 64-bits of data
- Each 64-bit sample contains a left & right audio word that has been sign extended to a 32-bit value
- The left & right stereo samples come directly from the codec and are not modified

The picture below shows the format of a 64-bit input sample:

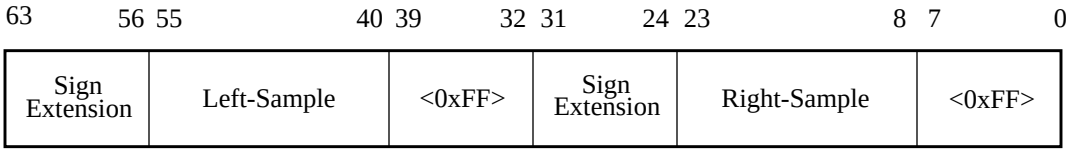


Figure: Stereo Input Pair Data Format

The stereo DMA input channels will always write four 64-bit samples to memory in one transaction.

3.5.2 Stereo pair output data format

The stereo DMA output channels collect samples from the ring buffers in main memory and write them to the codecs. The DMA engine will read from the input ring buffers as long as data is left in the ring. When data is exhausted the DMA channel outputs zeros to the codec repeatedly until more data is provided. The following general rules apply to the audio output DMA format:

- Each item read from main memory is a multiple of 64-bits of data
- Each 64-bit word contains a left and right audio sample
- The output sample format is compatible with the input format (a functional subset)
- The left & right stereo samples are sent directly to the codec
- Each 24-bit wide left/right sample is clipped (saturated) to 16-bit resolution

The picture below shows the format of a 64-bit output sample:

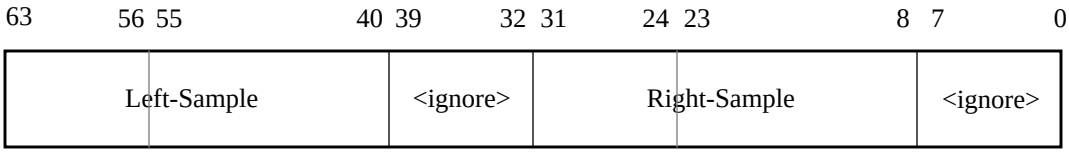


Figure: Stereo Output Pair Data Format

The stereo DMA output channels will always read four 64-bit samples from memory in one transaction. Note that the don’t care areas of the output sample are ignored by the hardware.

3.5.2.1 Output clipping/saturation

The rule that is followed for clipping the 24-bit samples to 16-bits is: if bit 31 is a ‘1’ and any of bits 30-24 are ‘0’ or if bit 31 is a ‘0’ and any of bits 30-24 are ‘1’, then saturate the output sample to the all ones value.

3.5.3 Stereo pair ring buffer

Each stereo input/output pair has a DMA ring buffer that is controlled by a set of channel registers. The size of all three of the audio ring buffers is fixed at 4Kbytes. The three ring buffers are all stored together in system main memory using a 32KB aligned base address supplied by the peripheral controller. Each 4KB ring buffer occupies one of eight 4KB pages within the 32KB range supplied. The three audio ring buffers occupy the first three 4KB pages in the 32KB block. Whenever the DMA engine reads or writes data from the ring it takes the value of the read or write pointer logic ORs it with the ring base address and ring ID to compute the memory address to use.

The address calculation is shown below:

Address[31:0] = BaseAddress[31:15] | RingID[2:0] | PointerOffset[11:5] | “00000”

Figure: Ring Buffer Address Calculation

TABLE 35. Ring ID

RingID	Ring Buffer
000	Audio input channel
001	Audio output channel #1
010	Audio output channel #2

The DMA ring buffers for the audio controller are all uni-directional. For each channel one of the two ring pointers is controlled by the hardware and one by system software. If the channel is an input channel, the DMA engine controls the ring write pointer and it is read-only. If the channel is an output channel, the DMA engine controls the ring read pointer and it is read-only. In both cases, the other pointer is controlled by system software and it is used to tell the DMA engine how full (or empty) the ring buffer is with data.

When the two pointers become equal the hardware assumes that the ring buffer is empty. When the write pointer is equal to one minus the read pointer the hardware assumes that the ring buffer is full. Note that DMA operation starts automatically as soon as system software changes it’s pointer so that the ring is no longer full/empty as long as DMA is enabled. In the case of a DMA FIFO underflow (the ring is not empty but CRIME failed to refill the outgoing data FIFO in time) the DMA channel does not start reading again even if the data eventually arrives but is late. This is a fatal error, no pointers are updated, and requires a DMA channel reset to clear.

3.5.4 Stereo output idle zero fill

For the audio interfaces, the audio output logic has a special property such that it will send zero stereo sample pairs to the codec until data is available from the DMA engine. This is true no matter what the current state of the DMA hardware. This is required so that the audio outputs remain quiet when the DMA is either inactive or not initialized.

3.6 Time Base Connections

The Analog Devices AD1843 codecs used in this design have the ability to sync their sample rate clocks to an external source such as a video input or output channel or other time base. The codec has a three external inputs that are wired up according to the diagram below:

TABLE 36. Time Base Sync Connections

Input	Connection
Sync[3]	External sync source
Sync[2]	Video input (A or B) horizontal sync
Sync[1]	Video output horizontal sync

The video input horizontal sync source comes from the MACE video back end. See that section of the spec for information on selecting which of the two video input channels to use for the Sync[2] input.

3.7 Software DMA Appendix

The audio DMA engine does not provide for scatter/gather operation, system software must arrange for the data to be copied into and out of the ring buffers. To aid in the copy operations, all dma data has been aligned on 64-bit boundaries in the ring buffers. The software device driver can use uncached load and store double instructions, R4K MIPS-III 64-bit integer ops, to copy the data words into and out of the ring buffers.

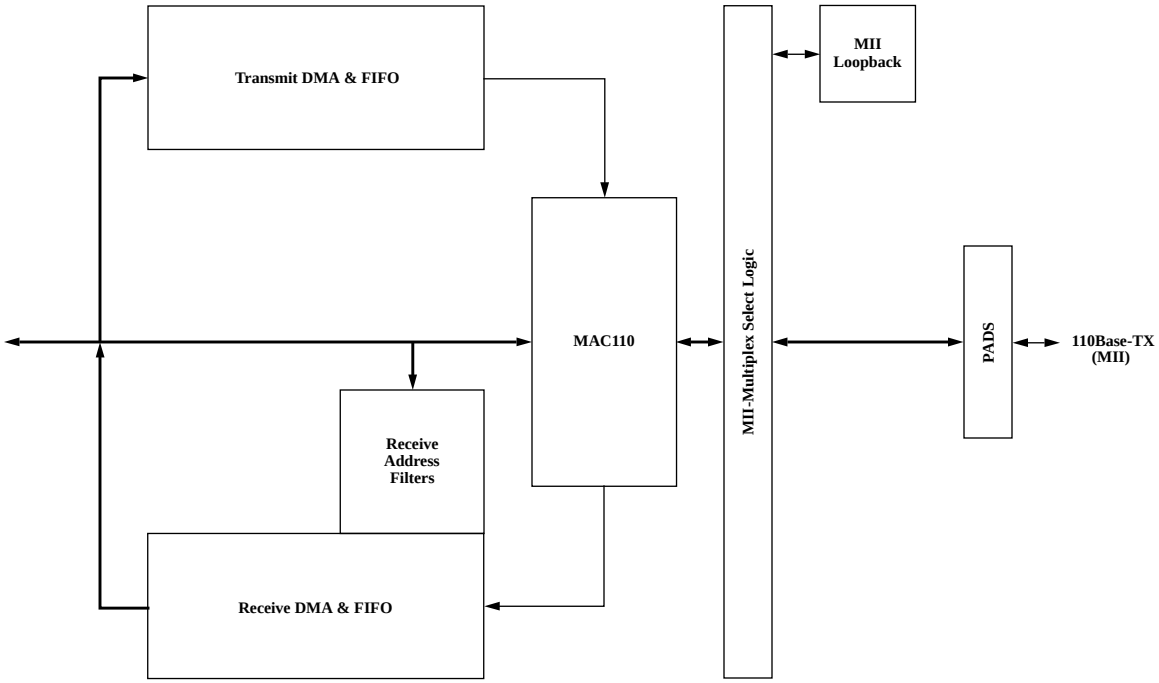
3.7.1 Sample device driver source code

The following samples are available from - rowan:/d1/moosehead/subsystem/io/doc/mace_spec

```
/*
 * Sample copy routine for audio input
 */

#include “ring.h”

audio_input(long long *src, long long *dst, int signextend)
{
        register long long mask;

        /* Setup mask for sign extension removal */
        if (signextend) {
                mask = 0xFFFFFF00FFFFFF00LL;
        } else {
                mask = 0x00FFFF0000FFFF00LL;
        }

        /* Copy everything out of the ring */
        while (count-- > 0) {
                *dst++ = *src++ & mask;
                src++;
        }
}
```

4 Fast Ethernet Interface

The *Moosehead* system I/O asic contains a simple ethernet interface that supports 10Base-T conventional ethernet and a 100Mbit TX fast ethernet connections through an MII interface. The interface also contains a small on chip message cluster address FIFO to hold the memory addresses for receive packets. The transmit and receive packets themselves are stored in system main memory. A block diagram of the ethernet interface is shown below:



Main features:

- Both 100Mbit and 10Mbit operation
- Both half-duplex CSMA-CD and full-duplex circuit switched modes
- Detailed transmit & receive status vectors for each packet
- Internet IP checksum computed for all received packets
- Multicast address filtering using 64 hash bit buckets
- Automatic transmit padding of short packets to minimum ethernet length
- Programmable receive interrupt delay timer
- Programmable receive gathering of short packets into single buffers
- Programmable inter-packet gap spacing
- Flexible transmit and receive buffering
- Functional subset of IOC3 ethernet interface

4.1 Ethernet

The ethernet interface consists of the back end MII mux, the LSI Logic Cascade-110 MAC, the receive dma section and address filtering, and the transmit dma section. The goal of this design was to have a simple hardware interface that is memory traffic efficient and one that can be implemented and verified quickly. The design also tried to stress minimum memory usage and the minimization of data copying.

4.2 Register Programming Interface

The following table shows all of the ethernet interface registers. All bits not explicitly defined are read as zeros. All registers are defined on 64-bit aligned boundaries and can be read or written using 64-bit programmed i/o operations.

TABLE 37. Ethernet Interface Registers

Offset	Register Name	Type	Bits	Function
0x00	MAC Control	RW	31:0	Ethernet MAC Control & Configuration flags
0x08	Interrupt Status	RW	30:0	Interrupt status flags
0x10	DMA Control	RW	15:0	Transmit & Receive DMA control
0x18	Timer	RW	5:0	Receive Interrupt delay timer
0x20	Transmit Interrupt Alias	WO	0:0	Alias of transmit interrupt control bits
0x28	Receive Interrupt Alias	WO	9:4	Alias of receive interrupt control bits
0x30	TX Ring buffer r/w ptr	RW	31:0	Transmit DMA ring buffer read & write pointers
0x38	Alias of above register	RW	31:0	Alias of above register
0x40	RX mcl FIFO w/r/d	RO	23:0	Receive msg cluster FIFO write/read pointer & depth
0x48	Alias of above register	RO	23:0	Alias of above register
0x50	Alias of above register	RO	23:0	Alias of above register
0x58	Interrupt Request	WO	31:0	Generate interrupt update packet (diag)
	Last Transmit Vector	RO	63:0	Status vector from last transmit packet (diag)
0x60	PHY data out	WO	15:0	PHY data out register
	PHY data in	RO	16:0	PHY data in register and busy status flag
0x68	PHY address	RW	9:0	PHY device and register address registers
0x70	PHY read start	WO	0:0	PHY read initiate operation register
0x78	Backoff	WO	10:0	Random number seed for LFSR backoff counter
0x80	Incoming msg hdr #1	RO	63:0	Incoming message queue, reserved
0x88	Incoming msg dword #1	RO	63:0	Incoming message queue, reserved
0x98	Incoming msg hdr #2	RO	63:0	Incoming message queue, reserved
0x98	Incoming msg dword #2	RO	63:0	Incoming message queue, reserved
0xA0	Physical Address	RW	47:0	Physical station address
0xA8	Secondary Physical Address	RW	47:0	Secondary physical or multicast station address
0xB0	Multicast Filter	RW	63:0	Multicast logical address filter hash mask
0xB8	Transmit Ring Base	RW	31:13	Transmit ring buffer base address in main memory
0xC0	Tx pkt #1, cmd hdr	RO	63:0	Transmit packet #1, command header word (diag)
0xC8	Tx pkt #1, cat ptr/size	RO	63:0	Transmit packet #1, concatenation buffer #1 (diag)
0xD0	Tx pkt #1, cat ptr/size	RO	63:0	Transmit packet #1, concatenation buffer #2 (diag)
0xD8	Tx pkt #1, cat ptr/size	RO	63:0	Transmit packet #1, concatenation buffer #3 (diag)
0xE0	Tx pkt #2, cmd hdr	RO	63:0	Transmit packet #2, command header word (diag)
0xE8	Tx pkt #2, cat ptr/size	RO	63:0	Transmit packet #2, concatenation buffer #1 (diag)
0xF0	Tx pkt #2, cat ptr/size	RO	63:0	Transmit packet #2, concatenation buffer #2 (diag)
0xF8	Tx pkt #2, cat ptr/size	RO	63:0	Transmit packet #2, concatenation buffer #3 (diag)
0x100 - 0x1F8	MCL Receive FIFO	RW	31:12	Receive msg cluster FIFO data port

4.2.1 Ethernet MAC Control Register

The following table shows the control and status bits for the ethernet MAC:

TABLE 38. Ethernet MAC Control & Status Register Bit Fields

Bits	Chip Reset	Type	Description
0	1	RW	Core RESET 0 - Mac110 active in run mode 1 - Global reset signal to MAC110 core is active
1	0	RW	Full duplex 0 - Disable full duplex operation 1 - Enable full duplex operation
2	0	RW	Loopback internal select 0 - Normal mode, follows 100/10 Mbit and M10T/MII select 1 - Internal loopback test mode, loops internal MII bus, selects ignored
3	0	RW	100/10 Mbit operation select 0 - 10Mbit operation 1 - 100Mbit operation
4	0	RW	M10T/MII select 0 - MII bus is selected 1 - SIA bus is selected Note: when internal loopback is selected this bit becomes the collision control for the internal looped back MII bus. Setting this bit to logic one true will cause a collision indication to be reported to the MAC. This can be used during loopback testing to force retries and transmit packet aborts.
6:5	0	RW	Destination address filter mode 0 - Accept physical station address only 1 - Accept physical, broadcast, and multicast filter matches only 2 - Accept physical, broadcast, and all multicast packets 3 - Accept any packet regardless of destination address
7	0	RW	Link Failure enable 0 - disabled 1 - hardware automatically scans for link failure condition in the external PHY
14:8	0	RW	Inter-Packet Gap IPGT Counter which sets the length of the inter-packet gap for back-to-back transmissions in the case where the MAC110 is the transmitter. Counter increments are based on twice the transmit clock period (40ns for 100Mbit and 400ns for 10Mbit operation).
21:15	0	RW	Inter-Packet Gap IPGR1 Counter which is used to reset the inter-packet gap timer when a carrier sense is detected within a short time of receiving a packet. If carrier sense goes active within two thirds the period of the IPG, the IPG counter will be reset as this carrier-sense may be because of a collision fragment.
28:22	0	RW	Inter-Packet Gap IPGR2 Counter which sets the length of the inter-packet gap for the MAC110 if it received a packet from the network and now wants to transmit a packet itself. Note that this value may need to be adjusted based on internal delays in the external PHY used.
31:29	1	RO	Implementation revision 0 - initial implementation 1 - first revision with improved transmit concatenation support

4.2.2 Ethernet Interrupt Status Register

The following table shows the individual status bits for the ethernet interrupts (CRIME IR Byte 3 / Bit 3):

TABLE 39. Ethernet Interrupt Status Register Bit Fields

Bit	MAC Reset	Type	Description
0	0	RW	TX ring empty interrupt event set 0 - no interrupt pending 1 - the transmit ring buffer is empty [read pointer = write pointer]
1	0	RW	TX packet user request interrupt event set 0 - no interrupt pending 1 - a transmit message had the interrupt request bit set, the packet has been sent
2	0	RW	TX link failure condition detected 0 - no interrupt pending 1 - the external PHY reported that the link has failed (see MAC control bit 7)
3	0	RW	TX crime memory error interrupt event set 0 - no interrupt pending 1 - a memory error occurred during a DMA transaction, DMA has stopped, fatal error
4	0	RW	TX abort interrupt event set 0 - no interrupt pending 1 - the transmitter aborted operation, DMA has stopped, fatal error, system software should examine the transmit status vector register for the abort reason
5	0	RW	RX threshold interrupt event set 0 - no interrupt pending 1 - the selected receive threshold interrupt condition is valid
6	0	RW	RX cluster FIFO underflow interrupt event set 0 - no interrupt pending 1 - the cluster FIFO was empty and new packets arrived that could not be queued
7	0	RW	RX dma FIFO overflow interrupt event set 0 - no interrupt pending 1 - the internal DMA FIFO overflowed, DMA has stopped, fatal error
12:8	0	RO	Alias of Receive mcl FIFO read-pointer
15:13	0	RO	Reserved, always zero
24:16	0	RO	Alias of Transmit ring buffer read-pointer
29:25	0	RO	Receive sequence number This is the starting sequence number for the message cluster at the queue top.
30	0	RO	Multicast hash output This is a test/debug interface bit. When RX dma enable is inactive, the output of the hash select logic is latched here (select based on DMA control 14 downto 9).
31	0	RO	Reserved, always zero

Note: interrupt bits can be cleared by system software by writing a one to the bit position. Writes of zeros to any of the interrupt status flags are ignored.

4.2.3 DMA control register

The following tables show the individual bits for the DMA control register:

TABLE 40. DMA Control Register

Bits	MAC Reset	Type	Description
15	0	RW	Receive DMA Enable 0 - channel inactive, receive interrupts masked 1 - channel active
14:12	0	RW	Receive DMA Starting Offset This is the index of the double word (64-bit word) were the DMA logic starts to fill the first receive bucket in each packet. All of the double words before the starting offset are treated as garbage padding. A starting offset of zero is not recommended.
11	0	RW	Receive Packet Gathering Enable 0 - don't gather packets 1 - attempt to gather back-to-back packets into one message cluster
10	0	RW	Receive Runt Packets Enable 0 - discard received runt packets (length < 64 bytes) 1 - receive all runt packets
9	0	RW	Receive Interrupt Enable 0 - interrupt output inactive 1 - interrupt output follows results of comparison !(Threshold != FIFO Count)
8:4	0	RW	Receive Interrupt Threshold Value to match against the message cluster FIFO counter. This is used as a water mark by the hardware. When the values are equal the interrupt output is disabled. When the values are not equal then the interrupt output is enabled. The software driver can use this register to set the desired depth for the message cluster FIFO.
3:2	0	RW	Transmit Ring Size Mask 00 - 8K byte transmit ring buffer 01 - 16K byte transmit ring buffer 10 - 32K byte transmit ring buffer 11 - 64K byte transmit ring buffer
1	0	RW	Transmit DMA Enable 0 - channel inactive, non-fatal transmit interrupts masked 1 - channel active
0	0	RW	Transmit Interrupt Enable 0 - interrupt output inactive 1 - interrupt output active if transmit ring buffer is empty (rptr = wptr)

4.2.4 Interrupt delay register

The following tables show the individual bits for the receive DMA interrupt delay register:

TABLE 41. Interrupt Delay Register

Bits	MAC Reset	Type	Description
5:0	0	RW	Interrupt Delay Count Down Value (in 30.69 microsecond increments) This timer delays the interrupt condition generated by Receive Interrupt Threshold. The rules used by the timer are as follows: when a new packet is received the timer starts to count down. When it reaches zero, the interrupt for the packets that have been collected so far is delivered to the interrupt status register. *Note that the timer will terminate early if the receive MCL FIFO drops below four entries.* Range: 0 - 2 milliseconds

4.2.5 Transmit interrupt alias register

The following table shows the individual bits for the transmit interrupt alias register:

TABLE 42. Transmit Interrupt Alias Register

Bits	MAC Reset	Type	Description
0	0	WO	Transmit Interrupt Enable 0 - interrupt output inactive 1 - interrupt output active if transmit ring buffer is empty (rptr = wptr)

4.2.6 Receive interrupt alias register

The following table shows the individual bits for the receive interrupt alias register:

TABLE 43. Receive Interrupt Alias Register

Bits	MAC Reset	Type	Description
9	0	WO	Receive Interrupt Enable 0 - interrupt output inactive 1 - interrupt output follows results of comparison !(Threshold != FIFO Count)
8:4	0	WO	Receive Interrupt Threshold Value to match against the message cluster FIFO counter. This is used as a water mark by the hardware. When the values are equal the interrupt output is disabled. When the values are not equal (i.e. some messages have been removed by the hardware) then the interrupt output is enabled. The software driver can use this register to set the desired depth for the message cluster FIFO.

4.2.7 Transmit ring buffer read & write pointer register

The transmit ring buffer read and write pointers point to the head and tail of the ring buffer. The ring buffer is considered to be empty when the two pointers are equal and the ring buffer is full when the write pointer points to the N - 1 entry in the ring buffer.

The write pointer is the only one of the two that software can write directly. The read pointer can only be reset to zero and then only by reseting the dma engine using the reset control in the dma control register.

The following table shows the individual bits for the transmit ring buffer read & write pointer register:

TABLE 44. Transmit Ring Buffer Read Pointer Register

Bits	MAC Reset	Type	Description
31:25	0	RO	Always zero
24:16	0	RO	Current DMA channel ring buffer read pointer
15:9	0	RO	Always zero
8:0	0	RW	Current DMA channel ring buffer write pointer This register is written by the software driver to queue new packets to be sent to the transmit DMA channel. For the ethernet transmit ring buffer, when the read pointer and write pointer are equal the ring buffer is empty, when the write pointer equals the read pointer minus one the ring buffer is full.

4.2.8 Receive DMA message cluster FIFO

A message cluster is a standard IRIX networking data buffer. The ethernet receiver contains a 16 entry register file that holds the base addresses of 4 kilobyte message clusters. These message clusters are always on 4 kilobyte alignments which means we only need to save 20 bits of address in a 32 bit physical address machine. Also, these buffers can be used directly by the IRIX networking code without any data copying.

The message cluster FIFO is a true first-in first-out memory. The system software would normally push base addresses into the FIFO and the receive DMA hardware would pop base addresses off of the FIFO. System software has the ability to read the FIFO, but this should never be done when the receive DMA logic is enabled.

The read index, write index, and element count registers of the FIFO are exposed to system software for use during device operation or as diagnostic registers. The registers are updated atomically by the hardware so that they are safe to read even when the receive DMA logic is operating.

Note that the contents of the register file are not initialized by the hardware at reset. Only the read index, write index, and element count register are reset to zero. Also, hardware does not prevent the system software from overrunning or underflowing the FIFO from the PIO access port. The receive DMA logic itself will never underflow the FIFO.

When system software writes a 32 bit address into the FIFO the lower 12 bits of the address are discarded. If the same address was then read back using a PIO read cycle the upper 20 bits would contain the user supplied address data while the lower 12 bits would contain zeros.

The following tables show the individual bits for the message cluster FIFO diagnostic register:

TABLE 45. Receive Message Cluster FIFO Information Register

Bits	MAC Reset	Type	Description
23:21	0	RO	Always zero
20	0	RO	Generation number
19:16	0	RO	FIFO write pointer
15:13	0	RO	Always zero
12	0	RO	Generation number
11:8	0	RO	FIFO read pointer
7:5	0	RO	Always zero
4:0	0	RO	Count of elements queued in the MCL FIFO

Note that the message cluster FIFO data port is aliased 32 times in the ethernet address space. All 32 locations perform the same function. The extra address locations do not provide any sort of index or direct addressing mechanism.

The following tables show the individual bits for the message cluster FIFO data port:

TABLE 46. Receive Message Cluster FIFO Data Port

Bits	MAC Reset	Type	Description
31:12	0	RW	Message Cluster Base Address
11:0	0	RO	Always read as zeros

4.2.9 PHY configuration bus

The ethernet interface provides the following small set of register for communication over the MDIO serial communication bus to and from an external PHY device:

TABLE 47. PHY Data I/O Register

Bits	MAC Reset	Type	Description
16	0	RO	Busy status 0 - incoming data below is valid 1 - incoming data is still being assembled and is invalid
15:0	0	RW	Data READ - incoming data values WRITE - outgoing data values

TABLE 48. PHY Address Register

Bits	MAC Reset	Type	Description
9:5	0	RW	PHY device address
4:0	0	RW	PHY internal register address

TABLE 49. PHY Read Initiate Register

Bits	MAC Reset	Type	Description
0	X	WO	Writing any value to this register initiates an external PHY read using the above addresses.

4.2.9.1 PHY Read

To read an external PHY register system software should first wait for the “Busy status” to drop to logic “0”, then write the PHY device address and internal register address desired to the PHY address register, and then write any value to the PHY read initiate register to start the read request. System software should then poll on the “Busy status” bit until it drops to a logic “0” value. The PHY data I/O register now holds the value read from the external PHY device.

4.2.9.2 PHY Write

To write an external PHY register system software should first wait for the “Busy status” to drop to logic “0”, then write the PHY device address and internal register address desired to the PHY address register, and then write the value to be written to the given register to the PHY data I/O register. System software should then poll on the “Busy status” bit until it drops to a logic “0” value to make certain that the write has completed.

4.2.10 MAC110 Backoff Seed

The following register allows software to provide a seed for the transmit backoff calculations in the MAC110:

TABLE 50. Backoff Random Number Seed Register

Bits	MAC Reset	Type	Description
10:0	X	WO	Writing a value to this register seeds the transmit function LFSR with a new pseudo-random starting value.

4.3 Receiver Address Filter Operation

The ethernet receiver filters all incoming packets by the 6 byte destination address at the start of the ethernet packet. The receive filter is controlled by the Destination Filter Mode field in the MAC control register. The receive filter provides for two physical station addresses, a 64-bit hashed multicast filter mask, and the broadcast address.

4.3.1 Physical Station Address Filter

The receive filter contains a 48-bit physical address that is the unique node address assigned by the ISO 8802-3 standard body (IEEE/ANSI 802.3) and used for internal address comparison. The physical address register contents are compared such that physical station address bit[0] is the first bit coming in from wire. Since the MAC110 core supplies data in byte at a time units, the address is compared PHYSADDR[7:0] with the first byte, PHYSADDR[15:8] with the second byte, and so on until all six bytes of the physical station address are compared.

4.3.2 Broadcast Address Filter

The receive filter contains a 48-bit broadcast address filter that watches for the ISO 8802-3 standard broadcast address of all logic 1s for internal address comparison. If the destination address at the start of the packet is all 1s, then the broadcast address filter will indicate that a broadcast address was detected.

4.3.3 Multicast Logical Address Filter

The receive filter contains a 64-bit hashed multicast mask that is used for multicast filtering. If the least significant bit of the first byte of the incoming destination address is a logic ‘1’, then the destination address is a multicast address. Every received multicast destination address is reduced to a 6-bit hash index into the multicast mask. If the bit in the mask for the selected hash index for the address is true, then the packet passes through the filter. Otherwise the packet is rejected (note: see Destination Filter Mode multicast filter enable/disable).

The 64-bit hashed multicast filter mask allows the receiver to filter out some portion of the multicast traffic, but the system software must still do an exact match to reject any unwanted multicast packets that get through the mask since the hardware mask is not a precise address filter.

4.3.3.1 Hashing Function

The 6-bit hash index into the 64-bit multicast mask register is generated by taking the 48-bit destination address and computing a 32-bit FCS CRC and then using the six most significant bits as the hash index. A ‘C’ version of this algorithm for a big endian machine is shown below:

```
static unsigned
lef_hash(unsigned char *Bytes, int BytesLength)
{
            unsigned Msb, Index, Bit, Shift, Crc = 0xFFFFFFFF;
            unsigned const Poly = 0x04c11db6;
            unsigned char CurrentByte;

            while (BytesLength-- > 0) {
                        CurrentByte = *Bytes++;
                        for (Bit = 0; Bit < 8; Bit++) {
                                    Msb = Crc >> 31;
                                    Crc <<= 1;
                                    if (Msb ^ (CurrentByte & 1)) {
                                                Crc ^= Poly;
                                                Crc |= 1;
                                    }
                                    CurrentByte >>= 1;
                        }
            }

            return Crc >> 26;
}
```

4.4 Ethernet Transmit DMA

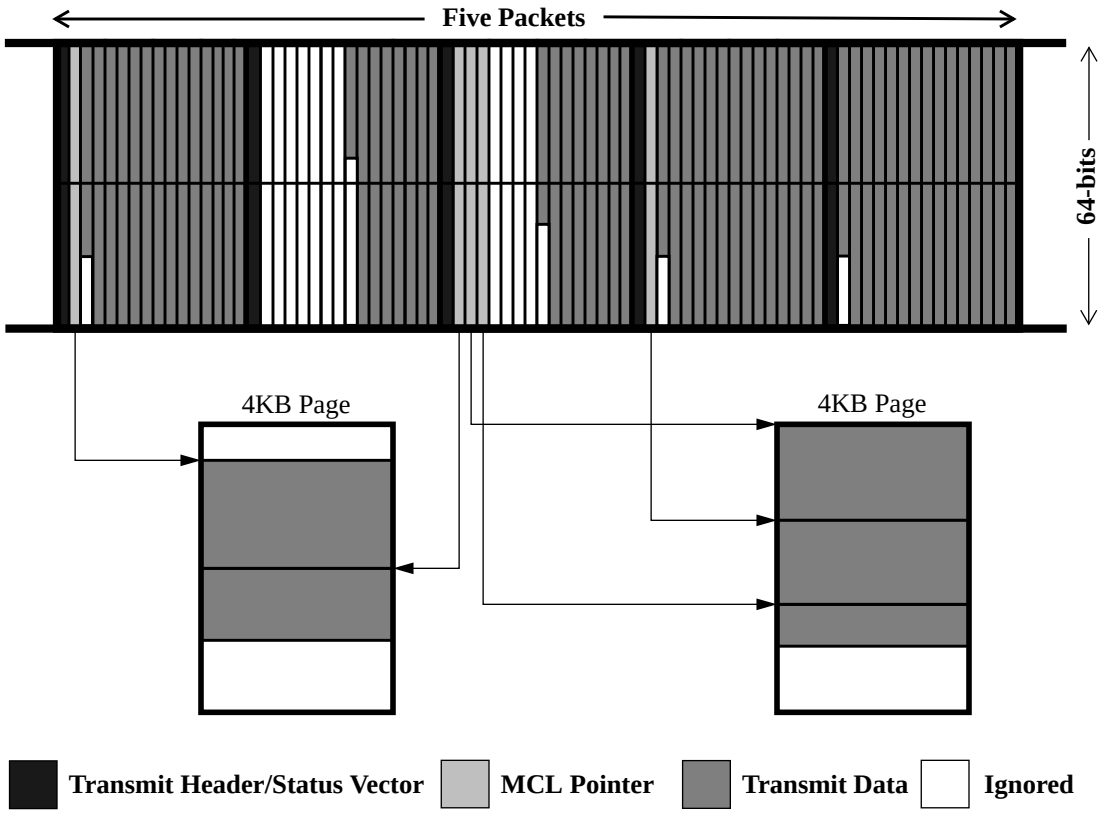
The transmit DMA channel operates off of a 64-512 entry ring buffer stored in system main memory that holds 128 byte messages. Each message contains a transmit command header, up to three 64-bit wide pointers to optional concatenation buffers, and/or up to 120 bytes of transmit packet data. The transmit command header contains the packet length and a few control flags. The concatenation pointers hold the address and length of optional buffers that contain the rest of the packet should it's length be greater than 120 bytes (i.e. it won't fit in the 120 byte ring data area).

4.4.1 Ethernet Transmit Memory Layout

The transmit DMA section reads data from the transmit ring buffer and delivers that data to the transmit back end:

- transmit message headers always start on a 128 byte alignment in the transmit ring buffer.
- the start of each transmit message consists of a 64-bit transmit command header which contains the packet byte length, any control flags, and valid flags for the concatenation buffer pointers.
- the second/third/fourth 64-bit word(s) in the transmit message are optional pointers to arbitrarily aligned data in the interior of 4KB aligned buffers that contain the remainder of the message to be concatenated onto the end of the initial data portion of the packet from the ring buffer data area.
- the initial data section of the packet must be back filled into the ring buffer data area so that the end is aligned on an 8 byte boundary. this is true even if the packet has no concatenation buffers.
- the first 64-bit word in the transmit message will be over written with the packet status vector (which is supplied by the MAC and written after packet transmission). note that the vector overwrites the length and control flags supplied initially and sets bit 63 in the packet header to logic *"1"*.

The picture below shows five packets that have been placed into the transmit ring buffer with the proper rounding. The large sixteen wide double high rectangles represent 128 byte bucket memory blocks. The packets show the spaces that need to be left for the 64-bit transmit header, the 64-bit concatenation buffer pointers, and any padding to the start offset. Note: the last packet shows that the buffer pointer bytes can be used for packet data if desired.



4.4.2 Transmit Command Header

The format of the transmit command header double word is given below:

Bits	Description
15:0	Length of valid packet data in bytes minus one
22:16	Starting byte offset of valid data in ring data block Note: minimum legal starting offset value is 8 bytes. A value of zero will cause all data in the ring header to be skipped and the first data word will come from the first concatenation pointer.
23	Terminate transmit DMA on a transmit abort condition
24	Generate user TX interrupt when packet has been sent
27-25	Concatenation pointer valid flags (buffers 3, 2, and 1)
63:28	Should be filled with zeros

4.4.3 Transmit Concatenation Pointer

The format of the transmit concatenation double word pointer is given below:

Bits	Description
2:0	Should be filled with zeros
31:3	Physical starting address of concatenation buffer data
47:32	Length of concatenation buffer data in bytes minus one Length = Length - 1;
63:48	Should be filled with zeros

4.4.4 Transmit Status Vector

The transmit status vectors are supplied by the MAC unit. The vector contains a 16-bit packet length field and 14 bits worth of status flags. The length field records the true length of the buffer in bytes. System software uses the status vector length and flags to check the success or failure to transmit a packet. The format is given below:

Bits	Description
15:0	Transmit length in bytes
16:19	Collision retry count
20	Late collision seen on at least one transmission attempt
21	CRC error seen on at least one transmission attempt
22	Packet deferred on at least one transmission attempt
23	Transmit completed successfully
24	Transmit aborted due to excessive length
25	Transmit aborted due to underrun
26	Transmit dropped due to excess collisions
27	Transmit canceled due to excessive deferral
28	Transmit dropped due to late collision
62:29	Always filled with zeros
63	Always filled with a one

4.5 Ethernet Receive DMA

The receive DMA channel operates off of a 16 entry FIFO stored inside the MACE chip that holds physical addresses of 4 kilobyte message cluster buffers. The physical addresses are loaded by the system software using programmed I/O write operations. Once an address is loaded, the DMA hardware can use that address as the physical base address of a 4KB buffer to store received ethernet packet(s). When DMA to a buffer is completed by the DMA hardware, it is popped off the bottom of the FIFO and system software will receive an interrupt notification.

The 16 entry message buffer address FIFO accepts physical addresses of 4 kilobyte buffers that are power of two aligned. The on chip read and write pointers for the FIFO are exposed to the system software for diagnostic purposes. To control the depth of the FIFO, system software can specify a threshold value from 0 to 16. Whenever the number of entries in the FIFO does not equal this number, and the receive interrupt is enabled, an interrupt request will be generated. The programmable threshold allows the software driver to use (or not use) as many entries in the FIFO as it wants.

4.5.1 Interrupt Delay Counter

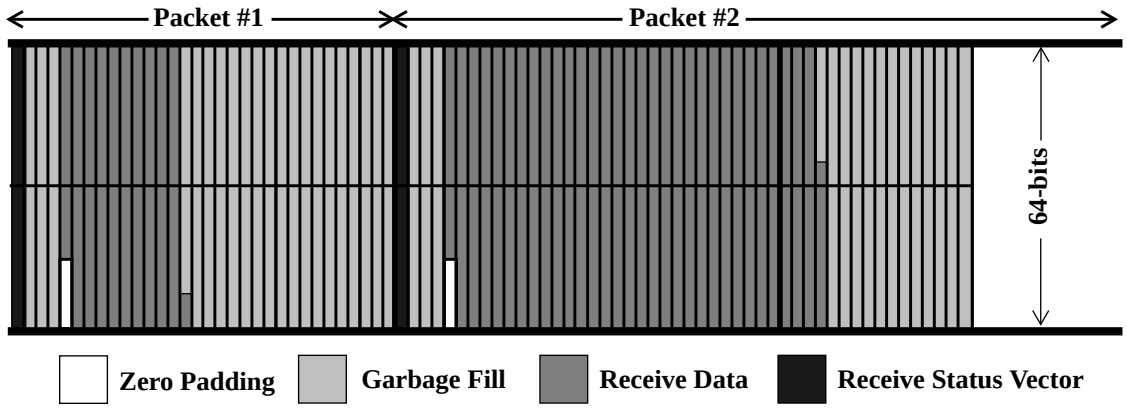
The receive DMA channel contains a 6-bit down counter that can be used to delay the delivery of a receive interrupt request by the specified number of 30.69 microsecond ticks. This is useful when the ethernet interface is operating in fast ethernet mode and the system software wants to delay interrupt delivery a short period of time to attempt to gather multiple back to back packets into a single interrupt request.

4.5.2 Data Format

The receive DMA section collects data from the address filter unit and status vectors from the MAC and writes the merged data stream into the physical data buffer in the following format:

- receive packets always start on a 256 byte alignment in the receive data buffer.
- the start of each packet contains two extra bytes of padding so that the data portion of the received ethernet packet will start on an 8 byte aligned boundary.
- the first 64-bit word in the first 256 byte bucket of each packet contains it’s status vector (which is supplied by the MAC).
- a variable number of 64-bit words can be skipped in the first 256-byte block for software headers

The picture below shows two packets that have been placed into the receive data buffer with the proper rounding. The starting receive DMA offset has been programmed to a value of four. The large thirty two wide double high rectangles represent 256 byte bucket memory blocks. The first packet shows the common case of an ethernet packet placed at the front of the message cluster buffer. The second packet shows the exception case where enough space was left in the data buffer to potentially hold another packet. This rare second case only happens when the first packet was very short (less than 256 bytes) and the interrupt delay counter had not yet expired for the first packet when the second packet started to arrive.



4.5.3 Internet Checksum

The receive DMA channel calculates a 16-bit wrap around carry sum for each packet that is received. The sum includes all valid bytes in the ethernet packet excluding the preamble. Any carry that remains after the last byte from the packet has been added into the sum is always folded back into the sum. The final sum is written out with the receive status vector. System software is responsible for subtracting unwanted header bytes and the crc from the checksum and performing any checksum validation.

4.5.4 Sequence number

The receive DMA channel stamps each packet with a sequence number that is stored in the packets status vector. This sequence number should be used by the system software driver to determine if a message cluster contains more than one complete packet. This can be done by comparing the starting sequence numbers of two consecutive message cluster buffers or by comparing the starting sequence number of the last message buffer cluster used by the DMA hardware with the sequence number provided in the interrupt status register.

4.5.5 Receive Status Vector

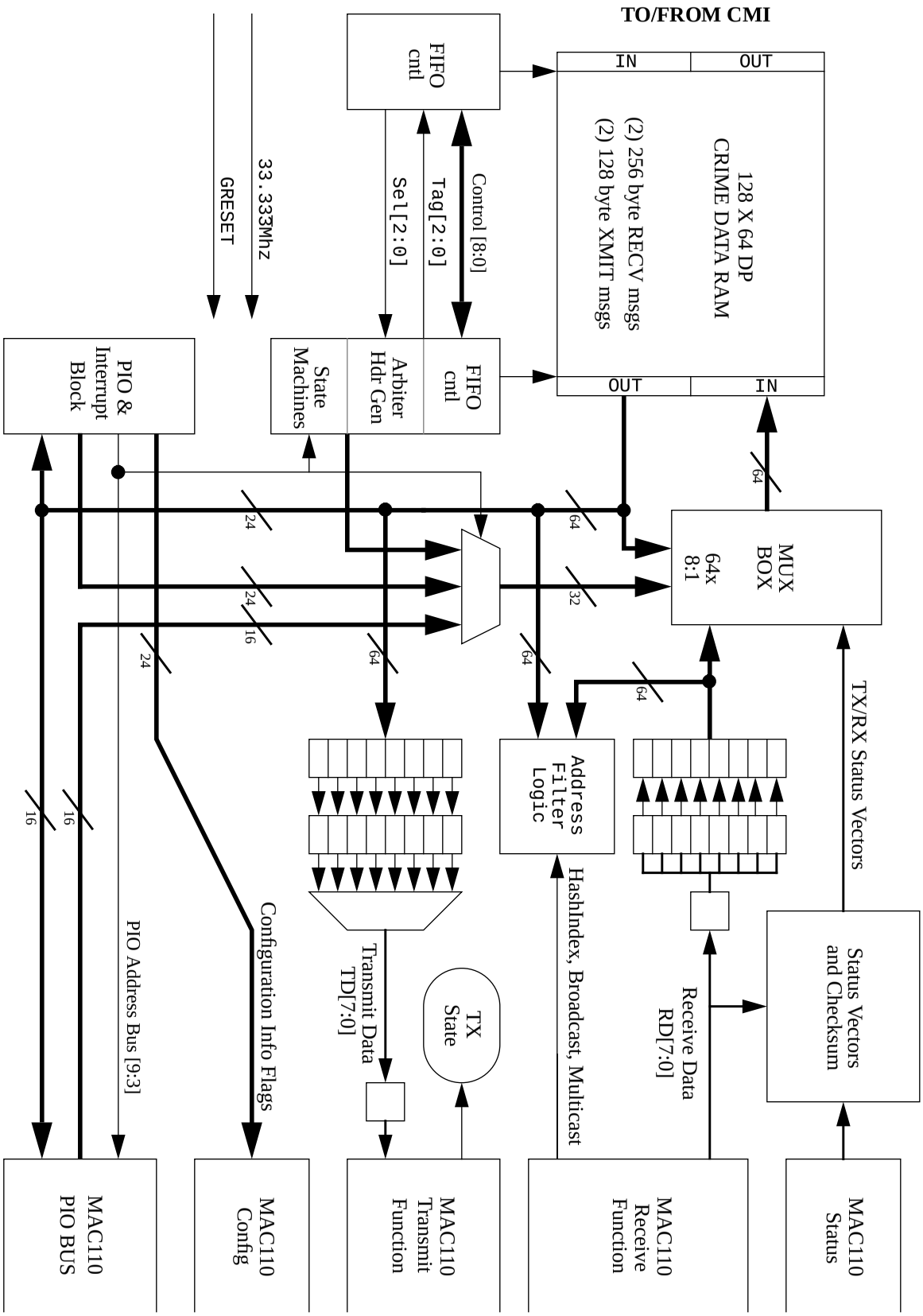
The receive status vectors are supplied by the MAC receive function. The vector contains a 16-bit packet length field, 14 bits worth of status flags, and a 16-bit Internet IP checksum. The length field records the true length of the packet in bytes. System software uses the status vector length field to determine the received packets real length.

The format of the receive status vector is given below:

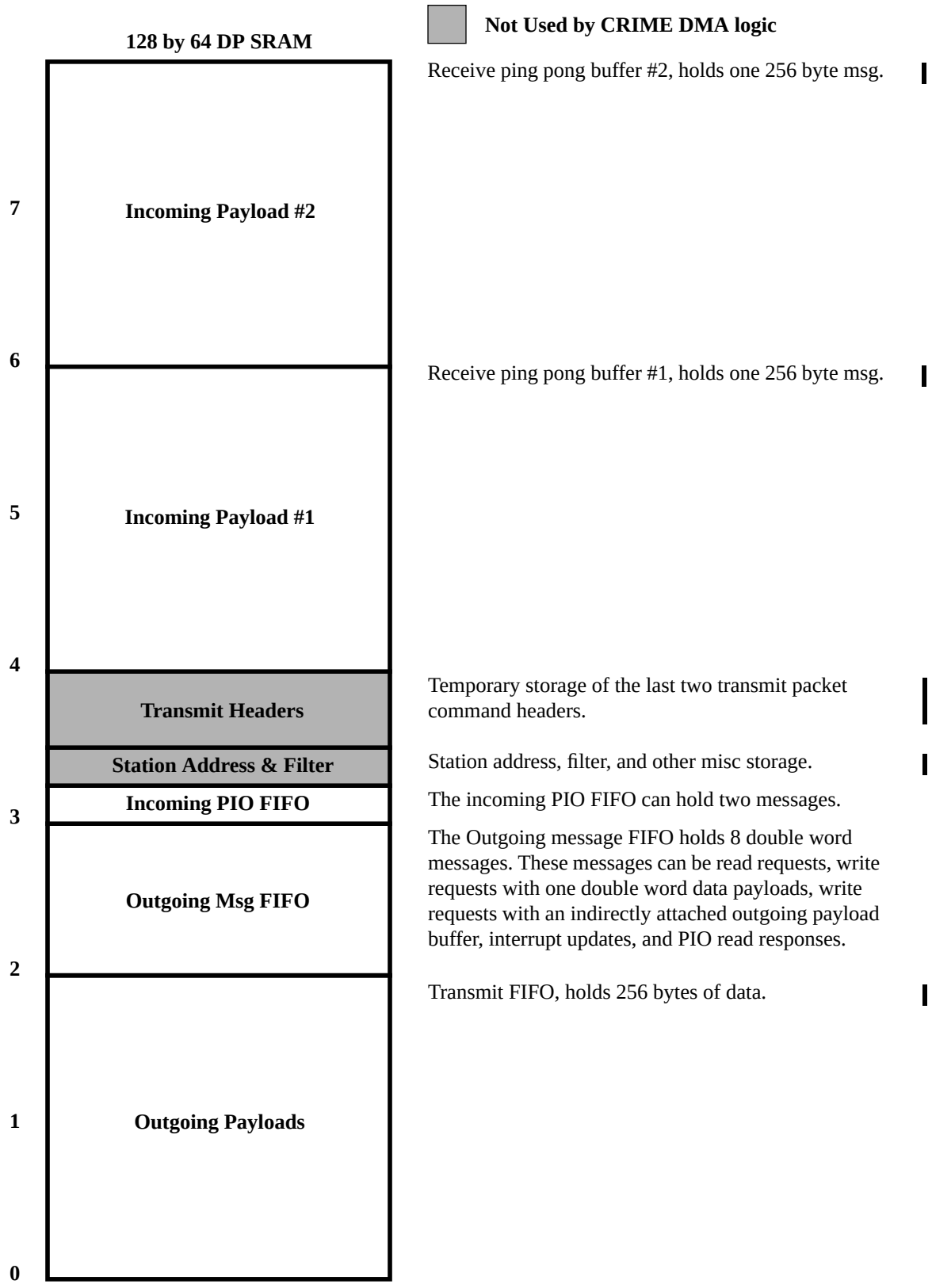
TABLE 51. Receive Status Vector Format

Bits	Description
15:0	Receive length in bytes
16	Receive code violation
17	Dribble nibble
18	CRC error
19	Multicast packet
20	Broadcast packet
21	Invalid preamble content, length or code
22	Long event previously seen
23	Received bad packet
24	Carrier event previously seen
25	Multicast filter match
26	Physical address filter match
31:27	Receive sequence number
47:32	Partial Internet IP checksum
62:48	Always filled with zeros
63	Always filled with a one

4.6 Hardware Block Diagram

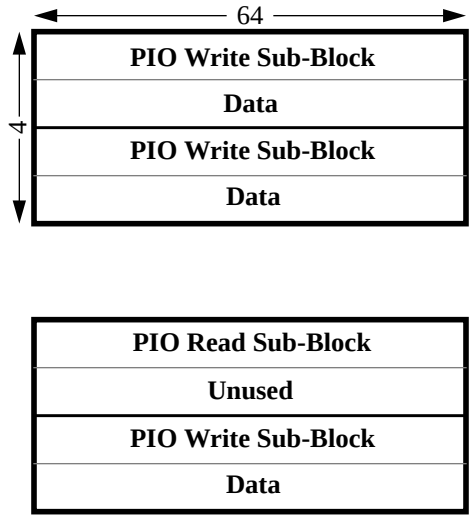


4.6.1 Internal RAM Organization



4.6.1.1 Message FIFO descriptions

Incoming message FIFO

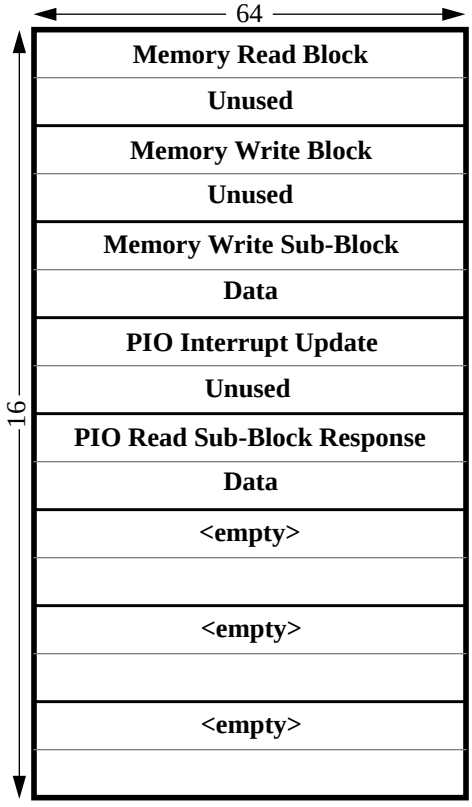


The incoming message FIFO can hold a maximum of two messages. Each message takes up two double words even if the message itself is only one double word in length.

The two examples at left show that for the ethernet interface the only messages that appear in the incoming FIFO are PIO read and write requests. The CRIME interface can post at most one read request, or two write requests, or one read and one write request.

The delivery of an incoming PIO message raises the input processing state machines request line. The state machine will then wait for the next arbitration slot. Note that if the message is a read sub-block request, the message will not be removed from the incoming FIFO until space exists in the outgoing FIFO to hold the read response.

Outgoing message FIFO



The outgoing message FIFO can hold a maximum of eight messages. Each message takes up two double words even if the message itself is only one double word in length.

The example at left shows one of each of the five message types in the outgoing FIFO with three free slots remaining.

The top two messages in the FIFO are block read and write requests. The data portions of those messages are stored in ping-pong buffers in the ethernet DP-RAM.

If the outgoing message is a memory write block request, the header comes from the outgoing message FIFO but the data portion comes from one of the ping-pong buffers. The ping-pong buffer to use is indicated both in the message tag field and on an independent tag bus that runs in parallel with the outgoing message FIFO.

If the outgoing message is a memory read block request, the data will be returned to the ping-pong buffer indicated by the message headers tag field. When the read data is delivered, the read state machine will receive a side-band notification along with a data ok/error indication.

4.7 Software DMA Appendix

For transmit packets the DMA design provides a simple gather function. Up to three optional buffers can be attached to a ring header for limited gather operation, system software must arrange for the data to be copied if the three gather elements provided are not enough to describe the packet. The software device driver can either use uncached load and store double instructions, 32-byte block cache operations, or the CRIME MTE BCOPY.

For received packets the DMA design has been done such that data will not normally need to be copied. The receive message clusters that the hardware fills can be passed directly up to the networking software in the IRIX kernel. The receive packets are also checksumed by the hardware, though system software is responsible for subtracting out any unwanted header byte from the sum (including the ethernet header).

4.7.1 Sample device driver source code

The following example source code is available from - rowan:/d1/moosehead/subsystem/io/doc/mace_spec

```
// ether.h

/*
 * Moosehead internal fast ethernet interface
 */

#ifndef SYS_MACE_ETHER_H
#define SYS_MACE_ETHER_H

#define MACE_ETHER_ADDRESS        0xBF280000

/* Ethernet interface registers */
volatile struct mac110 {
          long long   mac_control;            /* MAC mode setup */
          long long   interrupt_status;       /* Interrupt status */
          long long   dma_control;            /* DMA control */
          long long   timer;                  /* Timer */
          long long   transmit_alias;         /* Transmit interrupt (WO) */
          long long   receive_alias;          /* Receive interrupt (WO) */
          struct {
                      unsigned    _tpd;
                      unsigned    rptr:16,    /* ring buffer read pointer */
                                  wptr:16;    /* ring buffer write pointer */
          } tx_info;
#define tx_ring_rptr tx_info.rptr
#define tx_ring_wptr tx_info.wptr
          long long   pad1;
          struct {
                      unsigned    _rpd1;
                      unsigned    _rpd2:8,
                                  wptr:8,     /* MCL fifo write pointer */
                                  rptr:8,     /* MCL fifo read pointer */
                                  depth:8;    /* MCL fifo depth */
          } rx_info;
#define rx_fifo_rptr rx_info.rptr
#define rx_fifo_wptr rx_info.wptr
#define rx_fifo_depth rx_info.depth
          long long   pad2;
          long long   pad3;
          union {
                      long long   sintr_request;
```

```
                    long long   last_transmit_vector;
          } irltv;
          unsigned    __ppd1;
          unsigned    phy_dataio;             /* PHY data r/w */
          unsigned    __ppd2;
          unsigned    phy_address;            /* PHY fadr & radr */
          unsigned    __ppd3;
          unsigned    phy_read_start;         /* PHY read start */
          unsigned    __ppd4;
          unsigned    backoff;                /* Backoff LFSR */

          /* 64-bit DP-RAM locations in MACE */
          long long   msgqueue[4];            /* read-only diag */
          long long   physaddr;               /* Physical address */
          long long   secphysaddr;            /* Physical address #2 */
          long long   mlaf;                   /* Multicast filter */
          long long   tx_ring_base;           /* Transmit ring base */
          long long   tx1_cmd_hdr;            /* read-only diag */
          long long   tx1_cat_ptr1;           /* read-only diag */
          long long   tx1_cat_ptr2;           /* read-only diag */
          long long   tx1_cat_ptr3;           /* read-only diag */
          long long   tx2_cmd_hdr;            /* read-only diag */
          long long   tx2_cat_ptr1;           /* read-only diag */
          long long   tx2_cat_ptr2;           /* read-only diag */
          long long   tx2_cat_ptr3;           /* read-only diag */

          /* 64-bit DP-RAM FIFO locations in MACE */
          unsigned    _rpd;
          unsigned    rx_fifo;
          long long   reserved5[31];
};

/* Multicast Logical Address Filter Macros */
#define LAF_TSTBIT(laf, bit) \
          ((laf) & (1LL << ((bit) & 0x3F)))
#define LAF_SETBIT(laf, bit) \
          ((laf) |= (1LL << ((bit) & 0x3F)))
#define LAF_CLRBIT(laf, bit) \
          ((laf) &= ~(1LL << ((bit) & 0x3F)))

/* MAC Control Register */
#define MAC_RESET                 0x0001
#define MAC_FULL_DUPLEX           0x0002
#define MAC_LOOPBACK              0x0004
#define MAC_100MBIT               0x0008
#define MAC_SIA                   0x0010
#define MAC_FILTER                0x0060
#define   MAC_PHYSICAL                        0x0000
#define   MAC_NORMAL                          0x0020
#define   MAC_ALL_MULTICAST                   0x0040
#define   MAC_PROMISCOUS                      0x0060
#define MAC_LINKF                 0x0080
#define MAC_IPG                   0x1FFFF00
#define MAC_IPGT_SHIFT            8
```

```
#define MAC_IPGR1_SHIFT           15
#define MAC_IPGR2_SHIFT           22
#define MAC_DEFAULT_IPG           0x54A9500   /* 21, 21, 21 */
#define MAC_REV_SHIFT             29

/* Interrupt Status Register */
#define INTR_TX_DMA_REQ           0x01
#define INTR_TX_PKT_REQ           0x02
#define INTR_TX_LINK_FAIL         0x04
#define INTR_TX_MEMORY_ERROR      0x08
#define INTR_TX_ABORTED           0x10
#define ETHER_TX_ERRORS           (INTR_TX_LINK_FAIL | \
                                            INTR_TX_MEMORY_ERROR | \
                                            INTR_TX_ABORTED)
#define INTR_RX_DMA_REQ           0x20
#define INTR_RX_MSGS_UNDERFLOW    0x40
#define INTR_RX_FIFO_OVERFLOW     0x80
#define ETHER_RX_ERRORS           (INTR_RX_MSGS_UNDERFLOW | \
                                             INTR_RX_FIFO_OVERFLOW)

/* DMA control register */
#define DMA_TX_INT_EN             0x0001
#define DMA_TX_DMA_EN             0x0002
#define DMA_TX_RINGMSK            0x000c
#define DMA_TX_8K                 0x0000
#define DMA_TX_16K                0x0004
#define DMA_TX_32K                0x0008
#define DMA_TX_64K                0x000c
#define DMA_TX_RINGMSK_SHIFT      2
#define DMA_RX_THRSHD             0x01f0
#define DMA_RX_INT_EN             0x0200
#define DMA_RX_RUNTS_EN           0x0400
#define DMA_RX_GATHER_EN          0x0800
#define DMA_RX_OFFSET             0x7000
#define DMA_RX_OFFSET_SHIFT       12
#define DMA_RX_DMA_EN             0x8000

/* Phy MDIO interface busy flag */
#define MDIO_BUSY                 0x10000

/* Statistics vector format */
typedef long longstatistics_vector_t;

/* Receive message cluster FIFO control */
#define ETHER_RX_DMA_ENABLE       0x8000
#define ETHER_RX_DMA_OFFSET       0x7000
#define ETHER_RX_OFFSET_SHIFT     12
#define ETHER_RX_MERGE_ENABLE     0x0800
#define ETHER_RX_RUNT_ENABLE      0x0400
#define ETHER_RX_INTR_ENABLE      0x0200
#define ETHER_RX_THRESHOLD        0x01F0
#define ETHER_RX_THRESH_SHIFT     4

/* Transmit message cluster FIFO control */
```

```
#define ETHER_TX_RING_SIZE        0x000C
#define ETHER_TX_DMA_ENABLE       0x0002
#define ETHER_TX_INTR_ENABLE      0x0001

/* Receive status vector */
#define RX_VEC_LENGTH             0x00007FFF
#define RX_VEC_CODE_VIOLATION     0x00010000
#define RX_VEC_DRIBBLE_NIBBLE     0x00020000
#define RX_VEC_CRC_ERROR          0x00040000
#define RX_VEC_MULTICAST          0x00080000
#define RX_VEC_BROADCAST          0x00100000
#define RX_VEC_INVALID_PREAMBLE   0x00200000
#define RX_VEC_LONG_EVENT         0x00400000
#define RX_VEC_BAD_PACKET         0x00800000
#define RX_VEC_CARRIER_EVENT      0x01000000
#define RX_VEC_MULTICAST_MATCH    0x02000000
#define RX_VEC_PHYSICAL_MATCH     0x04000000
#define RX_VEC_RECEIVE_SEQNUM     0xF8000000
#define RX_VEC_RECEIVE_SEQNUM_SHIFT 27
#define RX_VEC_FINISHED           0x80000000000000ll
#define RX_PROMISCUOUS \
          (RX_VEC_BROADCAST|RX_VEC_MULTICAST_MATCH|RX_VEC_PHYSICAL_MATCH)
#define RX_VEC_CKSUM_SHIFT        32

/* Transmit command header */
#define TX_CMD_LENGTH             0x00007FFF
#define TX_CMD_OFFSET             0x007F0000
#define TX_CMD_OFFSET_SHIFT       16
#define TX_CMD_TERM_DMA           0x00800000
#define TX_CMD_SENT_INT_EN        0x01000000
#define TX_CMD_CONCAT_1           0x02000000
#define TX_CMD_CONCAT_2           0x04000000
#define TX_CMD_CONCAT_3           0x08000000
#define TX_CMD_NUM_CATS           3

/* Transmit status vector */
#define TX_VEC_LENGTH             0x00007FFF
#define TX_VEC_COLLISIONS         0x000F0000
#define TX_VEC_COLLISION_SHIFT    16
#define TX_VEC_LATE_COLLISION     0x00100000
#define TX_VEC_CRC_ERROR          0x00200000
#define TX_VEC_DEFERRED           0x00400000
#define TX_VEC_COMPLETED_SUCCESSFULLY 0x00800000
#define TX_VEC_ABORTED_TOO_LONG   0x01000000
#define TX_VEC_ABORTED_UNDERRUN   0x02000000
#define TX_VEC_DROPPED_COLLISIONS 0x04000000
#define TX_VEC_CANCELED_DEFERRAL  0x08000000
#define TX_VEC_DROPPED_LATE_COLLISION 0x10000000
#define TX_VEC_FINISHED           0x80000000000000ll

/* PHY defines */
#define PHY_QS6612X               0x0181441   /* Quality TX */
#define PHY_ICS1889               0x0015F41   /* ICS FX */
#define PHY_ICS1890               0x0015F42   /* ICS TX */
```

```
#define PHY_DP83840             0x20005C0   /* National TX */
#define PHY_PCTL_RESET          0x8000      /* RESET */
#define PHY_PCTL_LOOPBACK       0x4000      /* Loopback */
#define PHY_PCTL_RATE           0x2000      /* 100Mbits */
#define PHY_PCTL_AN_ENABLE      0x1000      /* AN enable */
#define PHY_PCTL_POWERDOWN      0x0800      /* Powerdown PHY */
#define PHY_PCTL_ISOLATE        0x0400      /* MII isolate */
#define PHY_PCTL_RESTART_AN     0x0200      /* Restart AN */
#define PHY_PCTL_DUPLEX         0x0100      /* Full duplex */
#define PHY_PCTL_COLL_TEST      0x0080      /* Full duplex */
#define PHY_PMSR_ANC            0x0020      /* PHY (1:5) */
#define PHY_PMSR_FAULT          0x0010      /* PHY (1:6) */
#define PHY_PMSR_ANA            0x0008      /* PHY (1:3) */
#define PHY_PMSR_LINK           0x0004      /* PHY (1:2) */
#define PHY_PMSR_JABBER         0x0002      /* PHY (1:1) */
#define PHY_PMSR_ECAP           0x0001      /* PHY (1:0) */
#define PHY_PLPA_TAF4           0x0200      /* PHY (5:9) */
#define PHY_PLPA_TAF3           0x0100      /* PHY (5:8) */
#define PHY_PLPA_TAF2           0x0080      /* PHY (5:7) */
#define PHY_PLPA_TAF1           0x0040      /* PHY (5:6) */
#define PHY_PLPA_TAF0           0x0020      /* PHY (5:5) */

#endif

// ether.c

/*
 * Moosehead MACE 10/100 Mbit/s Fast Ethernet Interface Driver
 *
 * Copyright 1995, Silicon Graphics, Inc. All rights reserved.
 */

#ident “$Revision: 1.14 $”

#include “sys/types.h”
#include “sys/param.h”
#include “sys/systm.h”
#include “sys/sysmacros.h”
#include “sys/cmn_err.h”
#include “sys/debug.h”
#include “sys/edt.h”
#include “sys/errno.h”
#include “sys/immu.h”
#include “sys/invent.h”
#include “sys/kopt.h”
#include “sys/mbuf.h”
#include “sys/sbd.h”
#include “sys/socket.h”
#include “sys/cpu.h”
#include “net/if.h”
#include “net/raw.h”
#include “net/soioctl.h”
#include “misc/ether.h”
#include “sys/kmem.h”
```

```
#include “netinet/in.h”
#include “netinet/in_systm.h”
#include “netinet/if_ether.h”
#include “netinet/ip.h”
#include “string.h”
#include “sys/idbgentry.h”
#include “sys/if_me.h”
#include “sys/atomic_ops.h”

/* config from master.d */
extern int me_hdwrcksum_enable;
extern int me_hdwrgather_enable;
extern int me_rxdelay;
extern int me_fullduplex_ipg[];
extern int me_halfduplex_ipg[];
extern struct phyerrata me_phyerrata[];

/* General MACE ethernet hardware defines */
#define ETHER_RX_BLOCK_SIZE    256
#define RCVBUF_SIZE            4096
#define TX_RING_SIZE           64
#define MSGCL_FIFO_SIZE        16
#define ETHER_HDRLEN           14
#define CRCLEN                 4
#define ETHERMAXLEN            1536
#define ETHERMINLEN            64
#define DMA_PADDING            2

#define BLOCKROUND(x, blk)     (((x) + (blk) - 1) & ~((blk) - 1))
#define FIFOINDEX(x,y)         ((x) & ((y) - 1))
#define RXRINGINDEX(x)         ((x) & (MSGCL_FIFO_SIZE - 1))
#define RXFIFOINDEX(x)         ((x) & ((MSGCL_FIFO_SIZE * 2) - 1))
#define TXFIFOINDEX(x)         ((x) & (TX_RING_SIZE - 1))

#define ei_ac          eif.eif_arpcom  /* common arp stuff */
#define ei_if          ei_ac.ac_if     /* network-visible interface */
#define ei_rawif       eif.eif_rawif   /* raw domain interface */

#define rx_controlreceive_alias
#define tx_controltransmit_alias

/* TX fifo entry */
typedef union{
 unsigned long long TXCmd;
 unsigned long long TXStatus;
 unsigned long long TXConcatPtr[4];
 unsigned long long TXData[16];
 char buf[128];
} TXfifo;

/* RX and TX buffers */
typedef union{
 unsigned long long RXStatus;
 char buf[2048];
```

```
} RXbuf;

typedef union{
 unsigned long long TXStatus;
 char buf[2048];
} TXbuf;

#define DMA_RX_PAD2+8          /* RX padding at front of buffer */

#define PHYS_START0x1
#define PHYS_WASUP0x2
#define PHYS_WASDOWN0x4
#define PHYS_UPDATE0x8

static struct maceif {
        /* Common Ethernet interface */
        struct etherif         eif;

        /* Multicast control */
        u_int                  lafcoll;
        u_int                  nmulti;
        long long              mlaf;

        /* Hardware structures */
        volatile struct mac110*mac;

        /* Operations */
        u_int                  mode;
        char                   revision;

        /* Phy xcvr info */
        signed char            phyaddr;
        char                   phyrev;
        char                   phystatus;
        int                    phytype;

        /* Transmit ring buffer */
        short                  tx_rptr, tx_wptr;
        int                    tx_free_space;
        struct mbuf            *tx_mfifo[TX_RING_SIZE];
        volatile TXfifo        *tx_fifo;

        /* Performance */
        int                    tcase[8];

        /* Receive message cluster FIFO */
        short                  rx_rptr, rx_rlen;
        struct mbuf            *rx_mfifo[MSGCL_FIFO_SIZE];
        int                    rx_boffset;

        /* Statistics */
        int                    tx_ring_errors;
        int                    rx_fifo_errors;
```

```
        /* TX link stats */
        int                     tx_late_collisions;
        int                     tx_crc_error;
        int                     tx_deferred;
        int                     tx_aborted_too_long;
        int                     tx_aborted_underrun;
        int                     tx_dropped_collisions;
        int                     tx_canceled_deferral;
        int                     tx_dropped_late_collision;

        /* RX link stats */
        int                     rx_octets_recv;
        int                     rx_code_violation;
        int                     rx_dribble_nibble;
        int                     rx_crc_error;
        int                     rx_multicast;
        int                     rx_broadcast;
        int                     rx_total_recv;
        int                     rx_invalid_preamble;
        int                     rx_long_event;
        int                     rx_carrier_event;
} mace_ether;

static int mace_ether_init(struct etherif *, int);
static void mace_ether_reset(struct etherif *);
static void mace_ether_watchdog(int);
static int mace_ether_output(struct etherif *, struct etheraddr *,
        struct etheraddr *, u_short, struct mbuf *);
static int mace_ether_ioctl(struct etherif *, int, void *);

static struct etherifops meops = {
        mace_ether_init, mace_ether_reset, mace_ether_watchdog,
        mace_ether_output,
        (int (*)(struct etherif *, int, void *))mace_ether_ioctl
};

static void mace_ether_intr(int);
static void mace_ether_receive(struct maceif *, int, int);
static void mace_ether_transmit_complete(struct maceif *);
static void mace_ether_dump(int);


/*
 * Read a phy register over the MDIO bus
 */
static int
mace_ether_mdio_rd(register struct maceif *mif, int fireg)
{
        volatile struct mac110 *mac = mif->mac;
        volatile int rval;

        mac->phy_address = (mif->phyaddr << 5) | (fireg & 0x1f);
        mac->phy_read_start = fireg;
        us_delay(25);
```

```
        while ((rval = mac->phy_dataio) & MDIO_BUSY) {
                us_delay(25);
        }

        return rval;
}

/*
 * Write a phy register over the MDIO bus
 */
static int
mace_ether_mdio_wr(register struct maceif *mif, int fireg, int val)
{
        volatile struct mac110 *mac = mif->mac;

        mac->phy_address = (mif->phyaddr << 5) | (fireg & 0x1f);
        mac->phy_dataio = val;
        us_delay(25);

        return val;
}


/*
 * Modify phy register using given mask and value
 */
static void
mace_ether_mdio_rmw(register struct maceif *mif, int fireg, int mask, int val)
{
        register int rval;

        rval = mace_ether_mdio_rd(mif, fireg);
        rval = (rval & ~mask) | (val & mask);
        mace_ether_mdio_wr(mif, fireg, rval);
}

/*
 * Process ERRATA data for the PHY found on the MDIO bus
 */
static void
mace_ether_mdio_errata(register struct maceif *mif)
{
        register struct phyerrata *pe;

        for (pe = me_phyerrata; pe->type != 0; ++pe) {
                if (pe->type != mif->phytype)
                        continue;
                if (pe->rev != mif->phyrev)
                        continue;
                mace_ether_mdio_rmw(mif, pe->reg, pe->mask, pe->val);
        }
}

/*
```

```
 * Probe the management interface for PHYs
 */
static int
mace_ether_mdio_probe(register struct maceif *mif)
{
        register int i, val, p2, p3;

        /* already found the phy? */
        if ((mif->phyaddr >= 0) && (mif->phyaddr < 32))
                return mif->phytype;

        /* probe all 32 slots for a known phy */
        for (i = 0; i < 32; ++i) {
                mif->phyaddr = (char)i;
                p2 = mace_ether_mdio_rd(mif, 2);
                p3 = mace_ether_mdio_rd(mif, 3);
                val = (p2 << 12) | (p3 >> 4);
                switch (val) {
                 case PHY_QS6612X:
                 case PHY_ICS1889:
                 case PHY_ICS1890:
                 case PHY_DP83840:
                        mif->phyrev = p3 & 0xf;
                        mif->phytype = val;
                        return val;
                }
        }
        mif->phyaddr = -1;

        return -1;
}

/*
 * Update our mode to match external xvcr
 *
 * Note: if the partner doesn't support fast-link pulse auto-negotiation,
 * we just assume half-duplex mode to be safe.
 */
static void
mace_ether_link_update(register struct maceif *mif, int msr)
{
        register int mode, p5, val = 0;

        /*
         * If auto-negotiation complete, pick up result and
         * set our operating mode accordingly
         */
        if ((msr & PHY_PMSR_ANC) && ((mif->phystatus & PHYS_UPDATE) == 0)) {
                /* read ANLPAR register */
                p5 = mace_ether_mdio_rd(mif, 5);
                if (p5 & PHY_PLPA_TAF4) {
                        val |= MAC_100MBIT;     /* 100Mb-T4 */
                } else if (p5 & PHY_PLPA_TAF3) {
                        val |= MAC_100MBIT | MAC_FULL_DUPLEX;/* 100Mb-TX */
```

```
                } else if (p5 & PHY_PLPA_TAF2) {
                        val |= MAC_100MBIT;     /* 100Mb-HD */
                } else if (p5 & PHY_PLPA_TAF1) {
                        val |= MAC_FULL_DUPLEX; /* 10Mb-FD */
                }

                /* update mac mode */
                mode = mif->mode;
                mode &= ~(MAC_IPG | MAC_100MBIT | MAC_FULL_DUPLEX);
                mode |= val;

                /* set ipg based on full or half duplex */
                if (val & MAC_FULL_DUPLEX) {
                 if (me_fullduplex_ipg[0]) {
                        mode |= me_fullduplex_ipg[0] << MAC_IPGT_SHIFT;
                        mode |= me_fullduplex_ipg[1] << MAC_IPGR1_SHIFT;
                        mode |= me_fullduplex_ipg[2] << MAC_IPGR2_SHIFT;
                 } else {
                        mode |= MAC_DEFAULT_IPG;
                 }
                } else {
                 if (me_halfduplex_ipg[0]) {
                        mode |= me_halfduplex_ipg[0] << MAC_IPGT_SHIFT;
                        mode |= me_halfduplex_ipg[1] << MAC_IPGR1_SHIFT;
                        mode |= me_halfduplex_ipg[2] << MAC_IPGR2_SHIFT;
                 } else {
                        mode |= MAC_DEFAULT_IPG;
                 }
                }
                mif->mode = mode;
                mif->mac->mac_control = mode;
                mif->mac->dma_control |= ETHER_TX_DMA_ENABLE;

                /* done */
                mif->phystatus |= PHYS_UPDATE;
        }

        return;
}

void
if_meedtinit(struct edt *edtp)
{
        struct maceif *mif = &mace_ether;
        struct etheraddr ea;
        register int pfn, unit = 0;
        static int mainit;
        extern char eaddr[];

        /* once only */
        if (mainit) {
                return;
        }
        mainit = 1;
```

```
        /* should we bother to probe? */
        mif->mac = (volatile struct mac110 *)MACE_ETHER_ADDRESS;

        /* get ethernet address from system */
        bcopy(eaddr, ea.ea_vec, ETHERADDRLEN);

        /* print BSD style device present message on console? */
        if (showconfig) {
                printf("ec%d: hardware ethernet address %s\n",
                        unit, ether_sprintf(ea.ea_vec));
        }

        /* init filters */
        mif->mac->physaddr = 0;
        mif->mac->secphysaddr = 0;
        mif->mac->mlaf = 0;
        mif->mlaf = 0;

        /* get MACE ethernet hardware revision */
        mif->revision = mif->mac->mac_control >> MAC_REV_SHIFT;

        /* don't know phy address */
        mif->phyaddr = -1;

        /* register ethernet interface */
        mif->eif.eif_private = (caddr_t)mif;
        ether_attach(&mif->eif, "ec", unit, (caddr_t)mif,
                &meops, &ea, INV_ETHER_EC, mif->revision);
        idbg_addfunc("me_dump", (void (*)())mace_ether_dump);

        /* attach to IP32 interrupt dispatch core */
        if (setcrimevector(MACE_INTR(3), SPL5,
                        (void(*)())mace_ether_intr, (int)mif, 0))
                cmn_err(CE_ALERT, "ec0: could not set interrupt vector");

        /* create transmit fifo */
        pfn = contmemall(2, 2, VM_DIRECT|VM_NOSLEEP);
        mif->tx_fifo = (TXfifo *)small_pfntova_K1(pfn);
}

static void
mace_hdwrether_init(struct maceif *mif)
{
        register struct mbuf *m;
        register int boffset, i;

        /* reset the ethernet */
        mif->mac->mac_control = MAC_RESET;
        mif->mac->mac_control = 0;

        /* set operating mode */
        mif->mac->mac_control = mif->mode;
```

```
        /* initialize transmit fifo */
        bzero((void *)mif->tx_fifo, TX_RING_SIZE * sizeof (TXfifo));
        mif->mac->tx_ring_base = kvtophys((void *)mif->tx_fifo);
        mif->tx_rptr = mif->tx_wptr = 0;
        mif->tx_free_space = TX_RING_SIZE;

        /* free any previously queued tx buffers (toss-em) */
        for (i = 0; i < TX_RING_SIZE; i++) {
                m_freem(mif->tx_mfifo[i]);
                mif->tx_mfifo[i] = NULL;
        }

        /* set watchdog */
        mif->ei_if.if_timer = IFNET_SLOWHZ;

        /* calculate proper receive fill offset */
        boffset = (sizeof (struct etherbufhead) -
                 sizeof (struct ether_header)) / sizeof (long long);

        /* setup receive message cluster list */
        mif->rx_rptr = 0;
        mif->rx_rlen = MSGCL_FIFO_SIZE;
        mif->rx_boffset = boffset * sizeof (long long);
        mif->mac->timer = me_rxdelay;
        for (i = 0; i < mif->rx_rlen; i++) {
                if ((m = m_vget(M_DONTWAIT, RCVBUF_SIZE, MT_DATA)) == NULL)
                        panic("m_vget failed");
                dki_dcache_inval(mtod(m, void *), RCVBUF_SIZE);
                mif->mac->rx_fifo = kvtophys(mtod(m, caddr_t));
                m_freem(mif->rx_mfifo[i]);
                mif->rx_mfifo[i] = m;
        }

        /* set DMA control bits */
        mif->mac->dma_control =
                ETHER_TX_DMA_ENABLE |
                ETHER_RX_DMA_ENABLE |
                ETHER_RX_INTR_ENABLE |
                (boffset << ETHER_RX_OFFSET_SHIFT) |
                (mif->rx_rlen << ETHER_RX_THRESH_SHIFT);
}

static int
mace_ether_init(struct etherif *eif, int flags)
{
        register struct maceif *mif = (struct maceif *)eif->eif_private;
        register int mode;
        union {
         char eaddr[8];
         long long laddr;
        } eau;

        /* reset the ethernet */
        mif->mac->mac_control = MAC_RESET;
```

```
        mif->mac->mac_control = 0;

        /* store station address in RAM */
        eau.laddr = 0;
        bcopy((caddr_t)mif->ei_ac.ac_enaddr,
                (caddr_t)&eau.eaddr[2], ETHERADDRLEN);
        write_reg64(eau.laddr, (__psunsigned_t)&mif->mac->physaddr);

        /* probe for the external transceiver */
        mif->phystatus = 0;
        if (mace_ether_mdio_probe(mif) < 0) {
                mif->phystatus = PHYS_WASDOWN;
                cmn_err(CE_ALERT, “ec0: phy device not found, probe failed”);
        }

        /* Load errata work-arounds into PHY */
        mace_ether_mdio_errata(mif);

        /* default to 100mbit half duplex (speed really doesn’t matter) */
        mode = MAC_NORMAL | MAC_100MBIT;
        if (flags & IFF_PROMISC)
                mode |= MAC_PROMISCOUS;
        if (me_halfduplex_ipg[0]) {
                mode |= me_halfduplex_ipg[0] << MAC_IPGT_SHIFT;
                mode |= me_halfduplex_ipg[1] << MAC_IPGR1_SHIFT;
                mode |= me_halfduplex_ipg[2] << MAC_IPGR2_SHIFT;
        } else {
                mode |= MAC_DEFAULT_IPG;
        }
        mif->mode = mode;

        /* hardware init */
        mace_hdwrether_init(mif);

        return 0;
}

static void
mace_ether_rx_fifo_error(register struct maceif *mif, int status)
{
        if (status & INTR_RX_MSGS_UNDERFLOW)
                cmn_err(CE_NOTE, “ec0: RX msg cluster list empty”);
        if (status & INTR_RX_FIFO_OVERFLOW) {
                cmn_err(CE_ALERT, “ec0: RX error, data FIFO overflow”);
                mace_hdwrether_init(mif);
        }
        mif->rx_fifo_errors++;
}

static char *
mace_ether_tx_emsg(unsigned status)
{
        if (status & TX_VEC_ABORTED_UNDERRUN)
                return “fifo underrun”;
```

```
        else if (status & TX_VEC_ABORTED_TOO_LONG)
                return “giant pkt”;
        else if (status & TX_VEC_DROPPED_COLLISIONS)
                return “excess collisions”;
        else if (status & TX_VEC_CANCELED_DEFERRAL)
                return “excess deferrals”;
        else if (status & TX_VEC_DROPPED_LATE_COLLISION)
                return “late collision”;
        else
                return “???”;
}

static void
mace_ether_tx_purge(register struct maceif *mif)
{
        register int i;

        /* turn off transmit ring dma */
        mif->mac->dma_control &= ~ETHER_TX_DMA_ENABLE;

        /* free any previously queued buffers (toss-em) */
        for (i = 0; i < TX_RING_SIZE; i++) {
                m_freem(mif->tx_mfifo[i]);
                mif->tx_mfifo[i] = NULL;
        }
        mif->tx_rptr = mif->tx_wptr = mif->mac->tx_ring_rptr;
        mif->mac->tx_ring_wptr = mif->tx_wptr;
        mif->tx_free_space = TX_RING_SIZE;
}

static void
mace_ether_tx_error(register struct maceif *mif, int status)
{
        register unsigned vstatus = mif->mac->irltv.last_transmit_vector;

        /* statistics */
        if (status & INTR_TX_LINK_FAIL) {
                if (mif->phystatus & PHYS_WASUP)
                        cmn_err(CE_ALERT, “ec0: no carrier: check Ethernet
cable”);
                else
                        mace_ether_tx_purge(mif);
        }
        if (status & INTR_TX_MEMORY_ERROR) {
                cmn_err(CE_ALERT, “ec0: TX memory read error”);
                mif->ei_if.if_flags &= ~IFF_UP;
        }
        if (status & INTR_TX_ABORTED) {
                cmn_err(CE_WARN, “ec0: TX aborted, %s (0x%08X)”,
                        mace_ether_tx_emsg(vstatus), vstatus);
                mif->mac->dma_control |= ETHER_TX_DMA_ENABLE;
        }
        mif->tx_ring_errors++;
}
```

```
static void
mace_ether_reset(struct etherif *eif)
{
        register struct maceif *mif = (struct maceif *)eif->eif_private;
        register int i;

        /* put mac through global reset state */
        mif->mac->mac_control = MAC_RESET;
        mif->mac->mac_control = 0;

        /* turn off watchdog */
        mif->ei_if.if_timer = 0;

        /* reset both DMA channels */
        mace_ether_rx_fifo_error(mif, 0);
        mif->rx_rptr = 0;

        /* free any receive buffers */
        for (i = 0; i < MSGCL_FIFO_SIZE; i++) {
                m_freem(mif->rx_mfifo[i]);
                mif->rx_mfifo[i] = NULL;
        }

        /* cleanup old TX buffers */
        mace_ether_tx_purge(mif);
}

static void
mace_ether_watchdog(int unit)
{
        register struct maceif *mif = &mace_ether;
        register int rval;

        /* Pick up status and free fifo space */
        mace_ether_transmit_complete(mif);

        /* Watch link status in the xcvr, broken for MACE rev 0 */
        if (mif->phystatus & PHYS_START) {
                if (((rval = mif->mac->phy_dataio) & MDIO_BUSY) == 0) {
                        mif->phystatus &= ~(PHYS_WASUP|PHYS_WASDOWN);
                        if (rval & PHY_PMSR_LINK) {
                                mace_ether_link_update(mif, rval);
                                mif->ei_if.if_flags |= IFF_UP;
                                mif->phystatus |= PHYS_WASUP;
                        } else {
                                mace_ether_tx_error(mif, INTR_TX_LINK_FAIL);
                                mif->ei_if.if_flags &= ~IFF_UP;
                                mif->phystatus |= PHYS_WASDOWN;
                                mif->phystatus &= ~PHYS_UPDATE;
                        }
                }
        }
        if (mif->revision) {
```

```
                mif->mac->phy_address = (mif->phyaddr << 5) | 0x1;
                mif->mac->phy_read_start = 1;
                mif->phystatus |= PHYS_START;
        }

        /* keep watchdog running */
        mif->ei_if.if_timer = IFNET_SLOWHZ;
}

static void
mace_ether_intr(int unit)
{
        register struct maceif *mif = &mace_ether;
        union {
                __uint32_t all;
                struct {
                        __uint32_t :2,
                                        rxseqnum:5,
                                        txrptr:9,
                                        rxrptr:8,
                                        isf:8;
                } comp;
        } status;

        /* Read interrupt status from dispatch register in MACE */
        while (((status.all = mif->mac->interrupt_status) & 0xff) != 0) {

         /*
         * Need to reclaim tx packets first until the NFS client
         * clntkudp_callit() routine is fixed to not be brain dead
         * in serializing all requests into the same private buffer.
         */
         if (mif->tx_rptr != status.comp.txrptr) {
                mace_ether_transmit_complete(mif);
         }

         /* Process received packets */
         if (status.comp.isf & INTR_RX_DMA_REQ) {
                mace_ether_receive(mif, status.comp.rxrptr,
                                        status.comp.rxseqnum);
         }

         /* Check for receive errors */
         if (status.comp.isf & ETHER_RX_ERRORS) {
                mace_ether_rx_fifo_error(mif, status.comp.isf);
                mif->mac->interrupt_status = ETHER_RX_ERRORS;
         }

         /* Check for transmit errors */
         if (status.comp.isf & ETHER_TX_ERRORS) {
                mace_ether_tx_error(mif, status.comp.isf);
                mif->mac->interrupt_status = ETHER_TX_ERRORS;
         }
```

```
         /* Check transmit complete status */
         if (status.comp.isf & INTR_TX_PKT_REQ) {
                mif->mac->interrupt_status = INTR_TX_PKT_REQ;
         }

         /* Check transmit fifo empty status */
         if (status.comp.isf & INTR_TX_DMA_REQ) {
                /* should be nothing left?, turn off drain interrupt */
                mif->mac->tx_control = 0;
         }

        }

        return;
}

/*
 * Hardware internet checksum support
 */
static void
mace_ether_hdwrcksum(
        struct maceif *mif,
        struct mbuf *m0,
        statistics_vector_t statistics)
{
        struct etherbufhead *ebh;
        struct ether_header *eh;
        __uint32_t cksum;
        __uint32_t x;
        struct ip *ip;
        char *crc;
        int hlen, rlen;

        /*
         * Finish TCP or UDP checksum on non-fragments.
         */
        cksum = (statistics >> RX_VEC_CKSUM_SHIFT) & 0xffff;
        rlen = statistics & RX_VEC_LENGTH;
        ebh = mtod(m0, struct etherbufhead *);
        eh = &ebh->ebh_ether;
        ip = (struct ip *)(ebh + 1);
        hlen = ip->ip_hl << 2;
        if ((ntohs(eh->ether_type) == ETHERTYPE_IP) &&
         ((ip->ip_off & (IP_OFFMASK|IP_MF)) == 0) &&
         ((ip->ip_p == IPPROTO_TCP) || (ip->ip_p == IPPROTO_UDP))) {

                /*
                 * compute checksum of the pseudo-header
                 */
                cksum += (ip->ip_len - hlen) +
                        htons((ushort)ip->ip_p) +
                        (ip->ip_src.s_addr >> 16) +
                        (ip->ip_src.s_addr & 0xffff) +
                        (ip->ip_dst.s_addr >> 16) +
```

```
                        (ip->ip_dst.s_addr & 0xffff);

                /*
                 * Subtract the ether header from the checksum.
                 * The IP header will sum to logical zero if it's correct
                 * so we don't need to include it here. This is safe since,
                 * if it's incorrect, ip input code will toss it anyway.
                 */
                x = ((u_short*)eh)[0] + ((u_short*)eh)[1] +
                 ((u_short*)eh)[2] + ((u_short*)eh)[3] +
                 ((u_short*)eh)[4] + ((u_short*)eh)[5] +
                 ((u_short*)eh)[6];
                x = (x & 0xffff) + (x >> 16);
                x = 0xffff & (x + (x >> 16));
                cksum += 0xffff ^ x;

                /*
                 * subtract CRC portion that is not part of checksum
                 */
                crc = &(((char *)ip)[rlen - (ETHER_HDRLEN + CRCLEN)]);
                if (rlen & 1) {/* odd */
                        cksum += 0xffff ^ (u_short) ((crc[1] << 8) | crc[0]);
                        cksum += 0xffff ^ (u_short) ((crc[3] << 8) | crc[2]);
                } else { /* even */
                        cksum += 0xffff ^ (u_short) ((crc[0] << 8) | crc[1]);
                        cksum += 0xffff ^ (u_short) ((crc[2] << 8) | crc[3]);
                }

                /*
                 * fold in carries
                 */
                cksum = (cksum & 0xffff) + (cksum >> 16);
                cksum = 0xffff & (cksum + (cksum >> 16));

                /*
                 * valid iff all 1's
                 */
                if (cksum == 0xffff) {
                        m0->m_flags |= M_CKSUMMED;
                }
        }
}

/*
 * Record statistics and send up to protocol level
 */
static void
mace_ether_input(
        struct maceif *mif,
        struct mbuf *m0,
        statistics_vector_t statistics)
{
        register int length, error = 0, snoopflags = 0;
        register unsigned stats = (unsigned)statistics;
```

```
        register void *rbp;

        /* save statistics info */
        mif->ei_if.if_ipackets++;
        length = stats & RX_VEC_LENGTH;
        mif->ei_if.if_ibytes += length;
        if (stats & (RX_VEC_MULTICAST | RX_VEC_BROADCAST |
                 RX_VEC_CODE_VIOLATION | RX_VEC_LONG_EVENT |
                 RX_VEC_CRC_ERROR | RX_VEC_INVALID_PREAMBLE |
                 RX_VEC_DRIBBLE_NIBBLE | RX_VEC_CARRIER_EVENT |
                 RX_VEC_BAD_PACKET)) {
                if (stats & RX_VEC_MULTICAST)
                        mif->rx_multicast++;
                if (stats & RX_VEC_BROADCAST)
                        mif->rx_broadcast++;
                if (stats & RX_VEC_CODE_VIOLATION) {
                        snoopflags |= SNERR_FRAME;
                        mif->rx_code_violation++;
                }
                if (stats & RX_VEC_LONG_EVENT) {
                        snoopflags |= SNERR_FRAME;
                        mif->rx_long_event++;
                }
                if (stats & RX_VEC_CRC_ERROR) {
                        snoopflags |= SNERR_CHECKSUM;
                        mif->rx_crc_error++;
                }
                if (stats & RX_VEC_INVALID_PREAMBLE)
                        mif->rx_invalid_preamble++;
                if (stats & RX_VEC_DRIBBLE_NIBBLE)
                        mif->rx_dribble_nibble++;
                if (stats & RX_VEC_CARRIER_EVENT)
                        mif->rx_carrier_event++;

                /* set master snoop error flags
                 if any error conditions set */
                if (stats & RX_VEC_BAD_PACKET) {
                        mif->ei_if.if_ierrors++;
                        snoopflags |= SN_ERROR;
                        error = 1;
                }
        }

        /* ifheader at the front of the received buffer */
        rbp = mtod(m0, void *);
        IF_INITHEADER(rbp, &mif->ei_if, sizeof(struct etherbufhead));

        /* test for promiscuous packets */
        if ((stats & RX_PROMISCUOUS) == 0) {
                snoopflags |= SN_PROMISC;
        }

        /* hardware internet checksum checker */
        if (!error && (length > ETHERMINLEN) && me_hdwrcksum_enable) {
```

```
                mace_ether_hdwrcksum(mif, m0, statistics);
        }

        /* call input function to deliver the packet */
        (void) ether_input(&mif->eif, snoopflags, m0);
}

/*
 * Takes a received message buffer cluster and breaks it up into the
 * individual packets contained within it.
 */
static void
mace_ether_receive_unwind(register struct maceif *mif, struct mbuf *m0)
{
        register statistics_vector_t statistics, *ps;
        register int length = 0;

/* ??? - do we really need to do this - it is VERY expensive ??? */
        /* Invalidate cache contents over entire buffer length */
        dki_dcache_inval(mtod(m0, void *), RCVBUF_SIZE);

        /* Get length */
        ps = mtod(m0, statistics_vector_t *);
        statistics = *ps;
        ASSERT(statistics & RX_VEC_FINISHED);
        length = statistics & RX_VEC_LENGTH;
        if (length > ETHERMAXLEN)
                length = ETHERMAXLEN;
        length -= ETHER_HDRLEN + CRCLEN;
        length += sizeof (struct etherbufhead);

        /* Set mbuf length and pass upstream */
        m0->m_len = length;
        mace_ether_input(mif, m0, statistics);
}

/*
 * Basic procedure that checks to see how many message cluster buffers the
 * hardware has used and replaces them. Note that if we can't replace a
 * buffer we must resubmit the old one and drop the received data.
 *
 * Note: this could be done at a lower software interrupt priority.
 */
static void
mace_ether_receive(register struct maceif *mif, int nptr, int seqnum)
{
        register int optr, rlen = mif->rx_rlen;
        register struct mbuf *m, *m0;

        /* RX msgs packet collection, also gather statistics */
        ASSERT(nptr < (MSGCL_FIFO_SIZE * 2));
        optr = mif->rx_rptr;
        while (optr != nptr) {
                /* Message cluster being processed */
```

```
                m = mif->rx_mfifo[RXRINGINDEX(optr)];

                /* Must have replacement cluster buffer */
                if ((m0 = m_vget(M_DONTWAIT, RCVBUF_SIZE, MT_DATA)) != NULL) {
                        mace_ether_receive_unwind(mif, m);
                } else {
                        m0 = m;
                }
                mif->mac->rx_fifo = kvtophys(mtod(m0, caddr_t));

                /* New message cluster is at end of queue */
                mif->rx_mfifo[RXRINGINDEX(optr + rlen)] = m0;

                /* Update pointer */
                optr = RXFIFOINDEX(optr + 1);

                ASSERT(mif->mac->rx_fifo_depth <= MSGCL_FIFO_SIZE);
        }
        mif->rx_rptr = optr;

        return;
}

/*
 * Transmit vector statistics
 */
static void
mace_ether_transmit_stats(struct maceif *mif, unsigned stats)
{
        register int collisions = (stats & TX_VEC_COLLISIONS) >>
                                  TX_VEC_COLLISION_SHIFT;

        mif->ei_if.if_obytes += stats & TX_VEC_LENGTH;
        mif->ei_if.if_collisions += collisions;
        if (stats & TX_VEC_DEFERRED) {
                mif->ei_if.if_collisions++;
                mif->tx_deferred++;
        }
        if ((stats & TX_VEC_COMPLETED_SUCCESSFULLY) == 0) {
                if (stats & TX_VEC_LATE_COLLISION)
                        mif->tx_late_collisions++;
                if (stats & TX_VEC_CRC_ERROR)
                        mif->tx_crc_error++;
                if (stats & TX_VEC_ABORTED_TOO_LONG)
                        mif->tx_aborted_too_long++;
                if (stats & TX_VEC_ABORTED_UNDERRUN)
                        mif->tx_aborted_underrun++;
                if (stats & TX_VEC_DROPPED_COLLISIONS)
                        mif->tx_dropped_collisions++;
                if (stats & TX_VEC_CANCELED_DEFERRAL)
                        mif->tx_canceled_deferral++;
                if (stats & TX_VEC_DROPPED_LATE_COLLISION)
                        mif->tx_dropped_late_collision++;
                if (stats & (TX_VEC_CRC_ERROR |
```

```
                        TX_VEC_ABORTED_TOO_LONG |
                        TX_VEC_ABORTED_UNDERRUN |
                        TX_VEC_CANCELED_DEFERRAL)) {
                       mif->ei_if.if_oerrors++;
                }
        }
}

/*
 * pick up status vectors and rack up free space, can be called either from
 * interrupt time or output time.
 */
void
mace_ether_transmit_complete(register struct maceif *mif)
{
        register int tx_rptr = mif->tx_rptr, tx_wptr = mif->tx_wptr, cnt = 0;
        register volatile TXfifo *f = &mif->tx_fifo[tx_rptr];
        register long long status;

        /* Empty? */
        if (tx_rptr == tx_wptr)
                return;

        /* TX FIFO garbage collection, also gather statistics */
        while ((status = f->TXStatus) & TX_VEC_FINISHED) {
                /* Gather info and record done */
                mace_ether_transmit_stats(mif, (unsigned)status);
                m_freem(mif->tx_mfifo[tx_rptr]);
                mif->tx_mfifo[tx_rptr] = 0;
                cnt++;
                tx_rptr = TXFIFOINDEX(tx_rptr + 1);
                if (tx_rptr == tx_wptr)
                        break;
                f = &mif->tx_fifo[tx_rptr];
        }

        /* Record new ring complete position */
        mif->tx_rptr = tx_rptr;
        (void) atomicAddInt(&mif->tx_free_space, cnt);
}

#ifdef ENETDEBUG
/*
 * DEBUG ONLY
 */
static void
mace_valid_txcmd(volatile TXfifo *f)
{
        register u_int ec, i, ccnt, clen, len;

#define MACE_ECODE(num){ ec = (num); goto bad; }

        /* Header */
        if ((len = (f->TXCmd & 0xFFFF)) > 1513)
```

```
                MACE_ECODE(0);
        if (((f->TXCmd >> 16) & 0xFF) > 0x7F)
                MACE_ECODE(1);
        if ((f->TXCmd >> 32) != 0)
                MACE_ECODE(2);
        for (ccnt = 0, i = 25; i < 32; ++i) {
                if ((f->TXCmd >> i) & 1)
                        ++ccnt;
        }
        if (ccnt > TX_CMD_NUM_CATS)
                MACE_ECODE(3);

        /* Concatenation buffers */
        for (i = 1; i <= ccnt; ++i) {
                if (f->TXConcatPtr[i] & 7)
                        MACE_ECODE(4);
                if ((f->TXConcatPtr[i] >> 44) != 0LL)
                        MACE_ECODE(5);
                if ((clen = ((f->TXConcatPtr[i] >> 32) & 0x3FF)) > 1513)
                        MACE_ECODE(6);
        }

        return;

bad:
        printf("mace_valid_txcmd: bad cmd blk detected (%d)\n", ec);
        printf("\tTXCmd = 0x%016llX\n", f->TXCmd);
        printf("\tTXConcatPtr[0] = 0x%016llX\n", f->TXConcatPtr[0]);
        printf("\tTXConcatPtr[1] = 0x%016llX\n", f->TXConcatPtr[1]);
        printf("\tTXConcatPtr[2] = 0x%016llX\n", f->TXConcatPtr[2]);
        for (i = 4; i < 16; i++) {
                printf("\tTXData[%d] = 0x%016llX\n", i, f->TXData[i]);
        }
}
#endif

/*
 * Try to build a concatenation list for the supplied mbuf chain. We
 * don't try to optimize the failure case.
 */
static int
mace_tx_catlist(
        struct maceif *mif,
        volatile TXfifo *f,
        struct mbuf *m,
        int *plen)
{
        register int rev = mif->revision, ccnt = 1, remain, hlen, flen, len;
        register paddr_t vaddr;

#define IO_PGSIZE4096
#define IO_PGOFFSET(IO_PGSIZE - 1)
#define IO_PGMASK~(IO_PGSIZE - 1)
```

```
/* Count protocol header bytes, usually one mbuf */
for (hlen = 0; m != NULL; m = m->m_next) {
        if (m->m_type != MT_HEADER) {
                break;
        }
        if ((hlen + m->m_len) > 96) {
                break;
        }
        hlen += m->m_len;
}

/* Process all remaining mbufs */
for (ccnt = 1; m != NULL; m = m->m_next) {
        /* Get data length, skip empty buffers */
        if ((len = m->m_len) == 0)
                continue;

        /* Valid aligned starting address */
        if ((vaddr = mtod(m, paddr_t)) & 7) {
                /* Pullup bytes in first buffer to align? */
                if (ccnt == 1) {
                        flen = 8 - (int)(vaddr & 7);
                        if (flen >= len) {
                                /* short buf, pullup */
                                flen = len;
                                hlen += flen;
                                continue;
                        } else {
                                /* clip bytes off front */
                                hlen += flen;
                                vaddr += flen;
                                len -= flen;
                        }
                } else {
                        ++mif->tcase[4];
                        return -1;
                }
        }

        /* Force dword length (for MACE ethernet rev 0 only) */
        if (!rev && m->m_next && ((len & 7) != 0)) {
                ++mif->tcase[5];
                return -1;
        }

        /* Writeback cache contents over virtual length */
        dki_dcache_wb((void *)vaddr, len);

        /* Create gather elements (check for page crossing) */
        if ((vaddr & IO_PGMASK) != ((vaddr + len - 1) & IO_PGMASK)) {
                remain = IO_PGSIZE - (int)(vaddr & IO_PGOFFSET);

                /* Set concatenation pointer address & length */
                f->TXConcatPtr[ccnt++] =
```

```
                                ((long long)(remain - 1) << 32) |
                                kvtophys((void *)vaddr);

                        /* Whats left on the following virtual page? */
                        vaddr += remain;
                        len -= remain;
                }

                /* Set concatenation pointer address & length */
                f->TXConcatPtr[ccnt] = ((long long)(len - 1) << 32) |
                        kvtophys((void *)vaddr);

                /* Advance and fail if at concatenation list limit */
                if (ccnt++ > TX_CMD_NUM_CATS) {
                        ++mif->tcase[6];
                        return -1;
                }
        }

        /* tell caller how much we want to copy */
        *plen = hlen;
        hlen += ETHER_HDRLEN;

        /* will the copied data and concatenation list overlap? */
        if (hlen > (sizeof(TXfifo) - ccnt * sizeof(long long))) {
                ++mif->tcase[7];
                return -1;
        }

        return ccnt - 1;
}

/*
 * Match the cnt words of packet headers pointed at by hp against sf.
 *
 * Note: hp is not aligned, must use LWL/LWR instructions here!!!!!
 */
static int
sfmatch(struct snoopfilter *sf, char *hp, int cnt)
{
        __uint32_t *mask, *match;

        hp -= 3;                /* hack to force lwl/lwr */
        mask = sf->sf_mask;
        match = sf->sf_match;
        while (--cnt >= 0) {
                if (((*(__uint32_t *)&hp[3]) & *mask) != *match)
                        return 0;
                hp += sizeof (__uint32_t), mask++, match++;
        }
        return 1;
}

/*
```

```
 * Slightly customized version of snoop_match() which expects
 * a word-aligned address *after* the ether_header so we don't
 * have to copy and align headers to make snoop_match() happy.
 *
 * XXX - should optimize this, need our own ether_selfsnoop.
 */
static void
mace_selfsnoop(struct maceif *mif, caddr_t eh, struct mbuf *m0, int plen)
{
        register struct rawif *rif = &mif->ei_rawif;
        register char *hp = mtod(m0, char *);
        register int i, len = m0->m_len;
        struct ether_header ehdr;
        struct snoopfilter *sf;

        len >>= RAW_ALIGNSHIFT;
        if (len > SNOOP_FILTERLEN)
                len = SNOOP_FILTERLEN;

        i = rif->rif_sfveclen;
        for (sf = rif->rif_sfvec; --i >= 0; sf++) {
                if (!sf->sf_active)
                        continue;
                if (sfmatch(sf, hp, len)) {
                        rif->rif_matched = sf;
                        bcopy((void *)eh, (void *)&ehdr, sizeof ehdr);
                        ether_selfsnoop(&mif->eif, &ehdr, m0, 0, plen);
                        return;
                }
        }

        return;
}

/*
 * High level ethernet output entry point, called by upper level protocol
 * stack to place a new message buffer chain into the output queue.
 *
 * Each 128byte MACE tx descriptor describes a single packet that contains
 * up to 120 bytes of data locally, plus up to three pointers to noncontiguous
 * data to be concatenated onto the end of the packet by the hardware.
 *
 * The MACE tx pointers have the following restrictions:
 * - addresses must start on dword aligned boundaries
 * - iff MACE1.0 is used, block lengths must be dword multiples
 */
static int
mace_ether_output(
        struct etherif *eif,/* on this interface */
        struct etheraddr *edst,/* with these addresses */
        struct etheraddr *esrc,
        u_short type,          /* of this type */
        struct mbuf *m0)/* send this chain */
{
```

```
register struct maceif *mif = (struct maceif *)eif->eif_private;
register volatile TXfifo *f = &mif->tx_fifo[mif->tx_wptr];
register int m0save = 0, ccnt, tlen, len, txcmd, nwptr;
register struct mbuf *m;
struct mehdr {
        struct etheraddr dst, src;
        char htype, ltype;
} *eh;
int plen;
static int cmdmap[] = {
        0,
        TX_CMD_CONCAT_1,
        TX_CMD_CONCAT_1 | TX_CMD_CONCAT_2,
        TX_CMD_CONCAT_1 | TX_CMD_CONCAT_2 | TX_CMD_CONCAT_3,
};

/*
 * Space in TX ring? (note: don't use last entry, fix this)
 */
if (mif->tx_free_space <= 1) {
        IF_DROP(&mif->ei_if.if_snd);
        m_freem(m0);
        return ENOBUFS;
}

/*
 * Link is down, don't queue new packets
 */
if (mif->phystatus & PHYS_WASDOWN) {
        mif->ei_if.if_odrops++;
        m_freem(m0);
        return EIO;
}

/*
 * Calculate length of packet
 */
len = m_length(m0);
tlen = len + ETHER_HDRLEN;

++mif->tcase[0];

/*
 * Init TX cmd header, interrupt every 1/4 ring
 */
txcmd = TX_CMD_TERM_DMA;
nwptr = TXFIFOINDEX(mif->tx_wptr + 1);
if ((nwptr & ((TX_RING_SIZE / 4) - 1)) == 0) {
        txcmd |= TX_CMD_SENT_INT_EN;
}

/*
 * CASE #1
 * If it's a short packet, just put directly in the TX ring.
```

```
 */
if (tlen < (sizeof(TXfifo) - sizeof(long long))) {
        /* Set up the fifo block cmd header */
        f->TXCmd = txcmd |
                        ((sizeof(TXfifo) - tlen) <<
                         TX_CMD_OFFSET_SHIFT) | (tlen - 1);

        /* Ethernet header insertion point */
        eh = (struct mehdr *)&f->buf[sizeof(TXfifo) - tlen];

        /* Copy the packet into the TX ring */
        m_datacopy(m0, 0, len,
                (void *)&f->buf[sizeof(TXfifo) - len]);

        ++mif->tcase[1];

/*
 * CASE #2
 * Pull up all the protocol headers, then check if we can DMA
 * directly out of the remaining mbufs using the concatenation
 * buffers. If not, copy the rest of the data into a cluster
 * and dma directly out of that instead. Turns out that this
 * works out ok 99.9% of the time (very few bcopys).
 */
} else if ((ccnt = mace_tx_catlist(mif, f, m0, &plen)) > 0) {
        /* Data insertion point */
        eh = (struct mehdr *)&f->buf[sizeof(TXfifo) - plen];

        /* Pullup plen data bytes of mbuf chain? */
        m_datacopy(m0, 0, plen, (void *)eh);

        /* Backup for ethernet header insertion point */
        plen += ETHER_HDRLEN;
        eh--;

        /* Set up the fifo block cmd header */
        ASSERT(ccnt <= TX_CMD_NUM_CATS);
        f->TXCmd = txcmd | cmdmap[ccnt] |
                ((sizeof(TXfifo) - plen) << TX_CMD_OFFSET_SHIFT) |
                 (tlen - 1);

        /* Need to save packet until dma is complete */
        mif->tx_mfifo[mif->tx_wptr] = m0;
        m0save = 1;

        ++mif->tcase[2];
/*
 * CASE #3
 * Just copy the packet into a single cluster and DMA out of it.
 */
} else {
        /* Need a message buffer cluster to hold the entire packet */
        if ((m = m_vget(M_DONTWAIT, len, MT_DATA)) == NULL) {
                mif->ei_if.if_odrops++;
```

```
                        m_freem(m0);
                        return ENOBUFS;
                }
                mif->tx_mfifo[mif->tx_wptr] = m;

                /* Set up the fifo block cmd header */
                f->TXCmd = txcmd | TX_CMD_CONCAT_1 |
                                ((sizeof(TXfifo) - ETHER_HDRLEN)
                                        << TX_CMD_OFFSET_SHIFT) |
                                (tlen - 1);

                /* Ethernet header insertion point */
                eh = (struct mehdr *)&f->buf[sizeof(TXfifo) - ETHER_HDRLEN];

                /* Copy the packet into a contiguous buffer */
                m_datacopy(m0, 0, m->m_len, mtod(m, caddr_t));

                /* Writeback cache contents over needed length */
                dki_dcache_wb(mtod(m, void *), m->m_len);

                /* Set concatenation pointer #1 physical address & length */
                f->TXConcatPtr[1] = ((long long)(m->m_len - 1) << 32) |
                        kvtophys(mtod(m, caddr_t));

                ++mif->tcase[3];
        }

        /* Create ether header at front of the packet (unaligned) */
        eh->dst = *edst;
        eh->src = *esrc;
        eh->htype = type >> 8;
        eh->ltype = type;

        /* Check whether snoopers want to copy this packet */
        if (RAWIF_SNOOPING(&mif->ei_rawif)) {
                /* Call snoop routine to filter and deliver pkt */
                mace_selfsnoop(mif, (caddr_t)eh, m0, len);
        }

        /* Free the original mbuf list, no longer needed */
        if (!m0save) {
                m_freem(m0);
        }

#ifdef ENETDEBUG
        /* Check if transmit command is valid (DEBUG ONLY) */
        mace_valid_txcmd(f);
#endif

        /* Place the packet into the TX hardware output queue */
        (void) atomicAddInt(&mif->tx_free_space, -1);
        mif->tx_wptr = nwptr;
        mif->mac->tx_ring_wptr = nwptr;
```

```
        return 0;
}

/*
 * DEC Poly routine
 */
static unsigned
CrcGen(unsigned char *Bytes, int BytesLength)
{
        unsigned Crc = 0xFFFFFFFF;
        unsigned const Poly = 0x04c11db7;
        unsigned Msb;
        unsigned char CurrentByte;
        int Bit;

        while (BytesLength-- > 0) {
                CurrentByte = *Bytes++;
                for (Bit = 0; Bit < 8; Bit++) {
                        Msb = Crc >> 31;
                        Crc <<= 1;
                        if (Msb ^ (CurrentByte & 1)) {
                                Crc ^= Poly;
                                Crc |= 1;
                        }
                        CurrentByte >>= 1;
                }
        }

        return Crc;
}

/*
 * Given a multicast ethernet address, this routine calculates the
 * address's bit index in the logical address filter mask
 */
static int
mace_laf_hash(u_char *addr, int len)
{
        return CrcGen(addr, len) >> 26;
}

static int
mace_ether_ioctl(
        struct etherif *eif,
        int cmd,
        void *data)
{
        register struct maceif *mif = (struct maceif *)eif->eif_private;
        struct mfreq *mfr;
        union mkey *key;

        mfr = (struct mfreq *)data;
        key = mfr->mfr_key;
```

```
        switch (cmd) {
         /* Enable one of the multicast filter flags */
         case SIOCADDMULTI:
                mfr->mfr_value =
                        mace_laf_hash(key->mk_dhost, sizeof (key->mk_dhost));
                if (LAF_TSTBIT(mif->mlaf, mfr->mfr_value)) {
                        ASSERT(mfhasvalue(&eif->eif_mfilter, mfr->mfr_value));
                        mif->lafcoll++;
                        break;
                }
                ASSERT(!mfhasvalue(&eif->eif_mfilter, mfr->mfr_value));
                LAF_SETBIT(mif->mlaf, mfr->mfr_value);
                write_reg64(mif->mlaf, (__psunsigned_t)&mif->mac->mlaf);
                if (mif->nmulti == 0)
                        eiftoifp(eif)->if_flags |= IFF_FILTMULTI;
                mif->nmulti++;
                break;

         /* Disable one of the multicast filter flags */
         case SIOCDELMULTI:
                if (mfr->mfr_value !=
                        mace_laf_hash(key->mk_dhost, sizeof(key->mk_dhost)))
                        return EINVAL;
                if (mfhasvalue(&eif->eif_mfilter, mfr->mfr_value)) {
                        /* Forget about this collision. */
                        --mif->lafcoll;
                        break;
                }

                /*
                 * If this multicast address is the last one to map
                 * the bit numbered by mfr->mfr_value in the filter,
                 * clear that bit and update the chip.
                 */
                LAF_CLRBIT(mif->mlaf, mfr->mfr_value);
                write_reg64(mif->mlaf, (__psunsigned_t)&mif->mac->mlaf);
                --mif->nmulti;
                if (mif->nmulti == 0)
                        eiftoifp(eif)->if_flags &= ~IFF_FILTMULTI;
                break;

         default:
                return EINVAL;
        }

        return (0);
}

static void
mace_hexdump(char *msg, char *cp, int len)
{
        register int idx;
        register int qqq;
        char qstr[512];
```

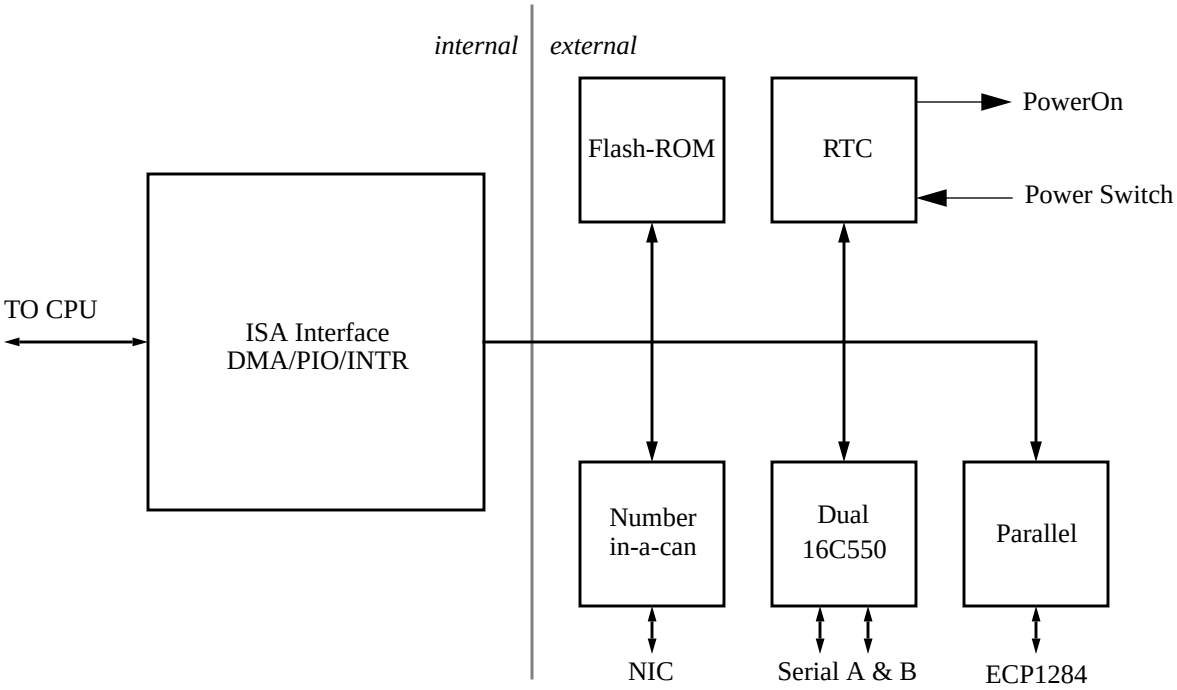
```
        int maxi = 128;
        static char digits[] = "0123456789abcdef";

        if (len < maxi)
                maxi = len;
        for (idx = 0, qqq = 0; qqq<maxi; qqq++) {
                if (((qqq%16) == 0) && (qqq != 0))
                        qstr[idx++] = '\n';
                qstr[idx++] = digits[cp[qqq] >> 4];
                qstr[idx++] = digits[cp[qqq] & 0xf];
                qstr[idx++] = ' ';
        }
        qstr[idx] = 0;
        qprintf("%s: %s\n", msg, qstr);
}

static void
mace_dumpif(struct ifnet *ifp)
{
        qprintf("if_name \"%s\" if_unit %d if_mtu %d if_flags 0x%x if_timer
%d\n",
                ifp->if_name, ifp->if_unit, ifp->if_mtu, ifp->if_flags,
                ifp->if_timer);
        qprintf("ifq_len %d ifq_maxlen %d ifq_drops %d\n",
                ifp->if_snd.ifq_len, ifp->if_snd.ifq_maxlen,
                ifp->if_snd.ifq_drops);
        qprintf("if_ipackets %d if_ierrors %d if_opackets %d if_oerrors %d\n",
                ifp->if_ipackets, ifp->if_ierrors,
                ifp->if_opackets, ifp->if_oerrors);
        qprintf("if_collisions %d if_ibytes %d if_obytes %d if_odrops %d\n",
                ifp->if_collisions, ifp->if_ibytes,
                ifp->if_obytes, ifp->if_odrops);
}

static char *
mace_phystr(int phytype)
{
        register char *pname = "";

        switch(phytype) {
         case PHY_QS6612X:
                pname = "QS6612";
                break;
         case PHY_ICS1889:
                pname = "ICS1889";
                break;
         case PHY_ICS1890:
                pname = "ICS1890";
                break;
         case PHY_DP83840:
                pname = "DP83840";
                break;
        }
```

```
        return pname;
}

static void
mace_dumpei(struct maceif *mif)
{
        qprintf("ac_enaddr %s\n", ether_sprintf(mif->ei_ac.ac_enaddr));
        qprintf("eif_rawif 0x%x\n", &mif->ei_rawif);
        qprintf("nmulti %d lafcoll %d ", mif->nmulti, mif->lafcoll);
        mace_hexdump("laf", (char *)&mif->mlaf, sizeof (long long));
        qprintf("phyaddr %d phyrev %d phystatus %x phytype %x %s\n",
                mif->phyaddr, mif->phyrev, mif->phystatus,
                mif->phytype, mace_phystr(mif->phytype));
        qprintf("tx_fifo 0x%08X mode 0x%08X\n", mif->tx_fifo, mif->mode);
        qprintf("tx_ring_errors %d rx_fifo_errors %d\n",
                mif->tx_ring_errors, mif->rx_fifo_errors);
        qprintf("tx_rptr %d tx_wptr %d tx_free_space %d\n",
                mif->tx_rptr, mif->tx_wptr, mif->tx_free_space);
        qprintf("tx_late_collisions %d tx_crc_error %d tx_deferred %d\n",
                mif->tx_late_collisions, mif->tx_crc_error, mif->tx_deferred);
        qprintf("tx_aborted_too_long %d tx_aborted_underrun %d\n",
                mif->tx_aborted_too_long, mif->tx_aborted_underrun);
        qprintf("tx_dropped_collisions %d tx_canceled_deferral %d\n",
                mif->tx_dropped_collisions, mif->tx_canceled_deferral);
        qprintf("tx_dropped_late_collision %d\n",
                mif->tx_dropped_late_collision);
        qprintf("txcases: %d %d %d %d / %d %d %d %d\n",
                mif->tcase[0], mif->tcase[1], mif->tcase[2], mif->tcase[3],
                mif->tcase[4], mif->tcase[5], mif->tcase[6], mif->tcase[7]);
        qprintf("rx_rptr %d rx_rlen %d rx_boffset %d\n",
                mif->rx_rptr, mif->rx_rlen, mif->rx_boffset);
        qprintf("rx_code_violation %d rx_dribble_nibble %d\n",
                mif->rx_code_violation, mif->rx_dribble_nibble);
        qprintf("rx_crc_error %d rx_invalid_preamble %d rx_long_event %d\n",
                mif->rx_crc_error, mif->rx_invalid_preamble,
                mif->rx_long_event);
        qprintf("rx_carrier_event %d rx_multicast %d rx_broadcast %d\n",
                mif->rx_carrier_event, mif->rx_multicast, mif->rx_broadcast);
}

static void
mace_dumpregs(struct maceif *mif)
{
        register long long *regs = (long long *)mif->mac;
        register int i;

        qprintf("intf regs: 0x%08X\n", regs);
        for (i = 0; i < 32; i += 4, regs += 4) {
                qprintf("%02x 0x%016llX 0x%016llX 0x%016llX 0x%016llX\n",
                        i, regs[0], regs[1], regs[2], regs[3]);
        }
}

static void
```

```
mace_ether_dump(int unit)
{
        struct maceif *mif = &mace_ether;

        mace_dumpif(&mif->ei_if);
        qprintf("\n");
        mace_dumpei(mif);
        qprintf("\n");
        mace_dumpregs(mif);
        qprintf("\n");
/*
        mace_dumpphy(mif);
*/
}
```

5 ISA Bus Interface

The *Moosehead* system I/O asic contains an interface to an external PC style ISA bus that is used to connect up the standard PC peripherals such as serial, parallel, game port, calendar clock, Flash-ROM, and NIC. The interface includes five DMA channels for the serial and parallel interfaces. The interface also provides all of the support for PIO bus cycles and interrupts from the external parts. A block diagram of the ISA interface is shown below:



ISA features:

- Provides interface to external serial, parallel, and game port devices
- Provides five DMA channels for connection to serial and parallel interfaces
- Provides interface to external Flash-ROM (up to 2 Megabytes maximum)
- Provides interface to external Dallas DS1687 Calendar Clock
- Provides simple interface to Number-In-a-Can serial PROM

Super I/O feature summary:

- Two high speed 16C550C serial ports with hardware RTS/CTS and per port HP IR link support
- Both serial ports have clock prescalers to support MIDI and normal bauds rates up to 460.8Kb
- EPP/ECP-1284 high speed parallel port with all downward compatible modes

Calendar Clock feature summary:

- Standard Dallas 12887 RTC core with century calendar extension
- Alarm feature extended from 24 hours to one month and connected to external power on circuit
- Unique 56-bit silicon serial # embedded in module (system serial # and ethernet address)
- Battery backed up NVRAM from 242 bytes to 4K bytes depending on model used

5.1 Register Programming Interface

The following table shows all of the ISA interface registers. All bits not explicitly defined are read as zeros. All registers are defined on 64-bit aligned boundaries and can be read or written using 64-bit programmed i/o operations. Each functional block of registers starts on a 16Kbyte aligned address. Gaps or unused space within each functional blocks 16Kbyte address range are aliased to existing registers. The register list is shown below:

TABLE 52. ISA Interface Registers

Page	Offset	Register Name	Type	Bits	Function
0	0x00	Ring Base & Reset	RW	31:0	Peripheral Controller Ring Base & Reset
	0x08	Misc. Control	RW	8:0	Flash-Rom/Dp-Ram/Led/NIC control register
	0x10	PC Interrupt Status	RW	31:0	Peripheral Controller Interrupt Status Register
	0x18	PC Interrupt Mask	RW	31:0	Peripheral Controller Interrupt Mask Register
	0x2000 - 0x3FFF	DP-RAM	RW	63:0	DP-RAM Programmed IO access
1	0x00	Parallel Context A	RW	63:0	Parallel DMA buffer context A register
	0x08	Parallel Context B	RW	63:0	Parallel DMA buffer context B register
	0x10	Parallel Cntl/Status	RW	4:0	Parallel DMA control and status register
	0x18	Parallel Diagnostic	RO	13:0	Parallel DMA diagnostic register
2	0x00	S1 TX Control	RW	10:5	Serial #1 TX DMA channel ring buffer control
	0x08	S1 TX Read Pointer	RW	11:0	Serial #1 TX DMA channel ring read pointer
	0x10	S1 TX Write Pointer	RW	11:5	Serial #1 TX DMA channel ring write pointer
	0x18	S1 TX Ring Depth	RO	11:5	Serial #1 TX DMA channel ring buffer depth
	0x20	S1 RX Control	RW	10:5	Serial #1 RX DMA channel ring buffer control
	0x28	S1 RX Read Pointer	RW	11:5	Serial #1 RX DMA channel ring read pointer
	0x30	S1 RX Write Pointer	RW	11:0	Serial #1 RX DMA channel ring write pointer
	0x38	S1 RX Ring Depth	RO	11:5	Serial #1 RX DMA channel ring buffer depth
3	0x00	S2 TX Control	RW	10:5	Serial #2 TX DMA channel ring buffer control
	0x08	S2 TX Read Pointer	RW	11:0	Serial #2 TX DMA channel ring read pointer
	0x10	S2 TX Write Pointer	RW	11:5	Serial #2 TX DMA channel ring write pointer
	0x18	S2 TX Ring Depth	RO	11:5	Serial #2 TX DMA channel ring buffer depth
	0x20	S2 RX Control	RW	10:5	Serial #2 RX DMA channel ring buffer control
	0x28	S2 RX Read Pointer	RW	11:5	Serial #2 RX DMA channel ring read pointer
	0x30	S2 RX Write Pointer	RW	11:0	Serial #2 RX DMA channel ring write pointer
	0x38	S2 RX Ring Depth	RO	11:5	Serial #2 RX DMA channel ring buffer depth

Note: The ISA control register space consists of 4 pages of 16Kbytes each.

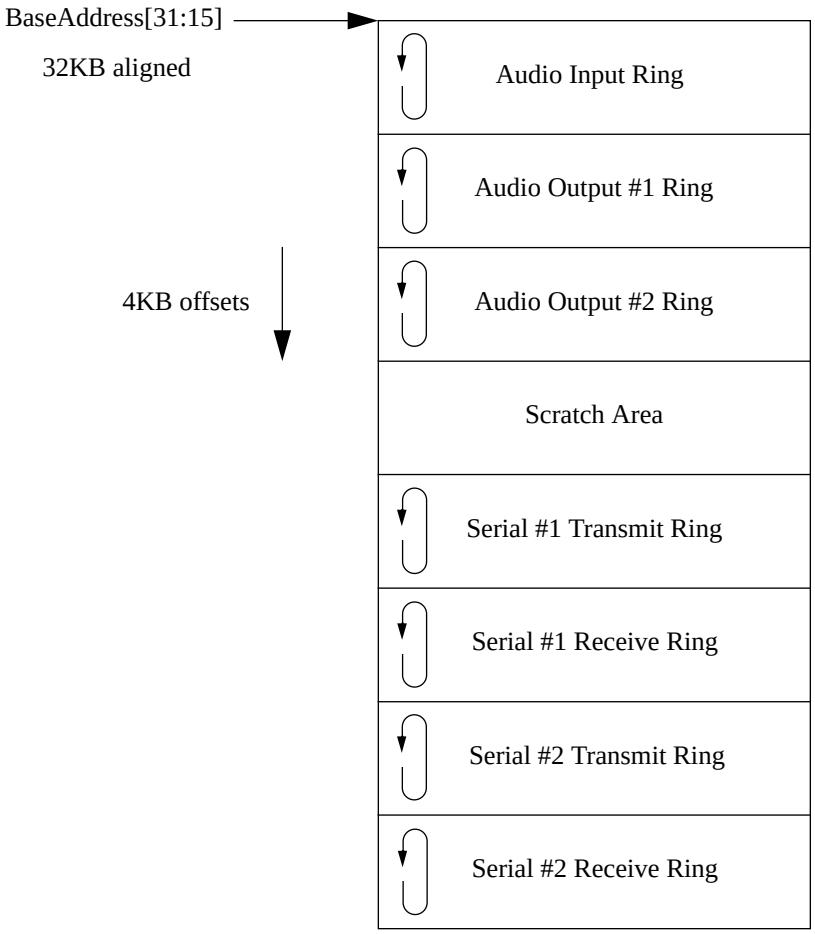
5.1.1 Peripheral controller ring base address and ISA external RESET

The following tables show the individual bits for the ring base and reset register:

TABLE 53. Peripheral ring base and reset register

Bits	Reset Value	Type	Description
0	1	RW	RESET (to external super i/o or other discrete serial/parallel interface chips) 0 - nop, reset inactive 1 - reset external controller
14:1	0	RO	<reserved, read as zeros>
31:15	0	RW	Base address of peripheral controller ring buffers

The base address field in this control register forms the 32KB aligned base address of all eight ring buffers in the peripheral controller. Each of the eight ring buffers within the specified block is 4KB in length and alignment.



5.1.2 Flash-ROM/LED/DP-RAM/NIC control register

The following tables show the individual bits for the Flash-ROM/LED/DP-RAM/NIC control register:

TABLE 54. Flash-ROM/NIC control register

Bits	Reset Value	Type	Description
0	0	RW	Write enable 0 - writes disabled (write operations go into bit bucket) 1 - writes to Flash-ROM enabled
1	0	RO	External password clear enable 0 - password clear disabled 1 - password clear enabled
2	0	RW	NIC interface deassert 0 - Hold NIC bidirectional interface low 1 - no operation
3	0	RO	Current value on NIC interface signal 0 - signal is low 1 - signal is high
4	0	RW	Red LED Enable 0 - enabled, red LED color active 1 - disabled
5	1	RW	Green LED Enable 0 - enabled, green LED color active 1 - disabled
6	0	RW	Dp-RAM Write Enable 0 - disabled 1 - PIO write access to the Dp-RAM is enabled
7	0	RW	Test UST Timer 0 - disabled 1 - Test mode enabled
8	0	RW	SA20 Enable 0 - sa20 functions as an address bit 1 - Game port chip select on sa20

5.1.3 Peripheral Controller Interrupt Status and Mask registers

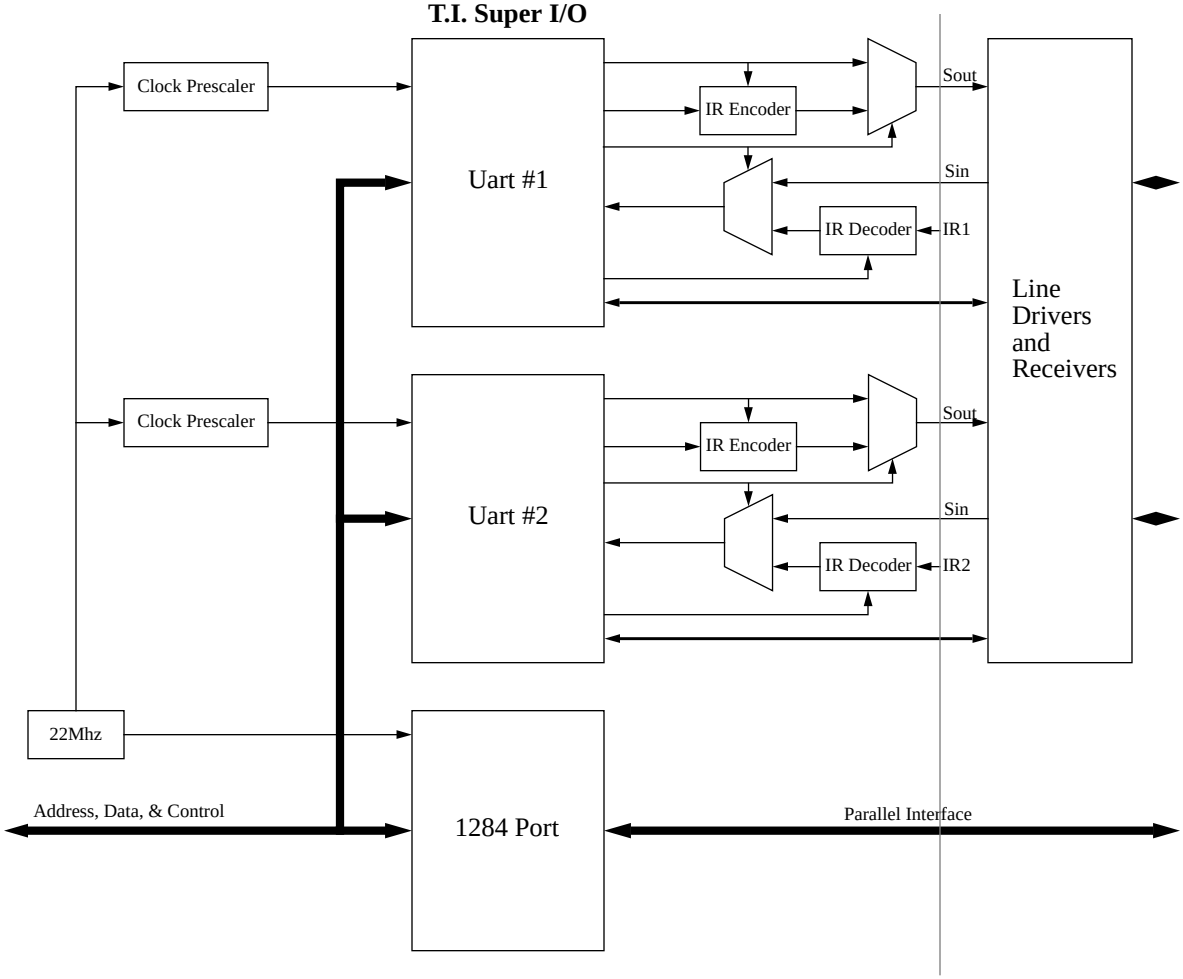
The following table shows the individual bits for interrupt status and mask registers:

CRIME Bit Slot	Index	Bit	Type	Interrupt Source
6	0	0	RO	Audio Control Channel Status Word Interrupt Request 0 - no interrupt pending 1 - at least one of the selected bits in the status word is logic '1' true
	1	1	RO	Audio Volume Control Status Change Interrupt Request 0 - no interrupt pending 1 - one or both of the volume control inputs is logic '1' true
	2	2	RO	Audio Stereo Input #1 DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	3	3	RO	Audio Stereo Input #1 DMA FIFO Overflow Error Interrupt Request 0 - no interrupt pending 1 - the internal DMA FIFO overflowed, DMA has stopped, fatal error
	4	4	RO	Audio Stereo Output #2 DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	5	5	RO	Audio Stereo Output #2 DMA Memory Error Interrupt Request 0 - no interrupt pending 1 - a crime memory error occurred during a DMA transaction, DMA has stopped, fatal error
	6	6	RO	Audio Stereo Output #3 DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	7	7	RO	Audio Stereo Output #3 DMA Memory Error Interrupt Request 0 - no interrupt pending 1 - a crime memory error occurred during a DMA transaction, DMA has stopped, fatal error
5	0	8	RO	RTC Interrupt Request
	1	9	RO	PS/2 Keyboard Interrupt Request
	2	10	RO	PS/2 Keyboard Poll Flag, Unmasked non-interrupt version of above signal
	3	11	RO	PS/2 Mouse Interrupt Request
	4	12	RO	PS/2 Mouse Poll Flag, Unmasked non-interrupt version of above signal
	5	13	RO	Count/Compare Timer #0
	6	14	RO	Count/Compare Timer #1
	7	15	RO	Count/Compare Timer #2
4	0	16	RW	ISA ECP1284 Parallel Device Interrupt Request 0 - no interrupt pending 1 - the ECP1284 core in the external Super I/O chip has requested an interrupt (edge triggered pulse)
	1	17	RO	ISA ECP1284 Parallel DMA Context A Exhausted Interrupt Request 0 - no interrupt pending 1 - parallel dma context A has been completely processed and is requesting service
	2	18	RO	ISA ECP1284 Parallel DMA Context B Exhausted Interrupt Request 0 - no interrupt pending 1 - parallel dma context B has been completely processed and is requesting service
	3	19	RO	ISA ECP1284 Parallel Memory Error Interrupt Request 0 - no interrupt pending 1 - a memory error occurred during a DMA transaction, DMA has stopped, fatal error
	4	20	RO	ISA Serial Port #1 Device Interrupt Request 0 - no interrupt pending 1 - the primary 16550 core in the external Super I/O chip is requesting an interrupt
	5	21	RO	ISA Serial Port #1 Transmit DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	6	22	RW	ISA Serial Port #1 Transmit DMA Pair Request Interrupt Request 0 - no interrupt pending 1 - a transmit DMA pair requested that an interrupt be posted
	7	23	RO	ISA Serial Port #1 Transmit DMA Memory Error Interrupt Request 0 - no interrupt pending 1 - a memory error occurred during a DMA transaction, DMA has stopped, fatal error
	8	24	RO	ISA Serial Port #1 Receive DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	9	25	RO	ISA Serial Port #1 Receive DMA Over-run Interrupt Request 0 - no interrupt pending 1 - an overrun error was indicated during a DMA transaction, DMA has stopped, fatal error
	10	26	RO	ISA Serial Port #2 Device Interrupt Request 0 - no interrupt pending 1 - the secondary 16550 core in the external Super I/O chip is requesting an interrupt
	11	27	RO	ISA Serial Port #2 Transmit DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	12	28	RW	ISA Serial Port #2 Transmit DMA Pair Request Interrupt Request 0 - no interrupt pending 1 - a transmit DMA pair requested that an interrupt be posted
	13	29	RO	ISA Serial Port #2 Transmit DMA Memory Error Interrupt Request 0 - no interrupt pending 1 - a memory error occurred during a DMA transaction, DMA has stopped, fatal error
	14	30	RO	ISA Serial Port #2 Receive DMA Threshold Interrupt Request 0 - no interrupt pending 1 - the selected ring buffer threshold interrupt condition has been reached
	15	31	RO	ISA Serial Port #2 Receive DMA Over-run Error Interrupt Request 0 - no interrupt pending 1 - an overrun error was indicated during a DMA transaction, DMA has stopped, fatal error

5.1.4 Super I/O

The external ISA bus on MACE connects to a Super I/O device manufactured by Texas Instruments. This device contains two enhanced 16C550 serial ports and one IEEE1284 parallel port. A block diagram of the chip is shown below:

T.I. Super I/O



The two enhanced 16C550 serial interfaces inside this device have an extra register that controls the IR encoder/decoders and the clock prescalers. This register is mapped to the same address as the uart internal scratch register but can only be accessed when the DLAB bit in the Line Control Register is set to a logic one. The format of this new register is shown below:

TABLE 55. 16C550 IR/CLOCK configuration register

Bits	Reset Value	Type	Description
7	X	RO	IR Encoder/Decoder select 0 - normal uart operation selected 1 - IR serial input and output paths selected
6	X	RO	Reserved
5:0	X	RO	Clock prescaler value The clock prescaler takes a 6-bit value which allows the input clock to be divided by value from 0 to 31.5 in 0.5 increments. With an input clock of 22mhz the divisor should be set to 3 to generate the 7.33mhz baud rate clock or set to 5.5 to generate the 4.00mhz MIDI baud rate clock.

5.1.5 Serial DMA Ring Buffer Registers

The following tables show the individual bits for a serial DMA channel:

TABLE 56. Channel Read Pointer Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RW	Ring buffer read pointer
4:0	0	RO	<reserved, read as zero>

TABLE 57. Channel Write Pointer Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RW	Ring buffer write pointer
4:0	0	RO	<reserved, read as zeros>

TABLE 58. Channel Control Register

Bits	Reset Value	Type	Description
10	1	RW	RESET 0 - channel active 1 - reset channel (inactive), all registers reset, interrupt output inactive, fifos flushed
9	0	RW	DMA enable 0 - channel disabled, but state not modified, just frozen 1 - channel enable and active (pointers must be setup)
8	0	RO	<reserved, read as zero>
7:5	0	RW	Interrupt threshold 000 - interrupt disabled, interrupt output at inactive level 001 - interrupt on input channel ring buffer >= 25% full (< 25% for output channels) 010 - interrupt on input channel ring buffer >= 50% full (< 50% for output channels) 011 - interrupt on input channel ring buffer >= 75% full (< 75% for output channels) 100 - interrupt on ring buffer empty 101 - interrupt on ring buffer not empty 110 - interrupt on ring buffer full 111 - interrupt on ring buffer not full
4:0	0	RO	<reserved, read as zeros>

TABLE 59. Channel Current Ring Depth Register

Bits	Reset Value	Type	Description
15:12	0	RO	<reserved, read as zeros>
11:5	0	RO	Number of 32-byte blocks in the ring buffer Computed using a subtracter: WritePointer - Readpointer. All zeros is the empty condition, all ones is the full condition.
4:0	0	RO	<reserved, read as zeros>

5.1.6 Serial DMA Ring Buffers

Each serial channel has a DMA ring buffer that is controlled by a set of channel registers. The size of all the ring buffers is fixed at 4Kbytes. The three ring buffers are all stored together in system main memory using a 32KB aligned base address supplied by the peripheral controller. Each 4KB ring buffer occupies one of eight 4KB pages within the 32KB range supplied. The four serial ring buffers occupy the last four 4KB pages in the 32KB block. Whenever the DMA engine reads or writes data from the ring it takes the value of the read or write pointer logic ORs it with the ring base address and ring ID to compute the memory address to use.

The address calculation is shown below:

Address[31:0] = BaseAddress[31:15] | RingID[2:0] | PointerOffset[11:5] | “00000”

Figure: Ring Buffer Address Calculation

TABLE 60. Ring ID

RingID	Ring Buffer
011	Scratch Area, Can be used by software
100	Serial port #1, transmit ring buffer
101	Serial port #1, receive ring buffer
110	Serial port #2, transmit ring buffer
111	Serial port #2, receive ring buffer

The DMA ring buffers for the serial controller are all uni-directional. For each channel one of the two ring pointers is controlled by the hardware and one by system software. If the channel is an input channel, the DMA engine controls the ring write pointer and it is read-only. If the channel is an output channel, the DMA engine controls the ring read pointer and it is read-only. In both cases, the other pointer is controlled by system software and it is used to tell the DMA engine how full the ring buffer is with data.

When the read and write pointers become equal the DMA engine assumes that the ring buffer is full (input channel) or empty (output channel) and hardware DMA will stop. Note that DMA operation will resume as soon as system software changes it’s pointer so that the ring is no longer full/empty. The DMA engine keeps track of how many 64-bit words are in the ring buffer. This value is readable and is used by the interrupt threshold logic.

5.2 Serial Port Operation

The external serial controller supported by the I/O asic is assumed to be a National 16550 serial core or compatible device. Two serial ports are supported by the asic along with an optional set of DMA channels for each port. The optional DMA channels provide hardware assistance for advance features such as hardware and software transmit flow control. These added hardware features can be enabled or disabled on a per serial port basis.

The ring buffer DMA for the serial controller operates on 64-bit chunks that are read from or written to the ring buffers in system main memory. Since the serial ports operate on 8-bit character streams that can be any byte length, the DMA to and from the serial ports has been packetized. The packet formats provide for automatic padding and stripping to simulate a true 8-bit character stream.

When a serial port is operated in 16650 mode, the uart can optionally perform both hardware CTS flow control of the transmit data stream and hardware RTS flow control of the receive data stream. If CTS flow control is enabled, the transmit data stream is automatically stopped and restarted, within one character time, without software intervention. If RTS flow control is enabled, the RTS output goes active whenever the internal uart FIFO is almost full.

All of the serial controllers internal registers are decoded and available in the ISA address space. This should make it fairly easy to port over a standard PC serial device driver for the *Moosehead* system (at least to start).

5.2.1 Serial Port Mode

The DMA engine for the serial ports is an optional feature that can be enabled or disabled. If the DMA engine is disabled, then the serial port behaves exactly like a standard National 16550. System software should never try to service the uart transmit and receive data interrupts itself or read the IIR, RBR, or LSR registers when the DMA engine is active. This will confuse the DMA logic and result in unexpected behavior and lost data. For proper operation, only the modem status change interrupt should be enabled when the uart is in DMA mode.

Note that when DMA is enabled on the serial ports it is recommended that the receive fifo threshold in the uarts be set to delay fifo ready requests by at least four character times. That way the receive DMA channel will have more than a single receive character to pack into a 64-bit word. For high data rates, setting the uart receive fifo character delay time to even larger values, 8 or 14 characters times, is recommended.

5.2.2 Serial DMA Format

The serial DMA engines transfer 64-bit blocks of four control bytes and four data bytes. Every time one of the DMA engines queues data to or from a ring buffer it reads or writes one or more of these pairs to or from memory. Since the ring buffers operate on 64-bit multiples, when an input DMA engine has less than four pairs to write to memory, it pads the write with pairs that are marked as invalid which software should ignore.

Similarly, when the system software queues pairs of control and data bytes to a transmit ring buffer it writes multiples of four pairs into the buffer for every 64-bit word. Since the system software may not always want to send multiples of four characters, individual transmit DMA data pairs can be marked as invalid. This tells the DMA engine to skip over (i.e. ignore) those pairs and operate on the rest.

The pairs are packed into 64-bit wide memory words in big endian format with all four control bytes in the high 32-bit position and all four data bytes in the low 32-bit position. The following picture shows the packing format:

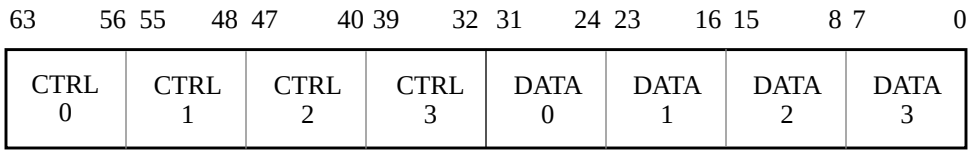


Figure: Pair Packing Format

5.2.3 Serial DMA hints

When reading data out of the serial DMA input rings, it isn't necessary to read the ring buffer write pointer register to see how far the DMA engine has written. Since the DMA engine never writes a 64-bit word into the ring that is zero, if software clears the ring entries it has read and looks for a zero word as it collects pairs, the first all zero 64-bit word marks the location after the current ring buffer write pointer. The only PIO needed by the hardware is the one to tell the DMA engine how far the system software has read in the ring buffer (one PIO write operation).

5.2.4 Serial Transmit Delay

When the hardware reads and processes the pairs for the serial transmit DMA channels, one of the possible pair types is a delay value. The hardware DMA channel will idle for the number of milliseconds indicated before going on to process any pair that follows the delay pair. This allows the system software to insert idle timing spaces in the serial output DMA data stream that are required by some terminals and are also used for MIDI note spacing.

5.2.5 Modem Control Signals

The DMA hardware needs to keep the data stream synchronized with changes to the external serial modem control signals. Since changes in these signals are rare, the easiest way to do this is to have the DMA state machine monitor the device interrupt line, and when it goes active, halt DMA and force system software to handle the transition.

5.2.6 Packet Formats

TABLE 61. DMA Input Control Byte Format

Bits	Src	Description
7	I/O	Pair is valid 0 - skip this pair (8-bit character is 0xFF) 1 - valid 8-bit character read from 16C550
6:4	None	Reserved, set to zero
3	LSR bit 4	Break Condition 0 - no break 1 - this is a break character
2	LSR bit 3	Framing Error 0 - no error 1 - this character had a framing error
1	LSR bit 2	Parity Error 0 - no error 1 - this character had a parity error
0	LSR bit 1	Overrun Error 0 - no error 1 - this character had an overrun error, DMA has stopped and a receive overflow interrupt has been posted.

TABLE 62. DMA Input Data Byte Format

Bits	Src	Description
7:0	RBR	8-bit binary character RBR contents when pair is valid 0xFF when pair is invalid

TABLE 63. DMA Output Control Byte Format

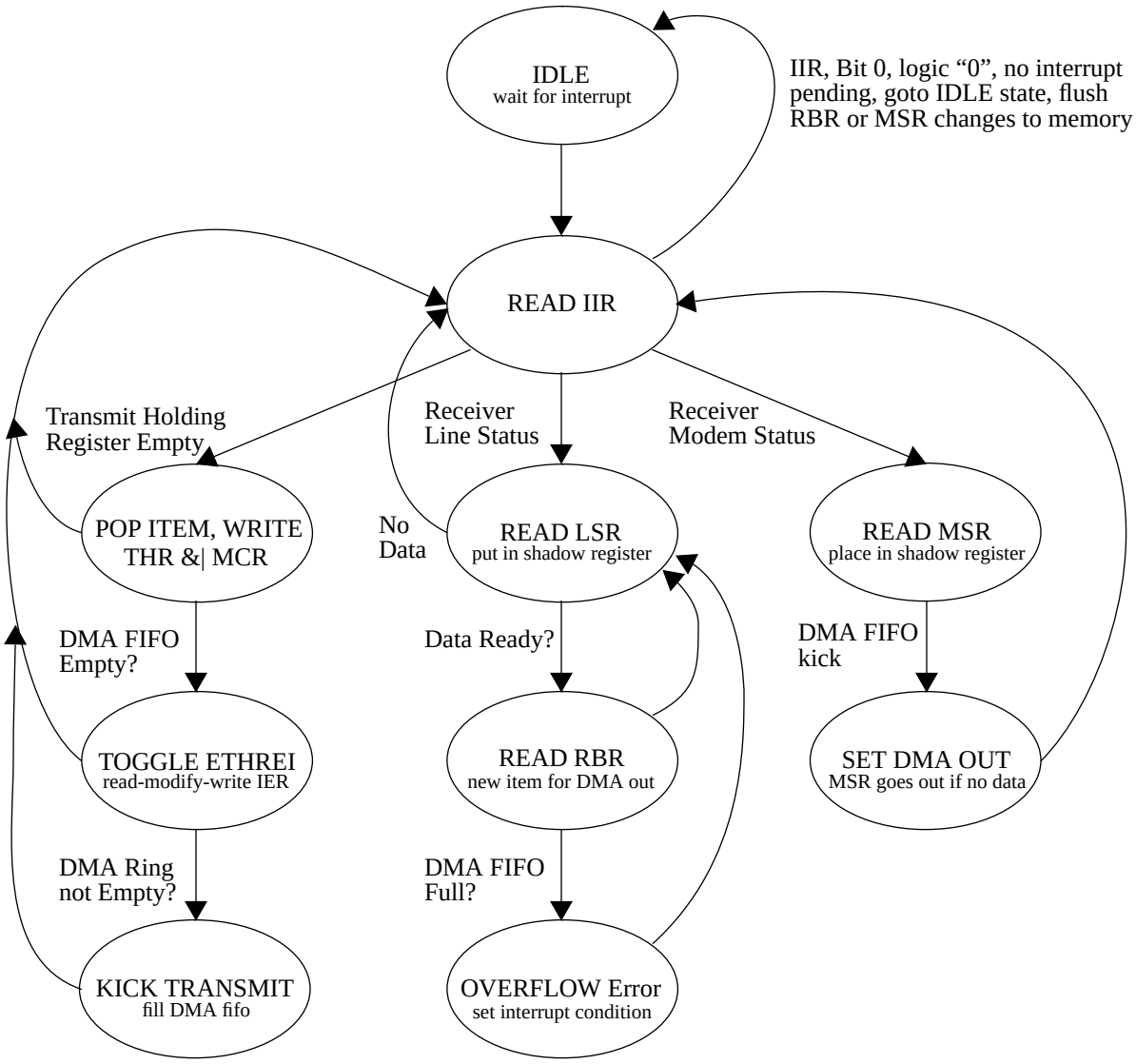
Bits	Dst	Description
7:6	I/O	Valid Command Code 00 - This pair is invalid 01 - Write data byte to THR 10 - Write data byte to MCR 11 - Time delay, 8-bit binary character is an N ms delay, DMA engine counts to zero before processing next pair in the transmit ring.
5	I/O	Post TX Interrupt 0 - no op 1 - set TX interrupt after processing this pair
4:0	None	Reserved, must be zero

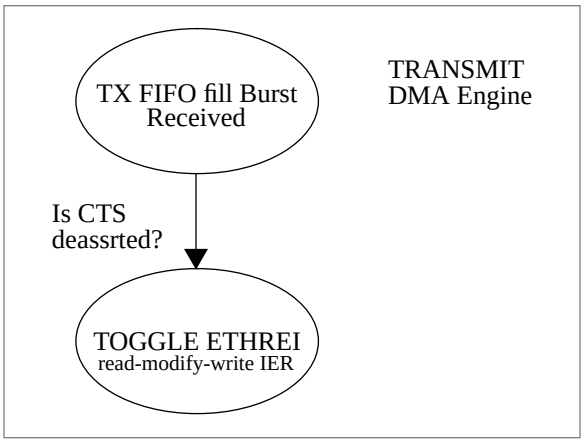
TABLE 64. DMA Output Data Byte Format

Bits	Dst	Description
7:0	THR or MCR	Data byte (use based on Valid Command Code) 00 - This pair is invalid and ignored 01 - Data[7:0] written to THR 10 - Data[7:0] written to MCR 11 - Time delay initial count down value

5.2.7 Serial State Machine Example

The following is a diagram of a typical software interrupt driven state machine:





5.3 Parallel DMA Registers

The following tables show the individual bits for the parallel DMA channel:

TABLE 65. Parallel Context Registers A and B

Bits	Reset Value	Type	Description
63	X	RW	Last Flag 0 - do nothing 1 - pulse Terminal Count to the parallel device after buffer is filled/drained
62:44	X	RO	<reserved, read as zeros>
43:32	X	RW	Byte Length of data (N - 1) Must not cross a 4K byte page boundary (i.e. base + length)
31:0	X	RW	Base Address Can be arbitrarily aligned on any byte boundary on output, 64 byte aligned on input

TABLE 66. Parallel DMA Control and Status Register

Bits	Reset Value	Type	Description
4	0	RO	Context A is Valid 0 - context is invalid 1 - context is valid (written by software) and not yet consumed by hardware
3	0	RO	Context B is Valid 0 - context is invalid 1 - context is valid (written by software) and not yet consumed by hardware
2	1	RW	RESET 0 - channel active 1 - reset channel (inactive), all registers reset, interrupt output inactive, fifos flushed
1	0	RW	DMA enable 0 - channel disabled, but state not modified, just frozen 1 - channel enabled and active
0	0	RW	DMA direction 0 - parallel output, from memory to external device 1 - parallel input, from external device to memory

TABLE 67. Parallel DMA Diagnostic Register

Bits	Reset Value	Type	Description
13:2	0	RO	Counter Number of bytes left in the context currently being processed
1	0	RO	Context processing active 0 - dma engine is idle 1 - the dma engine is currently processing the context given below
0	0	RO	Context in use 0 - Context A is currently being processed 1 - Context B is currently being processed

5.3.1 Parallel Port Interface Operation

The parallel port uses the ECP port on an external parallel/serial combo chip. This port is capable of all the standard modes supported by ECP including Centronix, PS/2, Parallel-Out with FIFO, and ECP modes. The MACE asic DMA engine supports the ECP-1284 and EPP data phase modes. All other modes must be supported by using PIO ops.

The DMA operation consists of setting up the ECP/EPP port for the proper mode and direction, then writing the address and byte count into the MACE parallel port DMA registers. Two sets of these register are provided; these are known as *Context A* and *Context B*. Each context is capable of interrupting the host independently so that the next buffer can be staged while the current one is being transferred. The LAST bit for each context is used to identify the final buffer in a chain; if LAST was set for the context being transferred, the MACE asic will signal TC (Terminal Count) to the ECP/EPP port at the end of the transfer, which causes the ECP/EPP to assert a service interrupt.

Buffers in host memory may be up to 4K bytes in size and me be arbitrarily aligned for output, but must be 64 byte aligned for input transfers. A buffer must not ever cross a 4K byte page boundary in the middle of a transfer. If a transfer needs to cross a page boundary, it should be broken up into two contexts by the software driver. Any number of contexts may be chained together by software to form a single transfer.

All of the parallel interface internal registers are decoded and available in the ISA address space. This should make it fairly easy to port over a standard PC parallel device driver for the *Moosehead* system.

Note that the parallel device can not overflow or underflow. Should the MACE internal ping-pong buffers fill up, the hardware will not respond to data transfer requests from the external ECP/EPP port (the same is true in the empty case for dma output). The hardware will automatically start transferring data again when the internal buffers drain (or are filled in the case of output). The same rules hold true if the dma engine runs out of valid contexts in the middle of a transfer, it will idle waiting until the software driver supplies another context and continue.

5.4 Calendar Clock Interface Operation

The ISA bus interface on the I/O asic supports the connection of an external Dallas Semiconductor Calendar Clock chip. The device has a decoded 256-byte address range in the ISA address space (see below). The clock chip has no DMA or other external support requirements other than the connection of it’s alarm interrupt output.

5.5 Flash-ROM Interface Operation

The ISA bus interface supports an external Flash-ROM of up to 2 Megabytes. Devices from 512 Kilobytes to 2 Megabytes may be connected to the external interface. If a smaller device is connected to the external pins, the contents of the device will be aliased to fill the entire 4 Megabyte address space.

A simple write protect register is provided so that system crashes will be less likely to erase or overwrite sections of the Flash. Before writing to the Flash-ROM, system software must enable write activity through the control register. Because of the danger of accidental erasure, it is a good idea to leave the write protect enabled at all other times.

Note that the Flash-ROM interface only supports individual byte writes. System software may only write one byte at a time to the external flash device. Reads on the other hand can be any size between one and eight bytes.

5.6 External ISA Address Map

The following table shows the address map for the external ISA bus interface:

TABLE 68. External ISA Address Map

Base Address	Offset	Description
0x00000	0x0000	Centronics/EPP Parallel Port Interface
	0x8000	ECP1284 Parallel Port Registers
0x10000	0x0000	Serial Port COM #1
	0x8000	Serial Port COM #2
0x20000	-	Dallas Calendar Clock chip
0x30000	-	Future Expansion

Note: in the internal view as seen from the CRIME asic the peripherals are distributed across the 512K byte address space allocated to the external ISA bus. Each 8-bit register spans 256 bytes in the internal view.

5.7 Software DMA Appendix

The following is a list of examples showing how to setup operations for the various ISA bus peripherals.

5.7.1 Interrupt Handler

The following example shows some of the special interrupt handling for the peripheral controller. The audio, game port, keyboard, and mouse interrupts for Moosehead will usually be serviced by the primary clock interrupt handler to save on the context switch overhead. For this reason, polled versions of the keyboard and mouse interrupts have been added to the peripheral controller interrupt status register.

```
/*
** MOOSEHEAD: Hard Clock Interrupt Handler
*/
ClockInterrupt()
{
          register unsigned PeripheralStatus;

          /* Read peripheral status register */
          PeripheralStatus = *(unsigned *)PERIPHERAL_STATUS_REGISTER;

          /* ...standard clock processing goes here... */

          /* Check for audio service condition */
          if (PeripheralStatus & AUDIO_NEEDS_SERVICE) {
                    AudioInterruptHandler();
          ]

          /* Check for keyboard service condition */
          if (PeripheralStatus & KEYBOARD_POLL_STATUS) {
                    KeyboardInterruptHandler();
          }

          /* Check for mouse service condition */
          if (PeripheralStatus & MOUSE_POLL_STATUS) {
                    MouseInterruptHandler();
          }
}
```

5.7.2 Parallel Port

The following example shows the register setup for a parallel DMA read operation:

```
/*
 * Parallel DMA setup example
 *
 * The following example shows how to use the MACE DMA hardware to
 * perform a simple parallel output DMA transfer.
 */
struct parallel_dma {
          unsigned long long      contextA, contextB;
          unsigned long long      dma_control;
#define   DMA_CONTEXTA_VALID      0x10                    /* read-only */
#define   DMA_CONTEXTB_VALID      0x08                    /* read-only */
#define   DMA_RESET               0x04
#define   DMA_ENABLE              0x02
```

```
#define    DMA_DIRECTION           0x01
           unsigned long long      diagnostic;
};

parallel_dma_output(struct buf *)
{
           register struct parallel_dma *pdp = MACE_PLP_ADDRESS;
           register unsigned long long contextA, contextB;
           register int overflow, remainder;

           /*
            * Reset parallel DMA
            */
           pdp->dma_control = DMA_RESET;
           pdp->dma_control = 0;

           /*
            * Check for page crossing (DMA page size is 4K bytes)
            */
           remainder = IO_PAGE_SIZE - pageoffset(bp->b_dmaaddr);
           overflow = (bp->b_bcount - bp->b_resed) - remainder;
           if (overflow > 0) {
                      contextA = bp->b_dmaaddr;
                      contextA |= remainder << 32;
                      contextB = bp->b_dmaaddr + remainder;
                      contextB |= overflow << 32;
                      contextB |= 1 << 63;               /* LAST flag */
           } else {
                      contextA = bp->b_dmaaddr;          /* Address */
                      contextA |= bp->b_bcount << 32;    /* Length */
                      contextA |= 1 << 63;               /* LAST flag */
                      contextB = 0LL;
           }
           pdp->contextA = contextA;
           pdp->contextB = contextB;

           /*
            * Start DMA
            */
           pdp->dma_control = DMA_ENABLE;

           return (0);
}
```

5.7.3 Serial Port

The following example shows the register setup for a serial port in DMA mode:

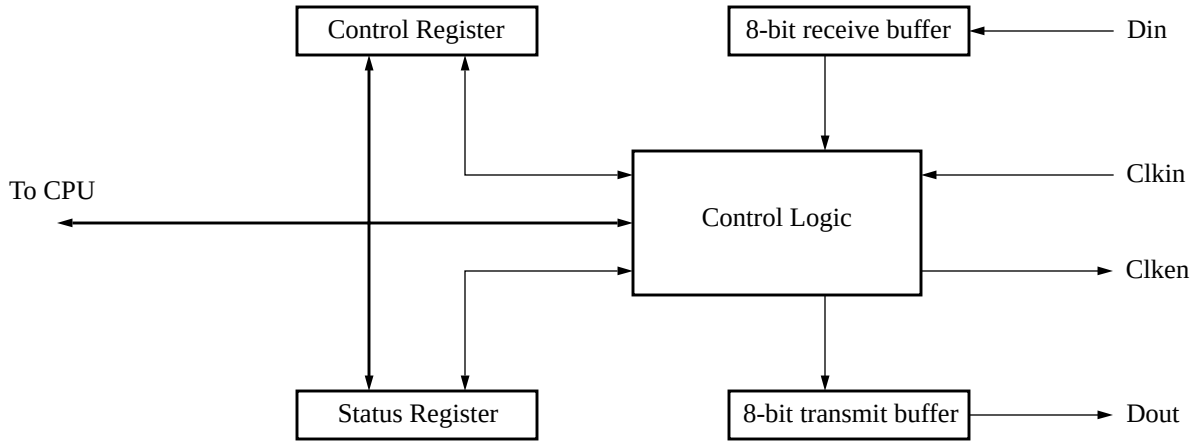
```
/*
 * Serial port DMA setup example
 */

#include “ring.h”
```

```
/* 16550 defines */
#define UART_IER_ERDAI            0x01
#define UART_IER_ETHREI           0x02
#define UART_IER_ERLSI            0x04
#define UART_IER_EMSI             0x08
#define UART_FCR_FIFO_ENABLE      0x01
#define UART_FCR_RCVFIFO_RST      0x02
#define UART_FCR_XMTFIFO_RST      0x04
#define UART_FCR_DMA_MODE         0x08
#define UART_FCR_RCV_TRIG_LSB     0x40
#define UART_FCR_RCV_TRIG_MSB     0x80
#define UART_LCR_WLS0             0x01
#define UART_LCR_WLS1             0x02
#define UART_LCR_STB              0x04
#define UART_LCR_PEN              0x08
#define UART_LCR_EPS              0x10
#define UART_LCR_STP              0x20
#define UART_LCR_BRK              0x40
#define UART_LCR_DLAB             0x80

/* One serial channels worth of registers */
volatile struct serial {
          long long   tx_control;
          long long   tx_read_ptr;
          long long   tx_write_ptr;
          long long   tx_depth;
          long long   rx_control;
          long long   rx_read_ptr;
          long long   rx_write_ptr;
          long long   rx_depth;
}:
#define SPP_MODE_DMA_ENABLE       0x200

/* Setup serial port */
serial_setup(port)
{
          register caddr_t ddl, dlm, lcr, fcr, ier, pre;
          register struct serial *spp;
          register paddr_t pp;

          /*
           * select port
           */
          if (!port) {
                    ier = (caddr_t)SERIAL_PORT_0_IER;
                    fcr = (caddr_t)SERIAL_PORT_0_FCR;
                    lcr = (caddr_t)SERIAL_PORT_0_LCR;
                    pre = (caddr_t)SERIAL_PORT_0_SCR;
                    spp = (struct serial *)SERIAL_PORT_0_DMA;
          } else {
                    ier = (caddr_t)SERIAL_PORT_1_IER;
                    fcr = (caddr_t)SERIAL_PORT_1_FCR;
                    lcr = (caddr_t)SERIAL_PORT_1_LCR;
                    pre = (caddr_t)SERIAL_PORT_1_SCR;
```

```
        }

        /*
         * DMA setup.
         */
        spp->rx_config = RING_DMA_ENABLE | RING_INTERRUPT_NEMPTY;
        spp->tx_config = RING_DMA_ENABLE | RING_INTERRUPT_OFF;

        /*
         * load operating mode into 16550
         *          baud rate set (9600 baud), 7.33Mhz / 48
         *          set prescaler to 3 for normal baud rates
         *          character format set (8 bits, no parity, 1 stop)
         *          FIFOs enabled, receive trigger at 8 bytes threshold
         *          all interrupts enabled
         */
        *lcr = UART_LCR_DLAB;
        *ddl = 48;
        *dlm = 0;
        *pre = 3;                           /* Prescale is 3, IR off */
        *lcr = UART_LCR_WLS0 | UART_LCR_WLS1;
        *fcr = UART_FCR_FIFO_ENABLE | UART_FCR_RCV_TRIG_MSB;
        *ier = UART_IER_ERDAI | UART_IER_ETHREI |
                UART_IER_ERLSI | UART_IER_EMSI;

        return (0);
}
```

6 PS/2 Keyboard & Mouse Interface

The *Moosehead* system I/O asic contains a simple PS/2 interface that supports both PS/2 style PC keyboards and mice. The interface supports two connections that are intended to be used, one for the keyboard and one for the mouse, but each interface could be either device. Note that the interface only supports PS/2 devices. No attempt has been made to support AT style keyboards. A block diagram of one the of the PS/2 interfaces is shown below:



Overview:

- Serial port that conforms to IBM PS/2 Keyboard and Auxiliary (type 1) Serial Port Protocol
- Supports PS/2 keyboard, mouse, touch-pad, and track-ball type devices
- Supports input clock rates as defined by the IBM PS/2 interface specification
- External open-collector buffer used for isolation from the PS/2 +5.0V interface

6.1 PS/2 Interface

The IBM PS/2 interface standard supports data transmission to and from the external device using an 11-bit data stream sent serially over the data line. The table below shows the function of each bit:

TABLE 69. PS/2 Serial Port Bits

Bits	Function
1	Start bit (always 0)
2	Data bit 0 (LSB)
3-8	Data bits 1 - 6
9	Data bit 7 (MSB)
10	Parity bit (odd parity)
11	Stop bit (always 1)

note that the PS/2 ports use odd parity (the eight data bits plus the parity bit always have an odd number of ones).

6.2 Register Programming Interface

The following table shows all of the PS/2 interface registers. All bits not explicitly defined are read as zeros. All registers are defined on 64-bit aligned boundaries and can be read or written using 64-bit programmed i/o operations.

TABLE 70. PS/2 Interface Registers

Offset	Register Name	Type	Bits	Function
0x00	Transmit Buffer	WO	7:0	Transmit 8-bit parallel shift out data buffer
0x08	Receive Buffer	RO	15:0	Receive 8-bit parallel shift in data buffer
0x10	Control	RW	5:0	Command and control register
0x18	Status	RO	7:0	Transmit & receive status and error register

Note: see the I/O chip appendix for the addresses of these registers.

6.2.1 PS/2 Transmit Buffer

The PS/2 serial port transmit buffer is an eight-bit parallel load shift register. The transmit buffer transmits a serial data stream to the PS/2 interface via the Dout pin. The following register is write only:

TABLE 71. Transmit Buffer Register

Bits	Reset Value	Type	Description
7:0	0	WO	This 8-bit field contains the bits currently being transmitted out of Dout

6.2.2 PS/2 Receive Buffer

The PS/2 serial port receive buffer is an eight-bit serial in parallel out shift register. The receive buffer receives a serial data stream from the PS/2 interface via the Din pin. The following register is read only:

TABLE 72. Receive Buffer Register

Bits	Reset Value	Type	Description
15:8	0	RO	Alias of transmit & receive status and error register bits
7:0	0	RO	This 8-bit field contains the bits currently being received from Din. The control logic ensures that an auxiliary device does not send another serial stream until this buffer has been cleared and there is no data to transmit via Dout.

6.2.3 PS/2 Control Register

The control register controls the PS/2 serial port operation:

TABLE 73. Command and Control Register

Bits	Reset Value	Type	Description
0	0	RW	Inhibit Clock after Transmission 0 - Keep Clken asserted 1 - De-assert Clken after transmitting the next serial data stream
1	0	RW	Transmit Enable 0 - Disable timeout. 1 - Enable shifting the data from the transmit buffer out the Dout pin and the negative edge of Clkin.
2	0	RW	Transmit Interrupt Enable 0 - Interrupt disabled 1 - Transmit interrupt will be asserted when the transmit buffer is empty
3	0	RW	Receive Interrupt Enable 0 - Interrupt disabled 1 - Receive interrupt will be asserted when the receive buffer is full
4	1	RW	Clock Inhibit 0 - De-assert Clken (i.e. pause current transmit of receive operation) 1 - The serial port will assert Clken
5	0	RW	Reset 0 - reset inactive 1 - internal state machines in reset state

6.2.3.1 Clock Inhibit

The PS/2 serial port interface uses a 1X serial data clock that is always supplied by the auxiliary device. The data clock travels on an open-collector signal that the command & control register can pull high at any time using the Clken output signal. When the data clock is de-asserted both the PS/2 serial port and the external auxiliary device stop sending and receiving data (depending on which was active at the time). The external auxiliary device is required to poll the serial data clock signal at 100us intervals. This implies that system software should always allow for some settling time when changing the Clken output signal.

6.2.4 PS/2 Status Register

The control register allows the software to monitor the operation of the PS/2 serial port. This register is read only:

TABLE 74. Status Register

Bits	Reset Value	Type	Description
0	X	RO	Clock signal 0 - external clock signal is de-asserted 1 - external clock signal is asserted
1	1	RO	Clock Inhibit 0 - Clken output signal is de-asserted 1 - Clken output signal is asserted
2	0	RO	Transmission in Progress 0 - transmitter idle 1 - data is in the process of being transmitted out the Dout pin
3	0	RO	Transmit Buffer Empty 0 - transmit buffer holds byte waiting to be sent 1 - transmit buffer empty
4	0	RO	Receive buffer full 0 - receive buffer empty 1 - receive buffer holds byte waiting to be read
5	0	RO	Reception in progress 0 - receiver idle 1 - serial port is in the process of receiving data
6	0	RO	Parity error 0 - no parity error 1 - indicates a parity error occurred on the last received byte
7	0	RO	Framing error 0 - no framing error 1 - indicates a framing error occurred on the last received byte

6.3 Receiving Serial Data on the PS/2 interface

<xxx>.

6.4 Transmitting Serial Data on the PS/2 interface

<xxx>.

6.5 Verilog source code

The following example source code is available from - rowan:/d1/moosehead/subsystem/io/doc/mace_spec

```
/* ps2 ports */

include lcb007;

const ctransmit_buf : 0b00;
const creceive_buf : 0b01;
const cstatus_reg : 0b10;
const ccommand_reg : 0b11;
```

```
extern module bcount3;
input cp0, sd, cd;
output q[3];
end;

const start_bit : 0;
const stop_bit : 1;

module ps;
input ps2_wr, ps2_rd, regadr[2],
                aux_clkin_i0/re, aux_datain_i1/re,
                rst/reset, pclkp/clock;
output ps2_int, aux_dataout_o1, aux_clken_o0, ps2_rcvint, ps2_txint;

bidirect cdatap[8];

register status[8]/default=0b00001000, command[8]/default=0b00010000,
/*              parityerr, rcvin, rcvbf, clkinhb, txmitn, txbmt,
                clk, /* status bits */
                clkinh, rcvint, txint, txen, inhtx, /* command bits */ */
                aux_clkin1, aux_clkin2, parity, txclk, rcvclk, aux_dataout, aux_clken;

signal fedge, txenable, txcomp, set_bitcount, tx_clk,
                zero_bitcount, bit_count[3],
                rcvbuf[8], txbuf[8] , framerr <=> status[7],
                parityerr <=> status[6], rcvin <=> status[5], rcvbf <=> status[4],
                txbmt <=> status[3],
                txmitn <=> status[2], clkinhb <=> status[1], clk <=> status[0],
                clkinh <=> command[4], rcvint <=> command[3], txint <=> command[2],
                txen <=> command[1], inhtx <=> command[0],
                shft_out[8], shft_in[8], wrtxbf, cdata_op[8], io_mode;


var i;

default
/*              command[7] : false; /* io_mode */
                command[6] : false; /* unsused */
                command[5] : false; /* unsused */
                command[4] : true; /* clkinh : false */
                command[3] : false; /* rcvint : false */
                command[2] : false; /* txint : false; */
                command[1] : false; /* txen : false; */
                command[0] : false; /* inhtx : false; */
                status[5] : false; /* rcvin : false; */
                status[4] : false; /* rcvbf : false; */
                status[3] : true; /* txbmt : false; */
                status[2] : false; /* txmitn : false; */ */
                aux_dataout : true;
                txcomp : false;
                set_bitcount : false;
                wrtxbf : false;

relations
```

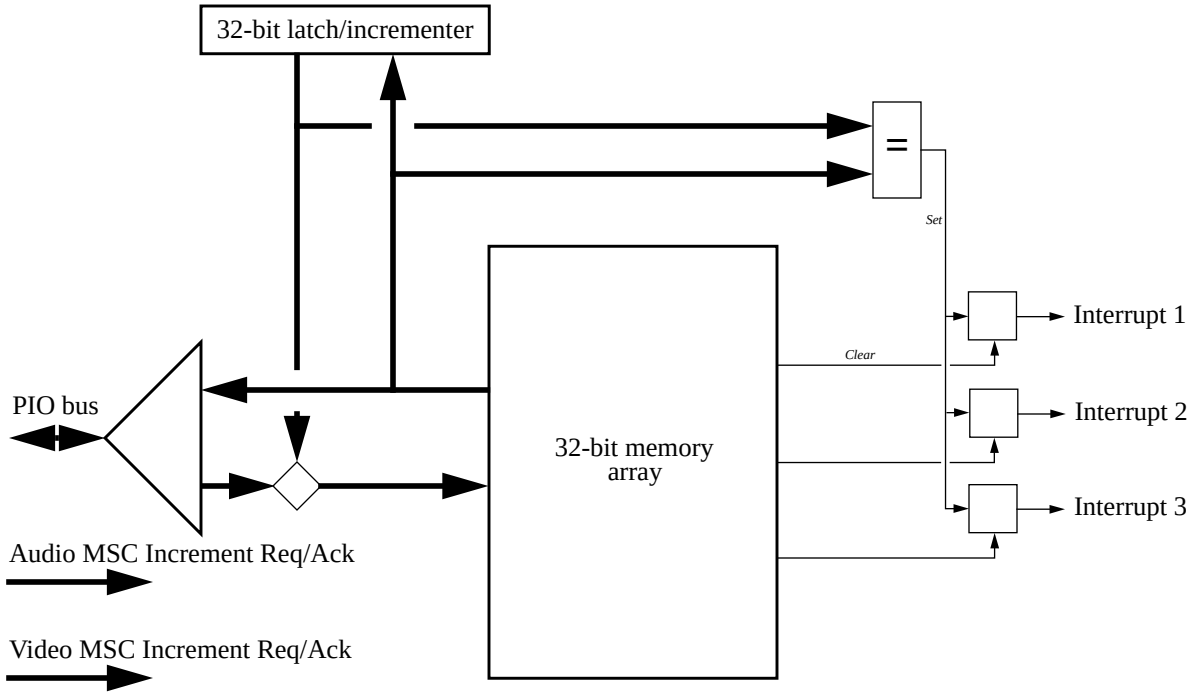
```
{
        io_mode = command[7];
        aux_clkin1 = aux_clkin_i0;
        aux_clkin2 = aux_clkin1;
        clk = aux_clkin1;
        fedge = !aux_clkin_i0 && !aux_clkin1 && aux_clkin2;

        aux_clken = !(clkinh || rcvbf && !txmitn && (!aux_datain_i1 || aux_clken)
                || !txmitn && inhtx);
        aux_clken_o0 = aux_clken && !io_mode || command[0] && io_mode;
        clkinhb = !aux_clken && !io_mode || aux_datain_i1 && io_mode;
        txenable = !txbmt && txen;
        ps2_int = (txint && txbmt || rcvint && rcvbf) && !io_mode;
        ps2_txint = txint && txbmt && !io_mode;
        ps2_rcvint = rcvint && rcvbf && !io_mode;

        zero_bitcount = (bit_count==0);

        shft_out = txbuf[0] // txbuf[7:1];
        shft_in = aux_datain_i1 // rcvbuf[7:1];

        tx_clk = wrtxbf || txclk;

        aux_dataout_o1 = aux_dataout && !io_mode || command[1] && io_mode;
}

always
{
        if (ps2_rd)
        switch (regadr) {
        case ccommand_reg :
                cdata_op = command;
        case cstatus_reg :
                cdata_op = status;
        case creceive_buf : {
                rcvbf = false;
                cdata_op = rcvbuf;
                }
        }
        if (ps2_wr)
        switch (regadr) {
        case ctransmit_buf : {
                txbmt = false;
                wrtxbf = true;
                }
        case ccommand_reg :
                command = cdatap;
        }
}

structure
{
        bcount(bit_count) = bcount3((rcvclk || txclk), !set_bitcount, !rst);
        drvmdata(cdatap) = 8 * bts4a(cdata_op, ps2_rd);
```

```
		#for (i := 0; i < 8; i := i+1)
				tbuf#i(txbuf[i]=q) = fd1sa(cdatap[i], tx_clk, shft_out[i], txmitn);

		#for (i := 0; i < 8; i := i+1)
				rbuf#i(rcvbuf[i]=q) = fd1sa(cdatap[i], rcvclk, shft_in[i], rcvin);
}

state idle/start;
{
		set_bitcount = true;
		if ( io_mode )
				goto idle;
		if ( txenable) {
				txmitn = true;
				goto txtdata;
		}
		 if (aux_clken_o0 && fedge && !aux_datain_i1 && aux_dataout) {
				rcvin = true;
				goto rcvdata;
		}
		rcvclk = false;
		txclk = false;
		aux_dataout = 1;
		rcvin = false;
		txmitn = false;
		goto idle;
}

state txtdata;
{
		aux_dataout = start_bit;
		parity = 1;
		while (!fedge && !(clkinh & !txen))
				state txstart;
		while (!zero_bitcount && !clkinh) {
				state dataop;
				aux_dataout = txbuf[0];
				parity = parity ^ txbuf[0];
				txclk = false;
				while (!fedge && !zero_bitcount && !clkinh) {
						state txdata_edge;
				}
				txclk = fedge;
		}
		do {
				state txpar_edge;
				txclk = false;
		} while (!fedge && !clkinh);
		aux_dataout = parity;
		do {
				 state txstop_edge;
		} while (!fedge && !clkinh);
		aux_dataout = stop_bit;
		do {
```

```
                        state lcntrl;
        } while (!(fedge && !aux_datain_i1 && !clkinh));
        txcomp = true;
        txbmt= true;
        goto idle;
}

state rcvdata;
{
        parityerr = 1;
        while (!zero_bitcount && !clkinh) {
                        state datain;
                        rcvclk = false;
                        while (!fedge && !clkinh) {
                                        state rcvdata_edge;
                        }
                        parityerr = aux_datain_i1 ^ parityerr;
                        rcvclk = fedge;
        }
        do {
                        state rcvpar_edge;
                        rcvclk = false;
        } while (!fedge && !clkinh);
        parityerr = aux_datain_i1 ^ parityerr;
        do {
                        state rcvstop_edge;
        } while (!fedge && !clkinh);
        framerr = !aux_datain_i1;
        rcvbf = true;
        goto idle;
}

end;
```

7 Counter & Timers

The *Moosehead* system I/O asic contains a set of counter timers for timer base and interrupt event generation. The subsystem contains a single 32-bit read-only counter that increments once every 960 nanoseconds. It also contains an array of three 32-bit event registers that generate interrupts when the lower 32-bits of the read-only counter equals the value in the register. A block diagram of the counter/timer subsystem is shown below:



Features:

- A 32-bit 960ns resolution read-only incrementing counter as the common time base
- Three 32-bit 960ns resolution read/write compare registers with individual interrupt outputs
- Interrupt output flop is set to logic true when register value equals lower 32-bits of time base
- Interrupt output flop is reset when the corresponding register value is written
- Interrupt flops reset to logic zero at power up reset time
- Twelve 32-bit registers for Audio and Video MSC/UST count & timestamp storage

7.1 Count Compare Timers

When the system powers up and resets, the 32-bit time base and interrupt flops are reset to logic zero. The time base will start to count immediately at 960ns intervals. The three counter compare registers operate independently and can be set by the system software to generate interrupts at particular time base values. The 32 bit range of the time base limits interrupt events to 68.718 minutes maximum.

To setup one of the compare registers to generate an event the system software should read the time base and then add the desired time delta to it and truncate the result to 32-bits. The result can then be written into the selected compare register which will latch it’s level sensitive interrupt when that time value is reached. Note that the system software also needs to mask and unmask the interrupt in the master interrupt mask register (see Interrupt Specification). The interrupt latch for a compare register can be cleared by writing a new value to the compare register.

7.2 Register Programming Interface

The following table shows all of the timer interface registers. All bits not explicitly defined are read as zeros. All registers are defined on 64-bit aligned boundaries and can be read or written using 64-bit programmed i/o operations.

TABLE 75. Timer Interface Registers

Offset	Register Name	Type	Bits	Function
0x00	Universal System Time	RW	31:0	UST master uptime counter (960ns period)
0x08	Compare Timer #1	RW	31:0	Value of interrupt generation compare register #1
0x10	Compare Timer #2	RW	31:0	Value of interrupt generation compare register #2
0x18	Compare Timer #3	RW	31:0	Value of interrupt generation compare register #3
0x20	Audio Input UST	RW	31:0	Audio Input last snapped UST value
0x24	Audio Input MSC	RW	31:0	Audio Input last MSC value
0x28	Audio Output #1 UST	RW	31:0	Audio Output #1 last snapped UST value
0x2C	Audio Output #1 MSC	RW	31:0	Audio Output #1 last MSC value
0x30	Audio Output #2 UST	RW	31:0	Audio Output #2 last snapped UST value
0x34	Audio Output #2 MSC	RW	31:0	Audio Output #2 last MSC value
0x38	Video Input #1 UST	RW	31:0	Video Input #1 last snapped UST value
0x3C	Video Input #1 MSC	RW	31:0	Video Input #1 last MSC value
0x40	Video Input #2 UST	RW	31:0	Video Input #2 last snapped UST value
0x44	Video Input #2 MSC	RW	31:0	Video Input #2 last MSC value
0x48	Video Output UST	RW	31:0	Video Output last snapped UST value
0x4C	Video Output MSC	RW	31:0	Video Output last MSC value

7.2.1 Paired MSC/UST atomic reads

The register programming interface for the Timer block supports a 64-bit atomic read of a MSC and UST pair. This is done by simply supplying the UST in the high 32 bits for any “mod8 = 0” 64-bit register read. When CRIME sends a read command down to the Timer block and it asks for offset 0x20, the 64-bit value returned will contain both the MSC at that offset and the corresponding UST at offset 0x24 (UST bits 63-32, MSC bits 31-0).

7.2.2 Theory of Operation

The Timer block operates off of the 33.33$\overline{3}$Mhz PCI bus clock. The PCI clock is divided down by 32 and used as the base of a 32 entry cycle. System software can think of the state machine cycle as having 32 assigned time slots. Each time slot is reserved for a specific action and the order of the slots preserves the atomic nature of the interface.

The list of actions and the number of cycles reserved for each (32 cycles total):

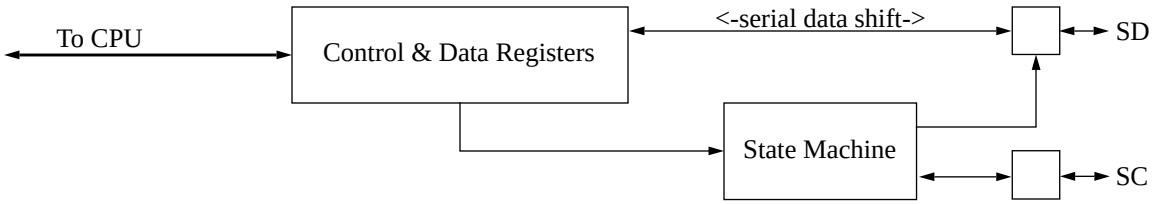
..... PIO (2)
..... UST increment (2)
..... Interrupt compare register checks (4)
.....Audio Input MSC/UST update (4)
.....Audio Output #1 MSC/UST update (4)
.....Audio Output #2 MSC/UST update (4)
..... Video Input #1 MSC/UST update (4)
..... Video Input #2 MSC/UST update (4)
..... Video Output MSC/UST update (4)

Note that the above action list shows that a PIO read or write can never happen in the middle of the update action list.

8 I^{2}C Bus Interface

The *Moosehead* system I/O asic contains two I^{2}C interfaces for the control of external codecs. The I^{2}C interface is a multimaster open-collector party-line serial bus. The arbitration protocol to take control of the bus to transmit data is implemented both by hardware and a software backoff retry mechanism. All messages sent on the bus are acknowledged so that the sender knows if a message was corrupted.

The I^{2}C interface in the *Moosehead* system I/O asic is functioanly identical to implementation in the *INDY* VINO asic. The software interface consists of two 8-bit registers, a control and status register and a data register. This implementation only supports the fast mode 400khz and standard mode 100Khz data rates defined in the original I^{2}C specification published by Philips Semiconductor Incorporated.



Highlights:

- Minimal design (approximately 500 gates)
- Collision detect implemented in hardware
- Ability to send multiple bytes as atomic message
- New feature control to support 400khz data rates

8.1 Registers

The two I^{2}C interface registers are described below. Note that the interface provided here is a minimal one. None of the higher level I^{2}C interface functions such as address recognition, transmit backoff retry, or remote reset are implemented in hardware by this design.

8.1.1 Control and Status

The control and status register contains the state and status signals needed by the software to control the I^{2}C party line serial bus. It allows selection of the data rate, whether to send or receive, and various bits used to arbitrate the bus. The most important function of the register is transmit arbitration. To transmit a message the following sequence should be followed:

- (1) Check to see if the bus is idle by reading Force Idle State Control
- (2) If it is begin transmittion, goto state six
- (3) Write a zero to Force Idle State Control to request the bus
- (4) Wait for the bus to go idle (reading as in step one), but timeout if > 400us passes and not idle
- (5) Write a sero to Bus Direction Control to enter write mode
- (6) Set Last Byte Control (0 is last byte to send, 1 otherwise) and then write data to data register
- (7) Repeat step six until all data sent and Last Byte Control set to zero
- (8) Read status bits to check for failure

The following table shows the registers that make up the I^2C interface:

TABLE 76. I^2C Interface Registers

Offset	Register Name	Type	Bits	Function
0x00	Config & Reset	RW	5:0	Bus configuration and reset
0x08	<reserved>	RW	5:0	<alias of register above>
0x10	Control & Status	RW	7:0	Control & Status Flags (same as VINO)
0x18	Data	RW	7:0	Data Transfer Register (same as VINO)

The following table shows the bit fields for the control and status register:

TABLE 77. Control & Status Register Format

Bits	Reset Value	Type	Description
0	0	RW	Force idle state control 1 - register write operation - no effect 0 - register write operation - force idle state 1 - register read operation - not idle 0 - register read operation - idle
1	0	RW	Bus direction control 1 - read data 0 - write data
2	0	RW	Last byte control 1 - more bytes hold onto bus 0 - last byte release bus
3	0	RO	<reserved>
4	0	RO	Transfer status 1 - transfer busy 0 - transfer done
5	0	RO	Acknowledge status 1 - acknowledge not received 0 - acknowledge received
6	0	RO	<reserved>
7	0	RO	Bus error status 1 - bus error 0 - no bus error

Note that bits three and six of the control and status register are reserved to be compatible with VINO.

The data register is used to read and write byte values that are received and sent on the external serial data bus.

TABLE 78. Data Register Format

Bits	Reset Value	Type	Description
7:0	0	RW	Bus data write to this address initiates a write cycle if bit 1 of the I^2C Control register is 1, a read of this address initiates a new read cycle.

The following tables shows the bit fields of the new configuration and status register:

TABLE 79. Configuration & Reset Register Format

Bit	Reset Value	Type	Description
0	0	RW	RESET (new extension) 0 - reset inactive 1 - reset active (internal state machines reset)
1	0	RW	Fast mode enable (new extension) 1 - enable 400khz data rate 0 - use standard 100khz data rate (default)
2	0	RW	Data pin override (new extension) 1 - pull external data signal low 0 - no effect
3	0	RW	Clock pin override (new extension) 1 - pull external clock signal low 0 - no effect
4	X	RO	Data input current value
5	X	RO	Clock input current value

8.2 VHDL source code

The following example source code is available from - rowan:/d1/moosehead/subsystem/io/doc/mace_spec/iic.vhdl

```
library sgi;
use sgi.sgi_logic.all;

entity iic is port(
        -- clock, read & write strobes, force to idle, read enable,
        -- more bytes to transfer, speed, reset
        Clk, iic_rd, iic_wr, force_idle_n, rd_ena, more,
                Reset_n, Scan_en : in mvl5w;
        -- busy signal, transfer done, ack received, arbitration failure,
        -- bus error
        not_idle, xfer_done, ack_rcv_n, bus_err : out mvl5w;
        -- host input data
        iic_wbyte : in mvl5w_vector(7 downto 0);
        -- host output data
        iic2hdata : out mvl5w_vector(7 downto 0);
        -- serial inputs
        scl_i, sda_i : in mvl5w;
        -- serial outputs
        scl_o, sda_o : out mvl5w;
        -- sunrise
        atpg_i2d_o : out mvl5w );
end iic;

architecture BEHAVIOR of iic is
--      use work.utils.all;

        component FD1S
```

```
                        port ( D : in mvl5w;
                         T : in mvl5w;
                         Q : out mvl5w;
                         QC: out mvl5w
                         );
        end component;


        component K1NQ
        port(
                        A : in mvl5w;
                        Y : out mvl5w
         );
        end component;

        type BR_TYPE is (RDY2XFER, XFERING, ACKING, BYTE_RDY);
        type II_TYPE is (IDLE, START1, START2, B_CNFLK, BT1, BT2, BT3, BT4,
                        ACK1, ACK2, ACK3, ACK4, STOP1, STOP2, STOP3, STOP4);
        signal i_clk0, iic_h_rd, iic_h_wr, force_idle : BOOLEAN;
        signal i_clk, i_clk_buf, ack_rcv0, bit_da : mvl5w;
        signal rd_byte : mvl5w_vector(7 downto 0);
        signal br_sm : BR_TYPE;
        signal ii_sm : II_TYPE;
        signal bit_cnt : INTEGER range 0 to 7;
        -- sunrise
        signal nc: mvl5w;
begin

-- synopsys dc_script_begin
-- dont_touch K1NQ
-- synopsys dc_script_end

        buff1: K1NQ
        port map ( A => i_clk,
                        Y => i_clk_buf
                        );

-- synopsys dc_script_begin
-- dont_touch FD1S
-- synopsys dc_script_end

        --
        -- sunrise latch
        --
        atpgi2d: FD1S
                        port map (
                                        D => ack_rcv0,
                                        T => i_clk_buf,
                                        Q => atpg_i2d_o,
                                        QC => nc);


        VREG: process(Reset_n, Clk)
                        variable v_cnt : INTEGER range 0 to 127;
```

```
begin


        if Reset_n = '0' then
                        v_cnt := 0;
                        i_clk0 <= false;
        elsif Clk'event and Clk = '1' then
                -- make clock no more than 400 KHz, divide by 128
                -- changed to GIO clk
                v_cnt := (v_cnt + 1) mod 128;
                i_clk0 <= v_cnt / 64 = 1;
        end if;
end process VREG;
i_clk <= Clk when Scan_en = '1' else '1' when i_clk0 else '0';
IREG: process(Reset_n, i_clk_buf)
        variable bc0 : INTEGER range 0 to 7;
begin
        if Reset_n = '0' then
                br_sm <= RDY2XFER;
                ii_sm <= IDLE;
                bit_cnt <= 0;
                ack_rcv0 <= '0';
                rd_byte <= "00000000";
                scl_o <= '1';
                sda_o <= '1';
        elsif i_clk_buf'event and i_clk_buf = '1' then


        -- byte ready state machine
        if force_idle then
                br_sm <= RDY2XFER;
        else
                br_sm <= br_sm; -- default
                case br_sm is
                when RDY2XFER => -- ready to transfer byte on iic bus
                        if (rd_ena = '1') or (iic_h_wr) then
                                br_sm <= XFERING;
                        end if;
                when XFERING => -- transfer in progress
                        if ii_sm = ACK1 thenbr_sm <= ACKING;
                        end if;
                when ACKING => -- in acknowledge phase
                        if (ii_sm = ACK4) or (ii_sm = STOP1) then
                                if rd_ena = '1' then
                                                br_sm <= BYTE_RDY;
                                else            br_sm <= RDY2XFER;
                                end if;
                        end if;
                when BYTE_RDY => -- byte is ready to read back
                        if iic_h_rd thenbr_sm <= RDY2XFER;
                        end if;
                end case;
        end if;
```

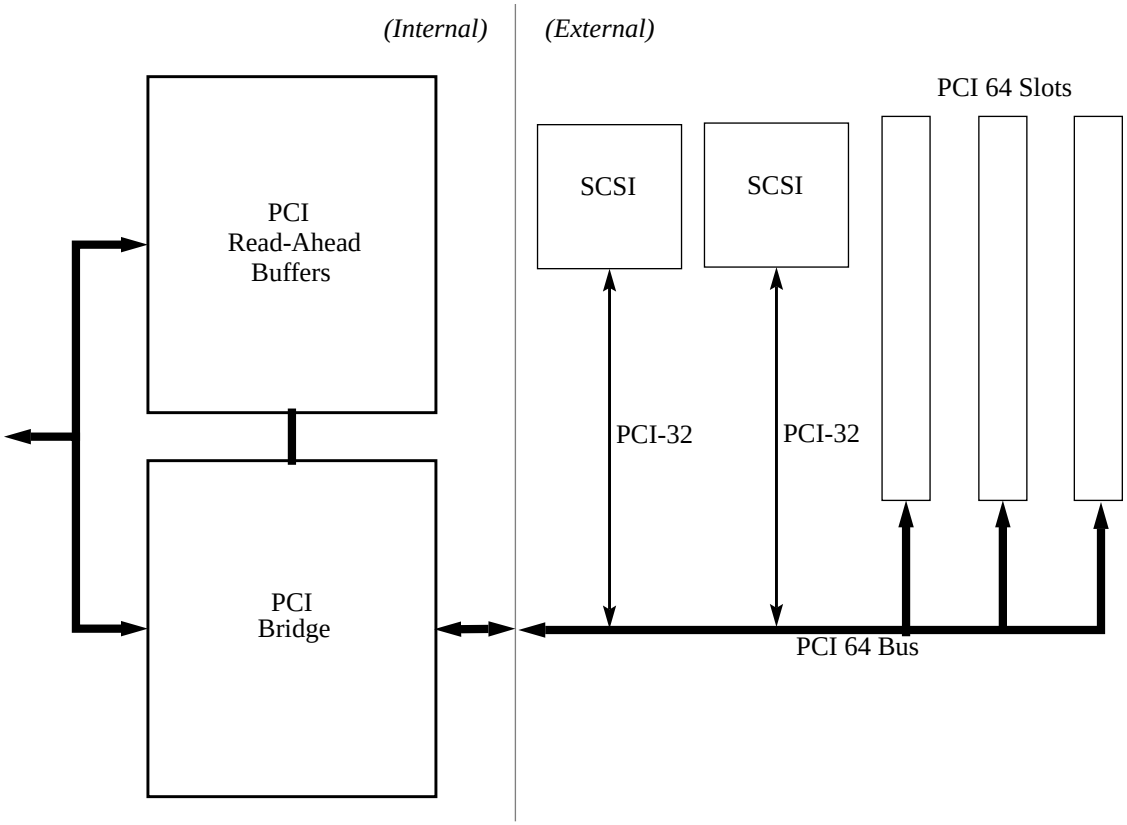
```
-- The big machine
scl_o <= '1';
sda_o <= '1';
if force_idle then
        ii_sm <= IDLE;
        ack_rcv0 <= '1';
else
        ack_rcv0 <= ack_rcv0;
        case ii_sm is
        when IDLE =>
                if (scl_i = '1') and (sda_i = '1') and
                        (br_sm = XFERING) then
                                ii_sm <= START1; sda_o <= '0';
                end if;
        -- issue the start sequence
        when START1 =>scl_o <= '0'; sda_o <= '0';
                        ii_sm <= START2;
        when START2 =>scl_o <= '0';
                        sda_o <= rd_ena or iic_wbyte(7);
                        bit_cnt <= 7; ii_sm <= BT1;
        -- transfer one bit at a time
        when BT1 =>
                if (rd_ena = '0') and (sda_i /= bit_da) then
                        ii_sm <= B_CNFLK;
                else    sda_o <= bit_da; ii_sm <= BT2;
                end if;
        when BT2 =>
                if (rd_ena = '0') and (sda_i /= bit_da) then
                        ii_sm <= B_CNFLK;
                else    sda_o <= bit_da; ii_sm <= BT3;
                end if;
        when BT3 =>
                if (rd_ena = '0') and (sda_i /= bit_da) then
                        ii_sm <= B_CNFLK;
                elsif scl_i = '1' then
                        scl_o <= '0'; sda_o <= bit_da;
                        ii_sm <= BT4; rd_byte(bit_cnt) <= sda_i;
                else    sda_o <= bit_da;
                end if;
        when BT4 =>
                scl_o <= '0';
                if bit_cnt = 0 then
                        ii_sm <= ACK1;
                else    bc0 := bit_cnt - 1;
                        sda_o <= rd_ena or iic_wbyte(bc0);
                        bit_cnt <= bc0;
                        ii_sm <= BT1;
                end if;
        -- read and check the acknowledge
        when ACK1 =>ii_sm <= ACK2;
        when ACK2 =>ii_sm <= ACK3;
        when ACK3 =>
                if scl_i = '1' then
                        if (sda_i = '0') and (more = '1') then
```

```
                                                scl_o <= ‘0’; ii_sm <= ACK4;
                                        elsif (sda_i = ‘1’) or (more = ‘0’) then
                                                scl_o <= ‘0’; ii_sm <= STOP1;
                                        end if;
                                        ack_rcv0 <= sda_i;
                                end if;
                        when ACK4 =>
                                scl_o <= ‘0’;
                                if br_sm = XFERING then
                                        sda_o <= rd_ena or iic_wbyte(7);
                                        bit_cnt <= 7; ii_sm <= BT1;
                                end if;
                        -- issue the stop sequence
                        when STOP1 =>scl_o <= ‘0’; sda_o <= ‘0’;
                                        ii_sm <= STOP2;
                        when STOP2 =>sda_o <= ‘0’; ii_sm <= STOP3;
                        when STOP3 =>
                                if scl_i = ‘1’ then ii_sm <= STOP4;
                                else        sda_o <= ‘0’;
                                end if;
                        when STOP4 =>
                                if scl_i = ‘0’ then
                                        sda_o <= ‘0’; ii_sm <= STOP3;
                                elsif (scl_i = ‘1’) and (sda_i = ‘1’) then
                                        ii_sm <= IDLE;
                                end if;
                        -- hang here for a bus error
                        when B_CNFLK =>null;
                        end case;
                end if;


                end if;
end process IREG;
-- latch of iic read pulse
IHR: process (iic_rd, i_clk_buf, br_sm, Scan_en, Reset_n)
begin
                if Reset_n = ‘0’ then
                        iic_h_rd <= false;
                elsif (iic_rd = ‘1’) and (Scan_en = ‘0’) then
                        iic_h_rd <= true;
                elsif i_clk_buf’event and i_clk_buf = ‘1’ then
                        if br_sm /= BYTE_RDY theniic_h_rd <= false;
                        end if;
                end if;
end process IHR;
-- latch of iic write pulse
IHW: process (iic_wr, i_clk_buf, br_sm, Scan_en, Reset_n)
begin
                if Reset_n = ‘0’ then
                        iic_h_wr <= false;
                elsif iic_wr = ‘1’ and (Scan_en = ‘0’) then
                        iic_h_wr <= true;
                elsif i_clk_buf’event and i_clk_buf = ‘1’ then
```

```
                                        if br_sm /= RDY2XFER theniic_h_wr <= false;
                                        end if;
                        end if;
        end process IHW;
        -- latch of reset
        IFI: process (Reset_n, force_idle_n, i_clk_buf, br_sm, ii_sm, Scan_en)
        begin
                        if ((Reset_n = '0') or (force_idle_n = '0')) and (Scan_en = '0')
                                then            force_idle <= true;
                        elsif i_clk_buf'event and i_clk_buf = '1' then
                                        if (br_sm = RDY2XFER) and (ii_sm = IDLE) then
                                                force_idle <= false;
                                        end if;
                        end if;
        end process IFI;
        iic2hdata <= rd_byte;
        bit_da <= '1' when rd_ena = '1' else iic_wbyte(bit_cnt);
        not_idle <= '0' when ii_sm = IDLE else '1';
        xfer_done <= '0' when
                        ((rd_ena = '1') and (br_sm = BYTE_RDY) and not iic_h_rd) or
                        ((rd_ena = '0') and (br_sm = RDY2XFER) and not iic_h_wr)
                        else '1';
        bus_err <= '1' when ii_sm = B_CNFLK else '0';
        ack_rcv_n <= ack_rcv0;
end BEHAVIOR;

-- synopsys translate_off
configuration iic_con of iic is
        for behavior
                        for all: K1NQ use configuration work.K1NQ_config;
                        end for;
                        for all: FD1S use configuration work.FD1S_config;
                        end for;
        end for;
end iic_con;
-- synopsys translate_on
```

9 PCI Expansion Bus

The MACE ASIC contains a Peripheral Component Interconnect (PCI) host bridge for interfacing to PCI devices as defined by the PCI 2.1 specification. The interface supports both 32-bit and 64-bit, 33.33MHz bus masters and up to five external master devices can be controlled directly from the built in arbiter. The interface also contains read-ahead buffers for high performance DMA operation. A block diagram of the PCI interface is shown below:



9.1 PCI Host Bridge

The PCI host bridge is based on the rev 2.1 specification with the 64-bit optional extensions. The Intel specific special cycles and Dual address cycle are currently not supported. The optional cache coherency signals are not supported since *Moosehead* systems do not provide for hardware cache coherent I/O. The interface has been designed to support three 64-bit expansion slots and two on board SCSI devices at a minimum.

As a PCI master the host bridge can initiated single data phase 32-bit configuration, memory and I/O transactions. As a PCI slave the host bridge will respond to 32-bit or 64-bit, single or multiple data phase memory transactions targeted to the lower two gigabytes of PCI memory space. Memory transactions to the upper two gigabytes of PCI memory space and all PCI I/O transactions are ignore by the host and assumed to be targeted to PCI devices other than the host bridge.

TABLE 80. PCI Commands Supported by Host Bridge

C/BE[3:0]	Host Bridge as a PCI Master	Host Bridge as a PCI Slave
0000	N.A.(Interrupt Acknowledge)	N.A.(Interrupt Acknowledge)
0001	N.A. (Special Cycle)	N.A. (Special Cycle)
0010	I/O Read	N.A.(I/O Read)
0011	I/O Write	N.A.(I/O Write)
0100	N.A. (Reserved)	N.A. (Reserved)
0101	N.A. (Reserved)	N.A. (Reserved)
0110	Memory Read	Memory Read
0111	Memory Write	Memory Write
1000	N.A. (Reserved)	N.A. (Reserved)
1001	N.A. (Reserved)	N.A. (Reserved)
1010	Configuration Read (can initiate)	N.A.(Configuration Read)
1011	Configuration Write (can initiate)	N.A.(Configuration Write)
1100	N.A.(Memory Read Multiple)	Memory Read Multiple
1101	N.A. (Dual Address Cycle)	N.A. (Dual Address Cycle)
1110	N.A.(Memory Read Line)	Memory Read Line
1111	N.A.	Memory Write & Invalidate (same as Mem. Write)

9.2 PCI Command Usage

The above table lists the PCI commands that are supported by the host bridge. Commands in the table marked as N.A. (not applicable) have different meaning whether the host bridge is a master or slave. As a PCI master the host bridge is not capable of issuing these commands and as a PCI slave the host bridge will ignore transactions issued with these commands regardless of the accompanying address.

The preferred use of the read commands is:

Memory Read (MR) & **Memory Read Line (MRL)**: These two commands are considered to be the same and should be use to indicate that the master will not read across a 128 byte address boundary.

Memory Read Multiple (MRM): This command should be used indicates that the master will read across a 128 byte address boundary and that the bridge should read-ahead (or prefetch) the next 128 bytes if prefetching is enabled. If prefetching is disabled then no read-ahead is performed and the MRM command is treated the same as MR & MRL.

To get the best read performance, prefetching should be enabled (bit #11 of the PCI Control register set) and a PCI master should use the MRM command whenever it will read across a 128 byte address boundary. This will help reduce the high read latency to memory by the host bridge reading ahead. This means that if a PCI master needs to read 256 bytes starting on a 128 byte address boundary then the MRM command should be used to read the first 128 bytes in a single burst. The host bridge will disconnect the master when the end of the 128 byte cache line has been reached. The master should then read the last 128 bytes using the MRL command because it will not read across another 128 byte boundary.

If a master is not intelligent enough to be able to issue MRM and MRL as described above then the master must be programmed not to use the MRM command or prefetching will have to be disabled. The draw back to disabling prefetching is that it will disable prefetching for all PCI devices.

The cache line size used by PCI masters for the MRL and MRM commands is variable and must be programmed into each external masters' PCI configuration spaces. A device can be programmed with a cache line size smaller than the maximum given here. Note that these cache lines are virtual since the system does not support hardware cache coherency. In this case, the cache line size is really only an indication of the prefetch size. The bridge can only supports a virtual cache line size of 128 bytes.

Note: Any data read and buffered in the bridge is invalidate in hardware by memory write traffic to the same 128 byte address region. This ensures that PCI masters that are intelligent will never read stale data from system main memory if they do READ->WRITE->READ to the same memory address using MRL.

9.3 Address Spaces

Figure , "PCI Address Mapping/Translation," on page 160, shows how PCI memory and I/O spaces are mapped into the CPU address space and how the one gigabyte host memory is mapped into the PCI memory space in both native and byte swapped views. The first 32 megabytes of PCI memory and I/O space is also double mapped to be visible within the CPU's KSEG1 address region.

9.3.1 PCI Memory Space

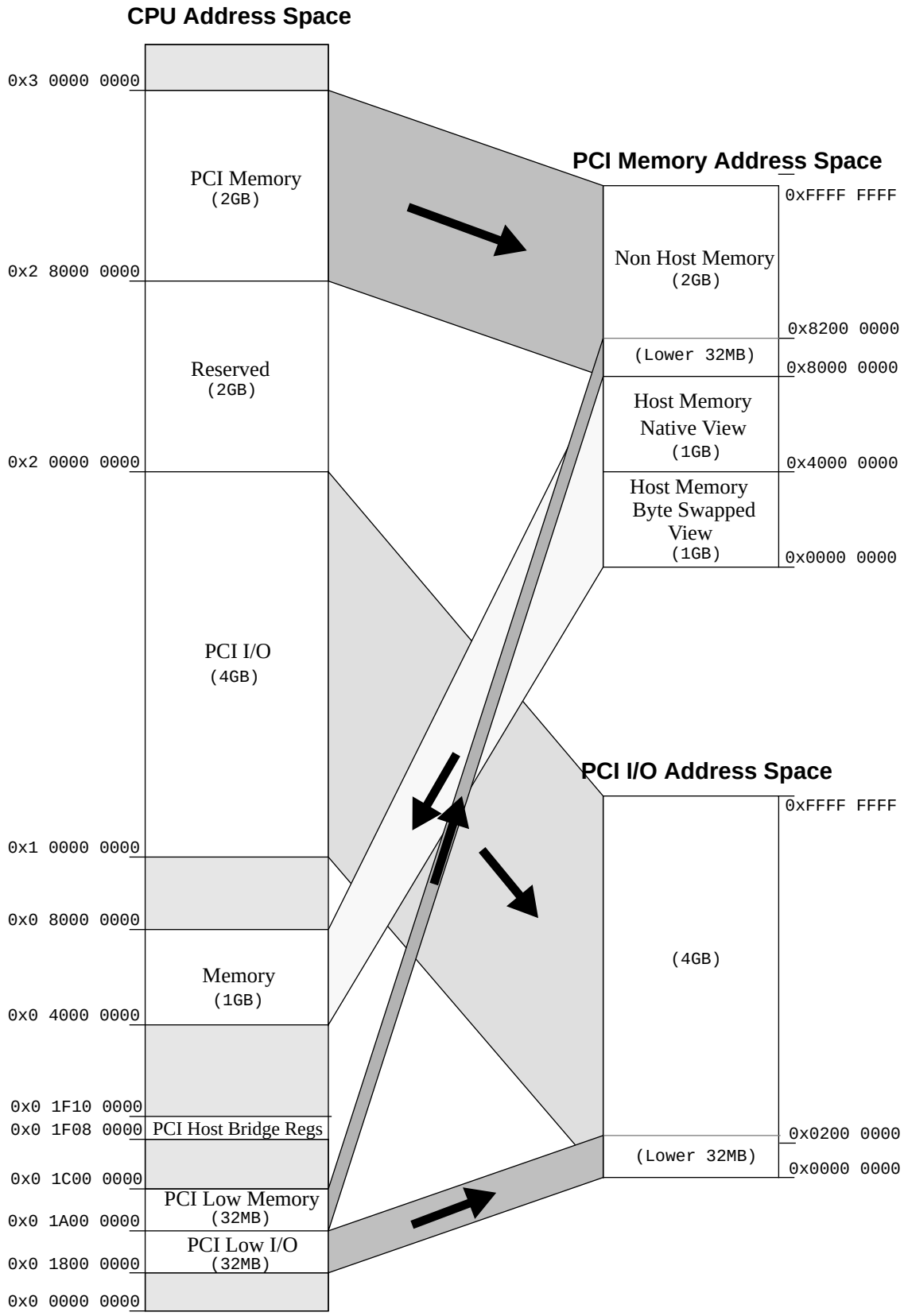
The PCI interface supports the full four gigabyte PCI memory space. The lower two gigabytes map to the host's main memory while the upper two gigabytes are left for PCI to PCI memory references. The host's main memory map is split into two halves, the upper gigabyte is a straight map or "native view" of the host's memory, while the lower gigabyte is a byte swapped view of the host's memory. Devices that need to byte swap DMA should be steered to the byte swapped view.

PCI devices must be able to be located above the lower 2 gigabyte area of PCI memory space. Devices that have bits 2 and 1 of their Memory Base Address registers encoded to indicate that they must be located below 1 megabyte can not be supported.

TABLE 81. Memory Base Address Register Bits 2/1 Encoding

Bits 2/1	Meaning
00	Locate anywhere in 32 bit address space
01	Locate below 1 Meg <NOT SUPPORTED>
10	Locate anywhere in 64 bit address space
11	Reserved

PCI Address Mapping/Translation



9.3.2 PCI I/O Space

The PCI interface supports the full four gigabyte I/O space defined in the PCI specification. All I/O space cycles are assumed to be targeted to devices on the PCI bus. Also, none of the *Moosehead* ASICs' registers are visible in the PCI I/O space or are accessible from the PCI bus.

9.3.3 PCI Configuration Space

The host bridge allows the processor to have access to configuration space in all devices on all PCI buses in the system through two 32-bit word registers, CONFIG_ADDRESS and CONFIG_DATA. These two registers are located in the PCI Host Bridge Internal Register space of the CPU address map: physical addresses 0x1F080CF8 and 0x1F080CFC, respectively.

The PCI bus that the host bridge is physically connected to has been hard wired as bus number zero and supports up to 255 subsidiary busses below it. Please see the PCI specification for more information on how configuration mechanism #1 operates.

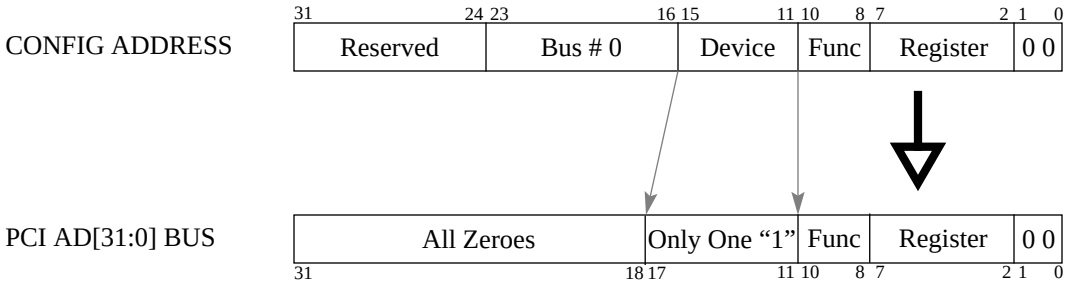


Figure: Bridge Translation for Type 0 Configuration Cycles

9.4 PCI Host Bridge Internal Registers

The PCI host bridge internal registers are located in the MACE internal register address space of the CPU address map. The 524KB region reserved for PCI starts at physical address 0x1F080000 and ends at 0x1F0FFFFF. Only six 32-bit registers are defined within this region and are listed in the following table. These registers are not aliased (e.g. repeated) to fill the entire 524KB region. Reads to addresses that are within this region and not listed in the table will return 0xFF in all byte lanes, and writes will be ignored with the data being discarded. If illegal transaction sizes are used to access any of the six registers, unexpected results may occur and the Illegal Transaction flag (bit # 27) will be set in the Error Flags Register.

Please note that SGI defines a *word* as a 32-bit quantity and a *double word* as a 64-bit quantity. The PCI specification defines a *word* as a 16-bit quantity and a *double word* as a 32-bit quantity. The SGI definition is used throughout this document.

TABLE 82. PCI Host Bridge Internal Registers

Physical Address	Bits	Reset	Type	Register Name	Legal Transaction Sizes
0x1F080000	31:0	0x0	R/O	Error Address Register	word accesses only
0x1F080004	31:0	0x0	R/W	Error Flags Register	word accesses only
0x1F080008	31:0	0x0	R/W	Control Register	word accesses only
0x1F08000C	31:0	0x0	R/W	(Write) Invalidate Read Buffers (Read) Revision Info	word accesses only
0x1F080CF8	31:0	0x0	W/O	CONFIG_ADDRESS	word accesses only
0x1F080CFC	31:0	0x0	R/W	CONFIG_DATA	byte, halfword, triplebyte & word accesses only

9.4.1 PCI Error Address Register

The PCI host bridge provides a register for reporting the PCI bus address when an error condition is detected during a PIO transaction. The content of this register is only valid if one of the four Error Address Type flags (bits #16 thru #19 of the PCI Error Flags Register) is set.

TABLE 83. PCI Error Address Register

Bits	Reset	Type	Description
31:0	0	R/O	PCI Address when Error Occurred

9.4.2 PCI Error Flags Register

The PCI Error Flags register reports the type of error(s) that have been detected. Bits #23 thru #31, and bit #4, can be individually cleared by writing a logic 0, or left as is by writing a logic 1 to each bit. Clearing bits #28 thru #31 causes the corresponding Error Address Type flags (bits #19 thru #16) to be cleared. For example, if the first error condition detected is a master abort during a configuration cycle, bits #31, #19 and #20 will get set. The PCI address of the configuration cycle will be captured in the PCI Error Address register. Clearing bit #31 will also cause bit #19 to be cleared to zero. Bits #20, #21 and the content of the Error Address register will remain unchanged and will now be considered invalid until the next error condition is detected.

The host can initiate an interrupt test by setting bit #25 to a logic 1. This will cause a single interrupt packet to be sent to CRIME indicating that an internal interrupt has been generated. In rev 0 of the host bridge the hardware automatically resets this bit back to a logic 0 one clock after being set. Thus, bit #25 will always be read as a logic 0. In rev 1, the bit is not automatically cleared by the hardware. Software needs to clear this bit after detecting the interrupt by writing a logic 0 to it. The interrupt test is intended for testing and debugging purposes only.

TABLE 84. PCI Error Flags Register

Bits	Reset	Type	Description
0	0	R/O	66 MHz Capable
1	1	R/O	Fast Back-to-Back Capable
3:2	1	R/O	DEVSEL timing 00 - fast 01 - medium 10 - slow
4	0	R/W	Signaled Target Abort
15:5	0	R/O	<Reserved>
16	0	R/O	Error Address for Retry Error
17	0	R/O	Error Address for Data Parity Error Detected
18	0	R/O	Error Address for Target Abort
19	0	R/O	Error Address for Master Abort
20	0	R/O	Error Address to Config Space
21	0	R/O	Error Address to Memory Space
22	0	R/O	Signaled System Error
23	0	R/W	Read Buffer Overrun
24	0	R/W	Detected Parity Error
25	0	R/W	Interrupt Test
26	0	R/W	Detected System Error
27	0	R/W	Illegal Host Transaction
28	0	R/W	Retry Error
29	0	R/W	Data Parity Error Detected
30	0	R/W	Received Target Abort
31	0	R/W	Received Master Abort

9.4.3 PCI Host Bridge Control Register

The PCI host bridge Control register controls the enabling and disabling of bridge functionality.

TABLE 85. PCI Host Bridge Control Register

Bits	Reset	Type	Description
0	0	R/W	PCI Interrupt #0 Enable (SCSI Controller 0) 0 - Disable Interrupt 1 - Enable Interrupt
1	0	R/W	PCI Interrupt #1 Enable (SCSI Controller 1)
2	0	R/W	PCI Interrupt #2 Enable (Slot zero INTA#)
3	0	R/W	PCI Interrupt #3 Enable (Slot one INTA#)
4	0	R/W	PCI Interrupt #4 Enable (Slot two INTA#)
5	0	R/W	PCI Interrupt #5 Enable (Slot zero INTB#, Slot one INTC#, Slot two INTD#)
6	0	R/W	PCI Interrupt #6 Enable (Slot zero INTC#, Slot one INTD#, Slot two INTB#)
7	0	R/W	PCI Interrupt #7 Enable (Slot zero INTD#, Slot one INTB#, Slot two INTC#)
8	0	R/W	SERR_N Enable 0 - Disable 1 - Enable
9	0	R/O R/W	<rev 0> Reserved <rev 1> PCI Arbiter Control - Arbitration Level for REQ_N[6] 0 - Round Robin 1 - 1st Priority of Fixed Level
10	0	R/W	Parity Error Response 0 - Disable 1 - Enable
11	0	R/W	Memory Read Multiple Read Ahead 0 - Disable 1 - Enable
12	0	R/W	PCI Arbiter Control - Arbitration Level for REQ_N[3] 0 - Round Robin 1 - 4th Priority of Fixed Level
13	0	R/W	PCI Arbiter Control - Arbitration Level for REQ_N[4] 0 - Round Robin 1 - 3rd Priority of Fixed Level
14	0	R/W	PCI Arbiter Control - Arbitration Level for REQ_N[5] 0 - Round Robin 1 - 2nd Priority of Fixed Level
15	0	R/W	PCI Arbiter Control - Enable Parking on Last in Use 0 - Park on Host 1 - Park on Last in Use
16	0	R/W	Invalidate Prefetch Buffers on PCI Interrupt #0 0 - Disable 1 - Enable
17	0	R/W	Invalidate all Read Buffers on PCI Interrupt #1
18	0	R/W	Invalidate all Read Buffers on PCI Interrupt #2
19	0	R/W	Invalidate all Read Buffers on PCI Interrupt #3
20	0	R/W	Invalidate all Read Buffers on PCI Interrupt #4

TABLE 85. PCI Host Bridge Control Register

Bits	Reset	Type	Description
21	0	R/W	Invalidate all Read Buffers on PCI Interrupt #5
22	0	R/W	Invalidate all Read Buffers on PCI Interrupt #6
23	0	R/W	Invalidate all Read Buffers on PCI Interrupt #7
24	0	R/W	Overrun Condition Interrupt Enable 0 - Disable 1 - Enable
25	0	R/W	Detected Parity Error Interrupt Enable
26	0	R/W	System Error Interrupt Enable
27	0	R/W	Illegal Transaction Interrupt Enable
28	0	R/W	Retry Error Interrupt Enable
29	0	R/W	Data Parity Error Detected Interrupt Enable
30	0	R/W	Target Abort Received Interrupt Enable
31	0	R/W	Master Abort Received Interrupt Enable

9.4.4 PCI Read Buffer Flush/Revision Info Register

This 32-bit register provides two different functions depending on whether its being written or read. A write to this register causes the read buffers to be flushed (or invalidated) regardless of the value written. The write data is discarded. A read to this register will return the revision number of the host bridge.The first version of MACE (rev 1.0) has a host bridge revision number of 0 and MACE 2.0 has a host bridge revision number of 1.

TABLE 86. Buffer Flush/Revision Info Register

Bits	Reset	Type	Description
31:0	Rev. Info	R/W	(write) Invalidate Read Buffers (read) Revision Info

9.5 External Master Arbitration

There are two arbitration levels for external masters. The default is for all devices to be included in the lower priority round robin ring. Bits #9, and #12 thru #14 of the PCI Control register allow devices #3 thru #6 to be removed from the round robin ring and included in the higher priority fixed arbitration level. Each device's priority level within the fixed arbitration scheme is determined by which PCI slot the device is plugged into. Slot #3 has highest priority and slot #1 has lowest priority.

9.6 External Interrupts

The PCI interface provides eight external interrupt inputs. Five of these interrupts are dedicated to the INTA signals from devices #1 thru #5. The other three signals are connected to the PCI expansion slots INTB through INTD signals in a spiral pattern. The spiral pattern distributes the extra three interrupt inputs such that the possibility of overlapped use is minimized. The following table shows the connections:

TABLE 87. External interrupt connections

Interrupt Input	Description of external destination
0	SCSI Controller INTA#
1	SCSI Controller INTB#
2	PCI Slot 0, INTA#
3	PCI Slot 1, INTA#
4	PCI Slot 2, INTA#
5	PCI Slot 0, INTB# - PCI Slot 1, INTC# - PCI Slot 2, INTD#
6	PCI Slot 0, INTC# - PCI Slot 1, INTD# - PCI Slot 2, INTB#
7	PCI Slot 0, INTD# - PCI Slot 1, INTB# - PCI Slot 2, INTC#

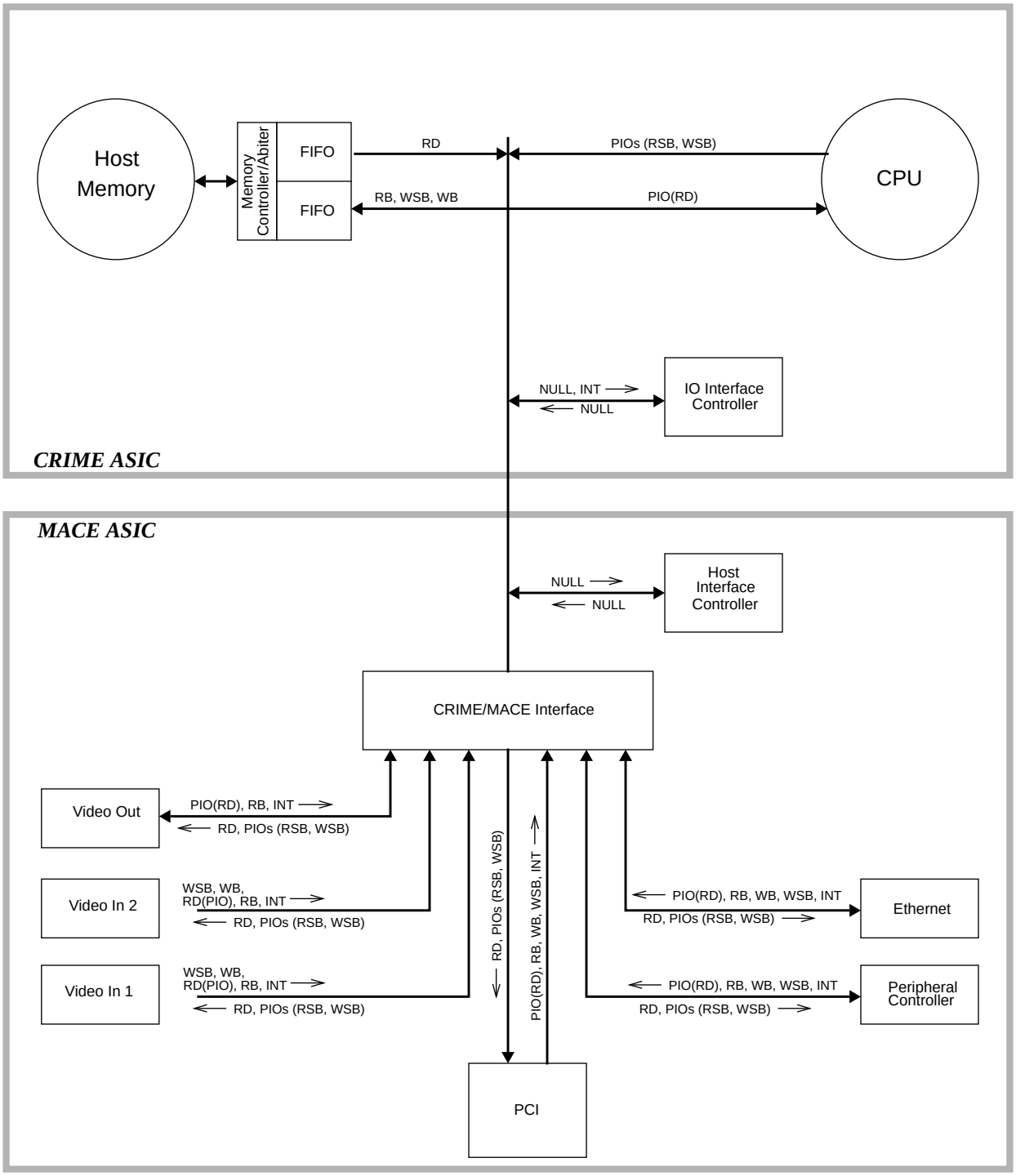
9.7 Bug List for Host Bridge Rev 0

- There are two mechanisms provided for invalidating the read buffers. Invalidation can be triggered by an interrupt or by software writing the Read Buffer Flush register. This functionality should be considered broken and not used. If either of these two mechanisms are used while a memory read is in progress on the PCI bus the host bridge might return bad data if several conditions happen to line up just right.
- The host bridge provides a 4-byte aligned address instead of a full byte address for PCI I/O transactions. This means that halfword and byte accesses to PCI I/O space may not work if the target device expects a full byte address.
- PIOs to the upper 2GB of PCI I/O space get aliased to the lower 2GB of PCI I/O space.
- When either a Memory Read or Memory Read Line command directly follows a Memory Read Multiple command, the host bridge treats the Memory Read/Memory Read Line as a Memory Read Multiple. Causing the next 128 bytes to be prefetched if prefetching is enabled.
- There is a bug in the PCI arbiter that can cause two PCI devices to be granted the bus at the same time if more than one PCI device is programmed to be in the fixed priority arbitration level. This will not be a problem for Moosehead but could be a problem for Road Runner.
- There is a bug that causes bad data to be written to host memory when a 64-bit PCI master writes to the *native* view of host memory with one or more disabled byte lanes. This bug does not affect 32-bit PCI devices.

NOTE: Petty Crime does not alias the *upper* 16MB of PCI I/O space (0x19000000 - 0x19FFFFFF) and PCI low memory (0x1B000000 - 0x1BFFFFFF) as shown in Figure , “PCI Address Mapping/Translation,” on page 160. Accesses to these two regions will cause the system to hang. This problem will be resolved in the final version of CRIME. Please note that the lower 16MB of PCI I/O space (0x18000000 - 0x18FFFFFF) and PCI low memory (0x1A000000 - 0x1AFFFFFF) are aliased correctly.

10 CRIME Link

The *Moosehead* system I/O asic contains a high speed link to the CRIME asic. The interface is a high speed queued message passing interface that uses 64-bit wide data and control words. The CRIME Link interface controls all communication between MACE and CRIME. A high level message path picture is shown below:



10.1 General Description

The System Interface consists of the host bus interface, bus selector and FIFOs. Together they provide the communication path for DMA and PIO packets to be sent between CRIME and subsytemswithin the MACE ASIC. Packets received from the host are routed directly to the appropriate FIFOs. Packets sent from the subsystems are qued in their individual FIFOs and sent to the host based on a fixed arbitration scheme. The figure at the start of this chapter shows the basic block diagram of the System Interface and the data paths to/from the subsystems.

Key characteriests of the System Interface:

Half-duplex, Synchronous Host Interface

DMA Transaction Flow Control

FIFO and Host Bus Error Notification

10.1.1 CRIME Interface

The CRIME Interface implements a half-duplex, 32-bit, double-speed, synchronous bus. The data bus runs at twice the internal clock rate to achieve the effective 64-bit per cycle transfer rate. Bus mastership is negotiated via the TOKEN_IN and TOKEN_OUT signals. Upon reset and power up, bus mastership defaults to CRIME. MACE negotiates for bus ownership by asserting its TOKEN_OUT signal and waiting for TOKEN_IN to be . To flow control write transactions there are write acknowledge signals back to each ASIC that indicate when an outstanding write transaction has completed.

10.1.1.1 Data Format and Endianness

All data passed through the CMI is expected to be byte ordered in big-endian format. No support is available for switching endianness within the CMI. The 64-bit doublewords passed between CRIME and MACE will be sent across the 32-bit bus as two 32-bit word tranfers.The following figure illustrates the byte/word ordering.

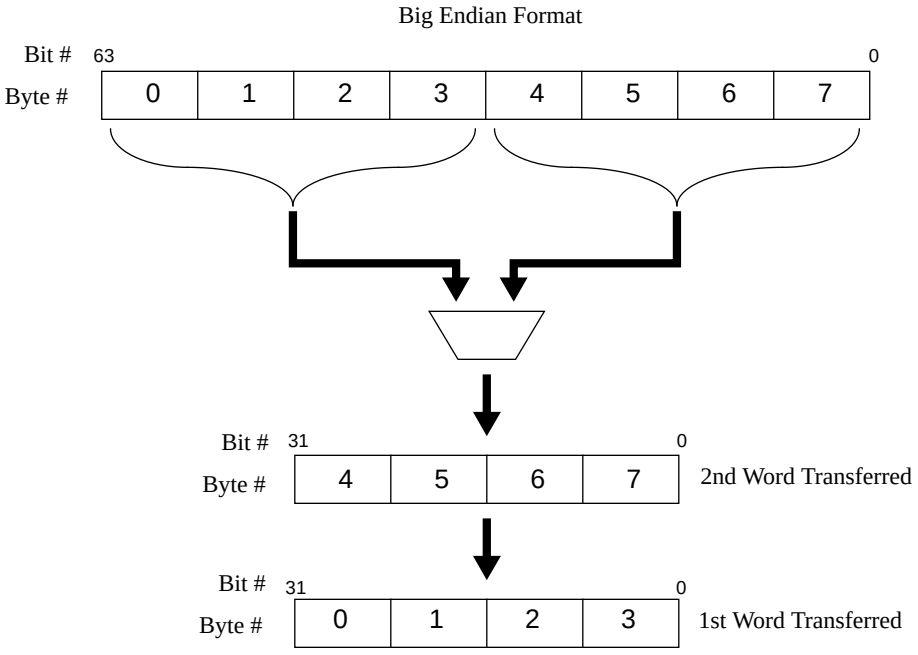


FIGURE 12. Byte Ordering of Bus Transfers

10.1.2 Transaction Ordering

Transactions from CRIME to MACE are loaded into the subsystem FIFOs in the order issued. Similarly, transactions issued from a subsystem are passed onto CRIME in the order issued. However, the ordering of transactions from different subsystems to CRIME is dependent upon the arbitration scheme. There is no ordering between transactions initiated by CRIME and transactions initiated by MACE's subsystems. These two types of transaction are flowing in opposite and independent directions.

10.1.3 Transaction Flow Control

Transaction flow control is maintained by the initiator assuring that the number of outstanding transactions it generates doesn't exceed the maximum allowed. This requires the host to keep track of the number of outstanding PIO transactions, and MACE to keep track of the number of outstanding Memory transactions.

The MACE bus arbiter keeps track of outstanding transactions to assure that the maximum number of memory transactions that CRIME's FIFO can handle is not exceeded. Within MACE, each subsystem's FIFO controller will be responsible for limiting the number of read request packets issued so that its return FIFO won't overflow.

DMA read transactions will be acknowledged after the Read Data Response packet has returned and loaded into the appropriate subsystem FIFO. This will cause the MACE bus arbiter to decrement the outstanding transaction counter, thus allowing another transaction to be sent to CRIME.

MACE expects CRIME to assert its Memory Write Acknowledge (MWA) signal for one clock period once a memory write transacion has completed.

10.1.3.1 PIO Transactions

PIO packets from CRIME to MACE will be routed to each subsystem within MACE based on the decoding of the MSBs of the header's ADR field. Each subsystem will be capable of storing 2 PIO write packets (total of four 64-bit words), or one write and one read. The following is a list of possible outstanding PIO combinations:

a) 1 or 2 PIO writes

b) 1 PIO write followed by a PIO read

c) 1 PIO read only

CRIME will assure that no more than 2 outstanding PIO writes will be issued and no more than 1 outstanding read. If there is an outstand PIO read then no other PIO transaction (read or write) will be issued until the current PIO read has completed. This will be determined by CRIME identifying the Read Data Response packet returned by MACE. The TAG and ADR fields of the Read Block packet will be returned in the corresponding Read Data Response packet's header.

Each completed PIO write will be acknowledged by PWA (PIO Write Acknowledge signal from MACE to CRIME) being asserted for 1 clock period of the 66Mhz system clock. MACE will NOT guarantee that 2 outstanding PIO writes to seperate subsystems will complete in the order issued. However 2 outstanding PIO transactions to the same subsystem will be completed in the order issued.

10.1.3.2 PIO Write Acknowledge

Each subsystem will supply a PIO write acknowledge signal to the CMI. Once a PIO write has completed, the subsystem will assert its PIO write acknowledge signal for one period of the system clock. The individual subsystems will be responsible for synchronizing their write acknowledge signal with the system clock. A simple state machine

within the CMI will monitor the write acknowledge signals and assert the PWA signal to CRIME for one system clock period for each write acknowledge signal that is asserted. If two simultaneous write acknowledges from different subsystems occure, PWA will be asserted for two consecutive system clocks.

10.1.4 Interrupt Packet Flow Control

The CMI will assure that only one outstanding interrupt packet will be sent to CRIME at any one time. Interrupt packets will be treated as out of band messages, in that they bypass pending memory writes in the CRIME asic. This means that system software provide the synchronization to ensure that pending memory writes are flushed if needed to signal the completion of a DMA transaction. This can usually be done by doing a single PIO read.

10.1.4.1 PCI Clock

The PCI clock will be generated externally by an oscillator located on the motherboard. The clock frequency will be 33.33MHz (30.00ns period). This clock will NOT be synchronized to the 66MHz system clock. Any signals passing to/from the 33MHz domain from/to the 66MHz domain will be re-synchronized to the appropriate clock.

10.1.5 Tag Codes

The CRIME messages contain a single six bit field that identifies the functional block involved in the message that should receive the message and it's type. The following table shows the current tag field decode:

TABLE 88. Tag Codes

Tag[5:0]	Subsystem	Tag[5:0]	Subsystem
0x00	Video In 1 - PIO	0x20	PCI - Prefetch Buffer #0
0x01	Video In 1 - Memory Read	0x21	PCI - Prefetch Buffer #1
0x02	Video In 2 - PIO	0x22	PCI - Prefetch Buffer #2
0x03	Video In 2 - Memory Read	0x23	PCI - Prefetch Buffer #3
0x04	Video Out - PIO	0x24	PCI - Prefetch Buffer #4
0x05	Video Out - Memory Read	0x25	PCI - Prefetch Buffer #5
0x06	Video Out - Memory Write	0x26	PCI - Prefetch Buffer #6
0x07	Video Out - (not used)	0x27	PCI - Prefetch Buffer #7
0x08	Ethernet - PIO from CRIME	0x28	PCI - Prefetch Buffer #8
0x09	Ethernet - Transmit DMA Cat Buf	0x29	PCI - Prefetch Buffer #9
0x0A	Ethernet - Status Vector Writes	0x2A	PCI - Prefetch Buffer #10
0x0B	Ethernet - PIO Read Response	0x2B	PCI - Prefetch Buffer #11
0x0C	Ethernet - Interrupt Update	0x2C	PCI - Prefetch Buffer #12
0x0D	Ethernet - Receive DMA Data Blks	0x2D	PCI - Prefetch Buffer #13
0x0E	Ethernet - Transmit DMA Ring Even	0x2E	PCI - Prefetch Buffer #14
0x0F	Ethernet - Transmit DMA Ring Odd	0x2F	PCI - Prefetch Buffer #15
0x10	ISA - PIO to Devices	0x30	PCI - (not used)
0x11	ISA - PIO to Flash-ROM	0x31	PCI - (not used)
0x12	ISA - Audio output DMA channel #1	0x32	PCI - (not used)
0x13	ISA - Audio output DMA channel #2	0x33	PCI - (not used)
0x14	ISA - Serial output DMA channel #1	0x34	PCI - (not used)
0x15	ISA - Serial output DMA channel #2	0x35	PCI - (not used)
0x16	ISA - Parallel DMA channel	0x36	PCI - (not used)
0x17	ISA - Audio input DMA channel	0x37	PCI - (not used)
0x18	ISA - Serial input DMA channel #1	0x38	PCI - (not used)
0x19	ISA - Serial input DMA channel #2	0x39	PCI - (not used)
0x1A	FUTURE - PIO	0x3A	PCI - (not used)
0x1B	FUTURE - <reserved>	0x3B	PCI - (not used)
0x1C	PCI - PIO to config & low io/mem	0x3C	PCI - (not used)
0x1D	PCI - PIO to PCI I/O	0x3D	PCI - (not used)
0x1E	PCI - PIO to PCI Memory	0x3E	PCI - (not used)
0x1F	PCI - (not used)	0x3F	PCI - (not used)

10.1.6 Transaction Types

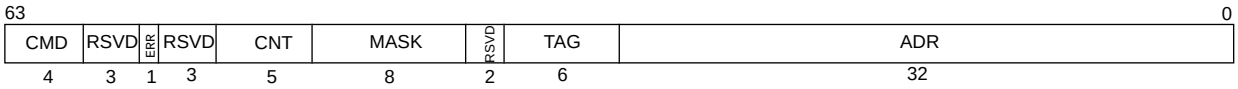
The following transaction types are supported:

TABLE 89. Supported Transaction Types

Type	CMD bits	# of Dwords	Description
NULL	0000	1	Null Transaction
INT	0001	1	Interrupt Update
WB	0010	2-33	Write Block
WSB	0011	2	Write Sub-Block
RB	0100	1	Read Block Request
RSB	0101	1	Read Sub-Block Request
RD	0110	2-33	Read Data Response

Null Packet

The Null (NULL) packet is one doubleword transfer of the following format:



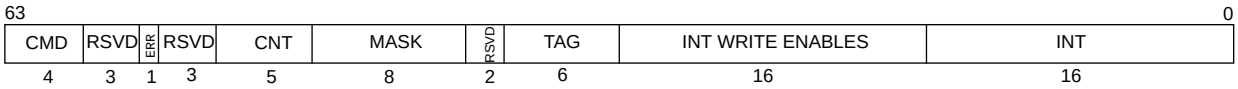
The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Null command code (0000)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	(Not Applicable) Doubleword Count
47..40	8	MASK	(Not Applicable) Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	(Not Applicable) Tag
31..0	32	ADR	(Not Applicable) Byte Address

NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes. Therefore, all 64 bits must be set to zero (deasserted).

Interrupt Packet

The Interrupt (INT) packet is a doubleword transfer of the following format:



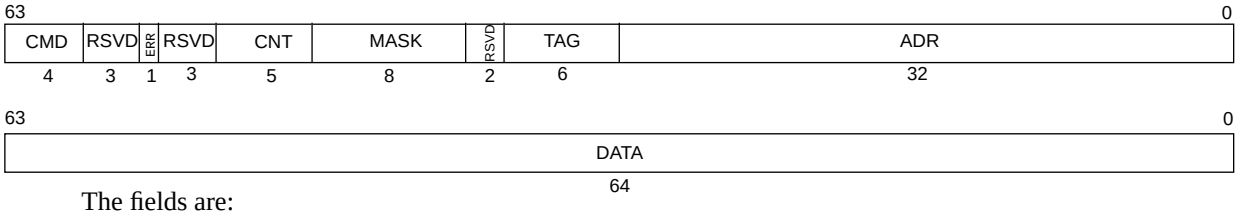
The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Interrupt command code (0001)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	(Not Applicable) Doubleword Count
47..40	8	MASK	(Not Applicable) Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..16	16	INT MASK	Interrupt Bit Write Enables
15..0	16	INT	Interrupt Bits

NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

Write Block Packet

The Write Block (WB) packet is a doubleword header followed by 1 to 32 doubleword data transfers of the following format:



The fields are:

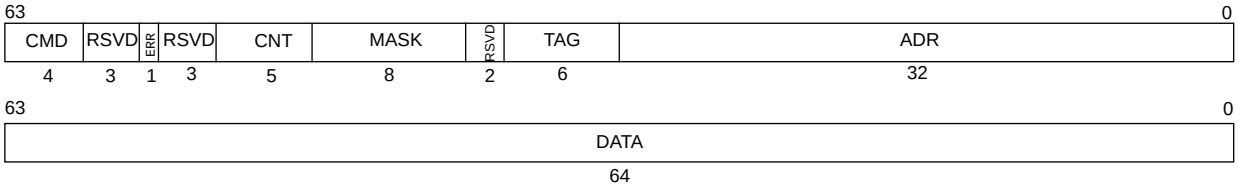
Bits	Width	Name	Description
63..60	4	CMD	Write Block command code (0010)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	Doubleword Count (00000 to 11111)
47..40	8	MASK	(Not Applicable) Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..0	32	ADR	Byte Address

NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

2. CNT field specifies the number *minus one* of data doublewords contained in the packet. (e.g. If CNT equals zero, then there is one data doubleword that follows the header.)

Write Sub-Block Packet

The Write Sub-Block (WSB) packet is a doubleword header followed by one doubleword data transfer of the following format:



The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Write Sub-Block command code (0011)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	Doubleword Count (00000)
47..40	8	MASK	Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..0	32	ADR	Byte Address

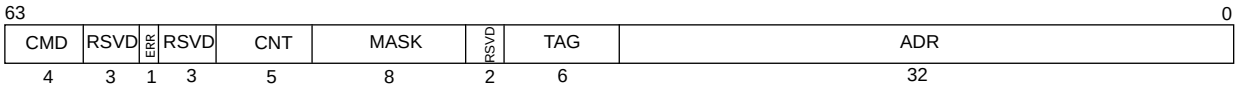
NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

2. CNT field specifies the number *minus one* of data doublewords contained in the packet. Only *one* data doubleword is allowed in a WSB packet, so the CNT field must be set to zero.

3. The MASK field specifies which bytes to be masked in the data doubleword.
   e.g. To mask byte #7 (big endian ordering - bits 7 down to 0) then the least significant bit of the MASK field should be asserted (bit #40 of the header).

Read Block Request Packet

The Read Block Request (RB) packet is a doubleword transfer of the following format:



The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Read Block Request command code (0100)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	Doubleword Count (00000 to 11111)
47..40	8	MASK	(Not Applicable) Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..0	32	ADR	Byte Address

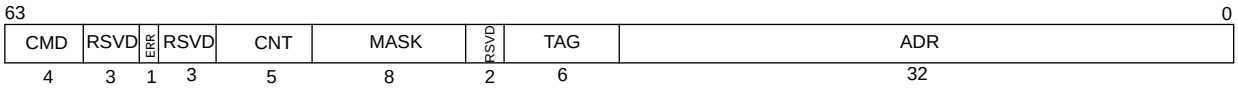
NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

2. CNT field specifies the number *minus one* of data doublewords requested.
(e.g. If CNT equals zero, then only one data doubleword is being requested.)

The RB packet is routed to the destination and at some later time a read data response packet (RD) will be sent back to the source that issued the RB packet.

Read Sub-Block Request Packet

The Read Sub-Block Request (RSB) packet is a doubleword transfer of the following format:



The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Read Block Request command code (0101)
59..57	3	RSVD	Reserved
56	1	ERR	(Not Applicable) Error
55..53	3	RSVD	Reserved
52..48	5	CNT	Doubleword Count (00000)
47..40	8	MASK	Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..0	32	ADR	Byte Address

NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

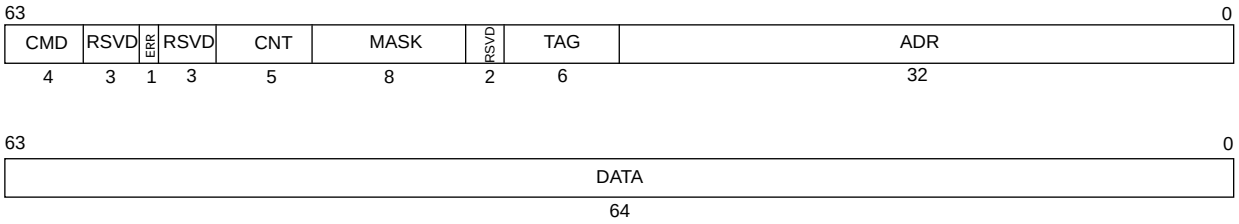
2. CNT field specifies the number *minus one* of data doublewords contained in the packet. Only *one* data doubleword is allowed to be requested in a RSB packet, so the CNT field must be set to zero.

3. The MASK field specifies which bytes to be masked.
e.g. To mask byte #7 (big endian ordering - bits 7 down to 0) then the least significant bit of the MASK field should be asserted (bit #40 of the header).

The RSB packet is routed to the destination and at some later time a read data response packet (RD) will be sent back to the source that issued the RSB packet.

Read Data Response Packet

The Read Data Response packet is a doubleword header followed by 1 to 32 doubleword data transfers of the following format:



The fields are:

Bits	Width	Name	Description
63..60	4	CMD	Read Data Response command code (0110)
59..57	3	RSVD	Reserved
56	1	ERR	Error
55..53	3	RSVD	Reserved
52..48	5	CNT	Doubleword Count (00000 to 11111)
47..40	8	MASK	(Not Applicable) Byte Mask
39..38	2	RSVD	Reserved
37..32	6	SRC	Tag
31..0	32	ADR	Byte Address

NOTE: 1. All bits of fields listed as *not applicable* or *reserved* must be set to zeroes.

2. CNT field specifies the number *minus one* of data doublewords contained in the packet. (e.g. If CNT equals zero, then there is one data doubleword that follows the header.)

The RD packet is routed back to the source that issued the corresponding RB or RSB packet. The ERR field indicates the status of the read data as follows:

ERR	Encoding	Description
OK	0	Data valid
INVLD	1	Read request invalid

The INVLD status may occur because of a reference to an invalid address.

11 Interrupt Map

The MACE chip contains a number of level sensitive interrupt signals that are mapped from all of the I/O asics internal functional blocks onto the sixteen CRIME interrupt bit slots allocated to the I/O asic. Each group of interrupt sources is effectively OR'd together, and the resulting logic '0' or '1' value is transmitted to the CRIME asic and stored in its master interrupt register. Whenever the logic value of a group changes from '1' to '0' or from '0' to '1', the new value is automatically sent to the CRIME asic.

The interrupt unit in the I/O asic needs to take into account the possible race condition between DMA write operations that have been posted to CRIME but not yet completed, and the interrupt signaling the processor that the DMA write operation has been completed. To protect against this race condition, all interrupt messages to the CRIME asic need to be strongly ordered with any DMA write traffic from the same source. No out-of-band signaling is allowed.

11.1 Mapping

The following table shows the assignment of all the interrupts in the I/O asic. The left hand column shows to which CRIME master interrupt bit slot each block of interrupts maps. The right most column gives a description of each interrupt source. The entire 16 entry table is shown below:

TABLE 90. Interrupt Assignments

CRIME Bit Slot	Interrupt Source
15	PCI Interrupt Input #7 (Slot zero INTD#, Slot one INTB#, Slot two INTC#)
14	PCI Interrupt Input #6 (Slot zero INTC#, Slot one INTD#, Slot two INTB#)
13	PCI Interrupt Input #5 (Slot zero INTB#, Slot one INTC#, Slot two INTD#)
12	PCI Interrupt Input #4 (Slot two INTA#)
11	PCI Interrupt Input #3 (Slot one INTA#)
10	PCI Interrupt Input #2 (Slot zero INTA#)
9	PCI Interrupt Input #1 (SCSI controller 1)
8	PCI Interrupt Input #0 (SCSI controller 0)
7	PCI Error Conditions
6	Peripheral Controller, Audio Interrupts
5	Peripheral Controller, Misc keyboard/mouse/timer Interrupts
4	Peripheral Controller, Serial/Parallel Interrupts
3	Fast-Ethernet Interface
2	Video Output Channel
1	Video Input Channel #2
0	Video Input Channel #1

12 Address Maps

The *Moosehead* system I/O asic internal register address map is shown below. The I/O asic has a total of 4 Megabytes of address space for internal registers which has been broken down into eight blocks for the internal subsystems. Note that the size of each block is so large that the subsystems do not use all of the space available. Those registers that exist are aliased, i.e. repeated, to fill the entire block. The main address map table is shown below:

TABLE 91. MACE Primary Address Map

Physical Address	Major Offset	PIO A[21:19]	CRIME PIO Tag	Block Description
0x1F000000	0x000000	000	0x1A	Future Subsystem Interface
	0x080000	001	0x1C	PCI Interface
	0x100000	010	0x00	Video Input #1
	0x180000	011	0x02	Video Input #2
	0x200000	100	0x04	Video Output
	0x280000	101	0x08	Fast-Ethernet Interface
	0x300000	110	0x10	Peripheral controller
	0x380000	111	0x10	ISA bus external I/O space

12.1 Peripheral Controller

The peripheral controller within the MACE I/O asic contains several subsystems. The table below shows the address map within that the peripheral controller:

TABLE 92. Peripheral Controller Sub-Address Map

Physical Address	Major Offset	PIO A[18:16]	Block Description
0x1F300000	0x00000	000	Audio Interface Registers
	0x10000	001	ISA DMA internal Registers
	0x20000	010	Keyboard & Mouse Registers
	0x30000	011	I^2C Interface Registers
	0x40000	100	Count/Compare (UST/MSC) Timer Registers
	0x50000	101	<alias of Compare register #1>
	0x60000	110	<alias of Compare register #2>
	0x70000	111	<alias of Compare register #3>

12.1.1 Keyboard & Mouse

The keyboard and mouse in the MACE I/O asic share an address slice within the peripheral controller. The table below shows the address decode for the keyboard and mouse section:

TABLE 93. Keyboard & Mouse Sub-Address Map

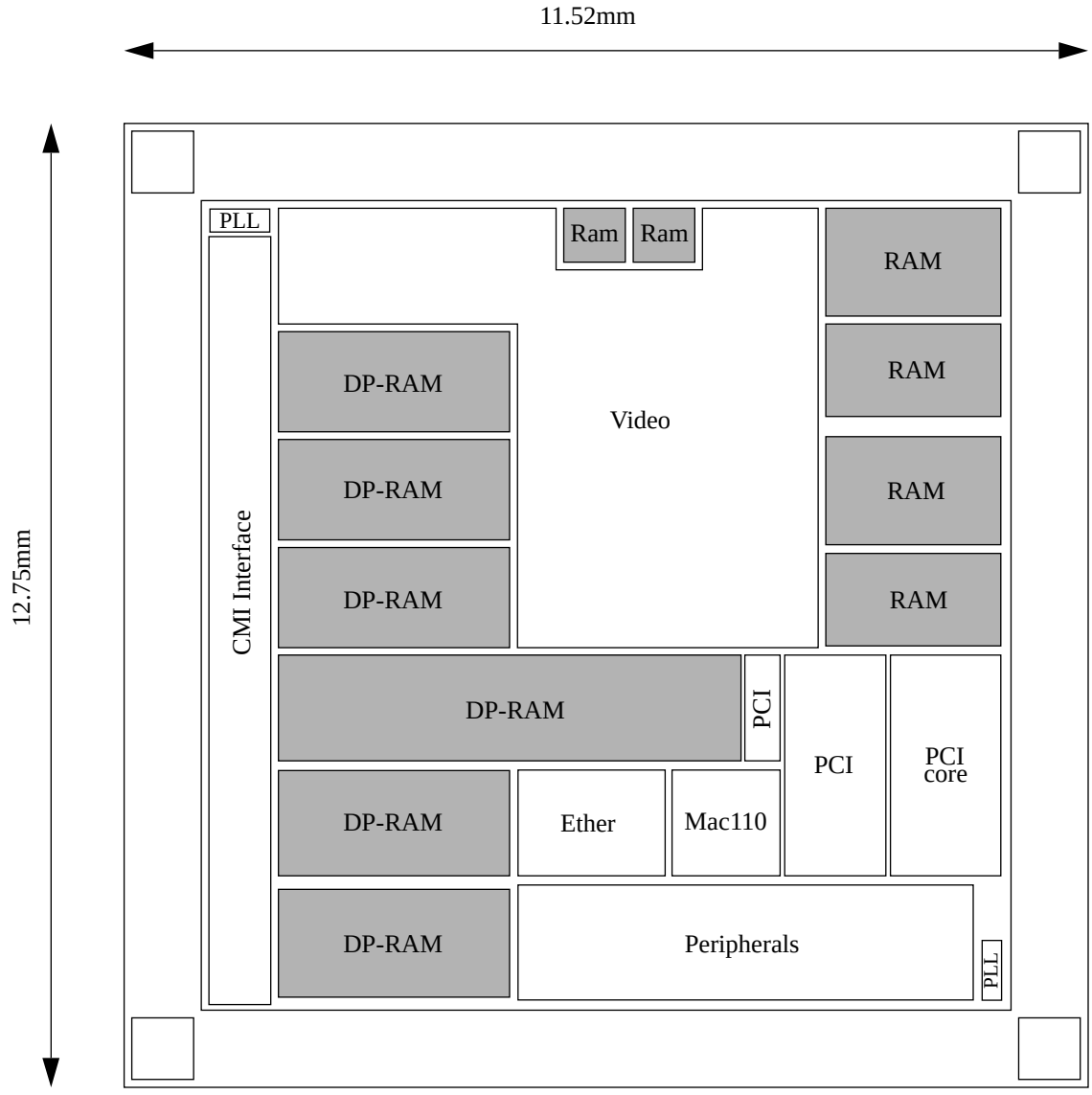
Physical Address	Major Offset	PIO A[5]	Block Description
0x1F320000	0x00	0	Keyboard registers
	0x20	1	Mouse registers

13 Physical

This chapter contains all the physical chip information on the *Moosehead* system I/O asic called MACE.

Floorplan:

LSI 500K
2LM
440 T-BGA
$78-$84



13.1 Pin List

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
10	C7	MIITxEn	bottom	out	5v	bt4rf	1	Ethernet
15	B5	MIITxD[3]	bottom	out	5v	bt4rf	1	Ethernet
20	D7	MIITxD[2]	bottom	out	5v	bt4rf	1	Ethernet
25	E7	MIITxD[1]	bottom	out	5v	bt4rf	1	Ethernet
30	C6	MIITxD[0]	bottom	out	5v	bt4rf	1	Ethernet
35	A4	MIITxEr	bottom	out	5v	bt4rf	1	Ethernet
45		VDD	bottom	pwr		VDD	0	Ethernet
40	A3	MIITxClk	bottom	in	5v	ibuff	1	Ethernet
105	D4	MIICrs	bottom	in	5v	ibuff	1	Ethernet
110	B3	MIICol	bottom	in	5v	ibuff	1	Ethernet
75	C5	MIIRxDV	bottom	in	5v	ibuff	1	Ethernet
85	D5	MIIRxD[3]	bottom	in	5v	ibuff	1	Ethernet
90	A2	MIIRxD[2]	bottom	in	5v	ibuff	1	Ethernet
95	C4	MIIRxD[1]	bottom	in	5v	ibuff	1	Ethernet
100	E5	MIIRxD[0]	bottom	in	5v	ibuff	1	Ethernet
5		VSS	bottom	pwr		VSS	0	Ethernet
80	B4	MIIRxEr	bottom	in	5v	ibuff	1	Ethernet
70	E6	MIIRxClk	bottom	in	5v	ibuff	1	Ethernet
115	B2	MIID_IO	bottom	bidi	5v	bd4crpf	1	Ethernet
120	E4	MIIDclk	bottom	out	5v	bt4rf	1	Ethernet
125		AudSclk	bottom	in	5v	ibuff	1	Audio
150		VSS	bottom	pwr		VSS	0	ISA
260		VDD	bottom	pwr		VDD	0	ISA
305		VSS	bottom	pwr		VSS	0	ISA
360		CoreVDD	bottom	pwr		VDD2	0	CorePwr
365		CoreVSS	bottom	pwr		VSS2	0	CorePwr
475		VDD	bottom	pwr		VDD	0	ISA
130		AudSdin	bottom	in	5v	ibuff	1	Audio
135		AudSdout	bottom	out	5v	bt4rf	1	Audio
140		AudSsync	bottom	in	5v	ibuff	1	Audio
145		AudRst	bottom	out	5v	bt4rf	1	Audio
265		SD[7]	bottom	bidi	5v	bd4crpf	1	ISA
270		SD[6]	bottom	bidi	5v	bd4crpf	1	ISA
275		SD[5]	bottom	bidi	5v	bd4crpf	1	ISA
280		SD[4]	bottom	bidi	5v	bd4crpf	1	ISA
285		SD[3]	bottom	bidi	5v	bd4crpf	1	ISA
290		SD[2]	bottom	bidi	5v	bd4crpf	1	ISA

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
295		SD[1]	bottom	bidi	5v	bd4crpf	1	ISA
300		SD[0]	bottom	bidi	5v	bd4crpf	1	ISA
155		SA[20]	bottom	out	5v	bt4rf	1	ISA
160		SA[19]	bottom	out	5v	bt4rf	1	ISA
165		SA[18]	bottom	out	5v	bt4rf	1	ISA
170		SA[17]	bottom	out	5v	bt4rf	1	ISA
175		SA[16]	bottom	out	5v	bt4rf	1	ISA
180		SA[15]	bottom	out	5v	bt4rf	1	ISA
185		SA[14]	bottom	out	5v	bt4rf	1	ISA
190		SA[13]	bottom	out	5v	bt4rf	1	ISA
195		SA[12]	bottom	out	5v	bt4rf	1	ISA
200		SA[11]	bottom	out	5v	bt4rf	1	ISA
205		SA[10]	bottom	out	5v	bt4rf	1	ISA
210		SA[9]	bottom	out	5v	bt4rf	1	ISA
215		SA[8]	bottom	out	5v	bt4rpf	1	ISA
220		SA[7]	bottom	out	5v	bt4rpf	1	ISA
225		SA[6]	bottom	out	5v	bt4rpf	1	ISA
230		SA[5]	bottom	out	5v	bt4rpf	1	ISA
235		SA[4]	bottom	out	5v	bt4rpf	1	ISA
240		SA[3]	bottom	out	5v	bt4rpf	1	ISA
245		SA[2]	bottom	out	5v	bt4rpf	1	ISA
250		SA[1]	bottom	out	5v	bt4rpf	1	ISA
255		SA[0]	bottom	out	5v	bt4rpf	1	ISA
310		IORD_N	bottom	out	5v	bt4rpf	1	ISA
315		IOWR_N	bottom	out	5v	bt4rpf	1	ISA
320		S1CS_N	bottom	out	5v	bt4rpf	1	ISA
325		S2CS_N	bottom	out	5v	bt4rpf	1	ISA
330		PRN1CS_N	bottom	out	5v	bt4rpf	1	ISA
340		ROMCS_N	bottom	out	5v	bt4rpf	1	ISA
345		RTCCS_N	bottom	out	5v	bt4rpf	1	ISA
350		RstISA	bottom	out	5v	bt4rpf	1	ISA
355		ADEN	bottom	out	5v	bt4rpf	1	ISA
370		IOCHRDY	bottom	in	5v	ibufuf	1	ISA
375		PDRQ	bottom	in	5v	ibufdf	1	ISA
380		PRDACK_N	bottom	out	5v	bt4rpf	1	ISA
335		PRN2CS_N	bottom	out	5v	bt4rpf	1	ISA
385		Pint	bottom	in	5v	ibufdf	1	ISA
390		S1Int	bottom	in	5v	ibufdf	1	ISA

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
395		S1RxRdy_N	bottom	in	5v	ibufnf	1	ISA
400		S1TxRdy_N	bottom	in	5v	ibufnf	1	ISA
405		S2Int	bottom	in	5v	ibufdf	1	ISA
410		S2RxRdy_N	bottom	in	5v	ibufnf	1	ISA
415		S2TxRdy_N	bottom	in	5v	ibufnf	1	ISA
420		KbDout_N	bottom	out	5v	bt6rf	1	ISA
425		KbDin_N	bottom	in	5v	schmitcnf	1	ISA
430		KbCkout_N	bottom	out	5v	bt6rf	1	ISA
435		KbCkin_N	bottom	in	5v	schmitcnf	1	ISA
440		MDout_N	bottom	out	5v	bt6rf	1	ISA
445		MDin_N	bottom	in	5v	schmitcnf	1	ISA
450		MCkout_N	bottom	out	5v	bt6rf	1	ISA
455		MCkin_N	bottom	in	5v	schmitcnf	1	ISA
460		IICDout_N	bottom	out	5v	bt6rf	1	ISA
465		IICDin_N	bottom	in	5v	schmitcnf	1	ISA
470		SCLO_N	bottom	out	5v	bt6rf	1	ISA
655		VSS	right	pwr		VSS	0	PCI
650		C_BEn[7]	right	bidi	5v	bdepcif	1	PCI
660		C_BEn[6]	right	bidi	5v	bdepcif	1	PCI
670		VDD	right	pwr		VDD	0	PCI
665		C_BEn[5]	right	bidi	5v	bdepcif	1	PCI
675		C_BEn[4]	right	bidi	5v	bdepcif	1	PCI
490		VSS	bottom	pwr		VSS	0	PCI
485		Par64	bottom	bidi	5v	bdepcif	1	PCI
495		AD[63]	bottom	bidi	5v	bdepcif	1	PCI
1025		Gnt2_N	right	out	5v	bt4rpf	1	PCI
500		VDD	bottom	pwr		VDD	0	PCI
510		AD[62]	bottom	bidi	5v	bdepcif	1	PCI
515		AD[61]	bottom	bidi	5v	bdepcif	1	PCI
985		Gnt1_N	right	out	5v	bt4rpf	1	PCI
520		VSS	bottom	pwr		VSS	0	PCI
525		AD[60]	bottom	bidi	5v	bdepcif	1	PCI
530		AD[59]	bottom	bidi	5v	bdepcif	1	PCI
1005		Req2_N	right	in	5v	ibufuf	1	PCI
540		VDD	bottom	pwr		VDD	0	PCI
535		AD[58]	bottom	bidi	5v	bdepcif	1	PCI
605		AD[57]	right	bidi	5v	bdepcif	1	PCI
545		!PCIClkBuf	bottom	clkbuf		clkc16i	0	PCI

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
610		AD[56]	right	bidi	5v	bdepcif	1	PCI
625		AD[55]	right	bidi	5v	bdepcif	1	PCI
620		Req3_N	right	in	5v	ibufuf	1	PCI
600		VDD	right	pwr		VDD	0	PCI
630		AD[54]	right	bidi	5v	bdepcif	1	PCI
640		AD[53]	right	bidi	5v	bdepcif	1	PCI
645		Gnt3_N	right	out	5v	bt4rpf	1	PCI
615		VSS	right	pwr		VSS	0	PCI
780		AD[52]	right	bidi	5v	bdepcif	1	PCI
790		AD[51]	right	bidi	5v	bdepcif	1	PCI
685		Req4_N	right	in	5v	ibufuf	1	PCI
695		VDD	right	pwr		VDD	0	PCI
800		AD[50]	right	bidi	5v	bdepcif	1	PCI
810		AD[49]	right	bidi	5v	bdepcif	1	PCI
705		Gnt4_N	right	out	5v	bt4rpf	1	PCI
715		VSS	right	pwr		VSS	0	PCI
820		AD[48]	right	bidi	5v	bdepcif	1	PCI
830		AD[47]	right	bidi	5v	bdepcif	1	PCI
725		Req5_N	right	in	5v	ibufuf	1	PCI
735		VDD	right	pwr		VDD	0	PCI
840		AD[46]	right	bidi	5v	bdepcif	1	PCI
850		AD[45]	right	bidi	5v	bdepcif	1	PCI
745		Gnt5_N	right	out	5v	bt4rpf	1	PCI
755		VSS	right	pwr		VSS	0	PCI
860		AD[44]	right	bidi	5v	bdepcif	1	PCI
870		AD[43]	right	bidi	5v	bdepcif	1	PCI
765		IntA_N	right	in	5v	ibufuf	1	PCI
775		VDD	right	pwr		VDD	0	PCI
880		AD[42]	right	bidi	5v	bdepcif	1	PCI
890		AD[41]	right	bidi	5v	bdepcif	1	PCI
785		IntB_N	right	in	5v	ibufuf	1	PCI
795		VSS	right	pwr		VSS	0	PCI
900		AD[40]	right	bidi	5v	bdepcif	1	PCI
910		AD[39]	right	bidi	5v	bdepcif	1	PCI
805		IntC_N	right	in	5v	ibufuf	1	PCI
815		VDD	right	pwr		VDD	0	PCI
920		AD[38]	right	bidi	5v	bdepcif	1	PCI
930		AD[37]	right	bidi	5v	bdepcif	1	PCI

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
825		IntD_N	right	in	5v	ibufuf	1	PCI
835		VSS	right	pwr		VSS	0	PCI
940		AD[36]	right	bidi	5v	bdepcif	1	PCI
950		AD[35]	right	bidi	5v	bdepcif	1	PCI
845		IntE_N	right	in	5v	ibufuf	1	PCI
855		VDD	right	pwr		VDD	0	PCI
960		AD[34]	right	bidi	5v	bdepcif	1	PCI
970		AD[33]	right	bidi	5v	bdepcif	1	PCI
865		PCIRst_N	right	out	5v	bt6rpf	1	PCI
875		VSS	right	pwr		VSS	0	PCI
980		AD[32]	right	bidi	5v	bdepcif	1	PCI
680		Req64_N	right	bidi	5v	bdepcif	1	PCI
885		IntF_N	right	in	5v	ibufuf	1	PCI
895		VDD	right	pwr		VDD	0	PCI
690		Ack64_N	right	bidi	5v	bdepcif	1	PCI
990		AD[0]	right	bidi	5v	bdepcif	1	PCI
905		IntG_N	right	in	5v	ibufuf	1	PCI
915		VSS	right	pwr		VSS	0	PCI
1000		AD[1]	right	bidi	5v	bdepcif	1	PCI
1010		AD[2]	right	bidi	5v	bdepcif	1	PCI
925		IntH_N	right	in	5v	ibufuf	1	PCI
935		VDD	right	pwr		VDD	0	PCI
1020		AD[3]	right	bidi	5v	bdepcif	1	PCI
1030		AD[4]	right	bidi	5v	bdepcif	1	PCI
945		IdSel	right	in	5v	ibuff	1	PCI
955		VSS	right	pwr		VSS	0	PCI
1040		AD[5]	right	bidi	5v	bdepcif	1	PCI
1045		AD[6]	right	bidi	5v	bdepcif	1	PCI
965		Req1_N	right	in	5v	ibufuf	1	PCI
975		VDD	right	pwr		VDD	0	PCI
1055		AD[7]	right	bidi	5v	bdepcif	1	PCI
700		C_BEn[0]	right	bidi	5v	bdepcif	1	PCI
995		VSS	right	pwr		VSS	0	PCI
1060		AD[8]	right	bidi	5v	bdepcif	1	PCI
1070		AD[9]	top	bidi	5v	bdepcif	1	PCI
1015		VDD	right	pwr		VDD	0	PCI
1075		AD[10]	top	bidi	5v	bdepcif	1	PCI
1085		AD[11]	top	bidi	5v	bdepcif	1	PCI

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1035		VSS	right	pwr		VSS	0	PCI
1095		AD[12]	top	bidi	5v	bdepcif	1	PCI
1105		AD[13]	top	bidi	5v	bdepcif	1	PCI
1050		VDD	right	pwr		VDD	0	PCI
1110		AD[14]	top	bidi	5v	bdepcif	1	PCI
1120		AD[15]	top	bidi	5v	bdepcif	1	PCI
1065		VSS	right	pwr		VSS	0	PCI
710		C_BEn[1]	right	bidi	5v	bdepcif	1	PCI
1125		PAR	top	bidi	5v	bdepcif	1	PCI
1080		VDD	top	pwr		VDD	0	PCI
1135		SERR_N	top	bidi	5v	bdepcif	1	PCI
1090		VSS	top	pwr		VSS	0	PCI
1140		PERR_N	top	bidi	5v	bdepcif	1	PCI
1100		VDD	top	pwr		VDD	0	PCI
720		STOP_N	right	bidi	5v	bdepcif	1	PCI
1150		DEVSEL_N	top	bidi	5v	bdepcif	1	PCI
1115		VSS	top	pwr		VSS	0	PCI
730		TRDY_N	right	bidi	5v	bdepcif	1	PCI
740		IRDY_N	right	bidi	5v	bdepcif	1	PCI
1130		VDD	top	pwr		VDD	0	PCI
750		FRAME_N	right	bidi	5v	bdepcif	1	PCI
760		C_BEn[2]	right	bidi	5v	bdepcif	1	PCI
1145		VSS	top	pwr		VSS	0	PCI
1155		AD[16]	top	bidi	5v	bdepcif	1	PCI
1165		AD[17]	top	bidi	5v	bdepcif	1	PCI
1160		VDD	top	pwr		VDD	0	PCI
1170		AD[18]	top	bidi	5v	bdepcif	1	PCI
1180		AD[19]	top	bidi	5v	bdepcif	1	PCI
1175		VSS	top	pwr		VSS	0	PCI
1185		AD[20]	top	bidi	5v	bdepcif	1	PCI
1195		AD[21]	top	bidi	5v	bdepcif	1	PCI
1190		VDD	top	pwr		VDD	0	PCI
1200		AD[22]	top	bidi	5v	bdepcif	1	PCI
1210		AD[23]	top	bidi	5v	bdepcif	1	PCI
1205		VSS	top	pwr		VSS	0	PCI
770		C_BEn[3]	right	bidi	5v	bdepcif	1	PCI
1215		AD[24]	top	bidi	5v	bdepcif	1	PCI
1220		VDD	top	pwr		VDD	0	PCI

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1225		AD[25]	top	bidi	5v	bdepcif	1	PCI
1230		AD[26]	top	bidi	5v	bdepcif	1	PCI
1240		VSS	top	pwr		VSS	0	PCI
1245		AD[27]	top	bidi	5v	bdepcif	1	PCI
1250		AD[28]	top	bidi	5v	bdepcif	1	PCI
1255		VDD	top	pwr		VDD	0	PCI
1260		AD[29]	top	bidi	5v	bdepcif	1	PCI
1265		AD[30]	top	bidi	5v	bdepcif	1	PCI
1270		AD[31]	top	bidi	5v	bdepcif	1	PCI
1275		VSS	top	pwr		VSS	0	PCI
1285		VDD	top	pwr		VDD	0	Video
1290		VGenClk	top	in	5v	ibufdf	1	Video
1295		VSS	top	pwr		VSS	0	Video
1300		Hsync1	top	out	5v	bt4rpf	1	Video
1305		Hsync2	top	out	5v	bt4rpf	1	Video
1310		CBlank	top	out	5v	bt4rpf	1	Video
1315		CoreVDD	top	pwr		VDD2	0	CorePwr
1325		Field	top	out	5v	bt4rpf	1	Video
1330		VBlank	top	out	5v	bt4rpf	1	Video
1335		HBlank	top	out	5v	bt4rpf	1	Video
1340		GBELok	top	out	5v	bt4rpf	1	Video
1320		CoreVSS	top	pwr		VSS2	0	CorePwr
1345		D1OutF[0]	top	out	5v	bt4rpf	1	Video
1350		D1OutF[1]	top	out	5v	bt4rpf	1	Video
1355		D1OutF[2]	top	out	5v	bt4rpf	1	Video
1360		D1OutF[3]	top	out	5v	bt4rpf	1	Video
1365		D1OutF[4]	top	out	5v	bt4rpf	1	Video
1370		D1OutF[5]	top	out	5v	bt4rpf	1	Video
1375		D1OutF[6]	top	out	5v	bt4rpf	1	Video
1380		D1OutF[7]	top	out	5v	bt4rpf	1	Video
1385		D1OutF[8]	top	out	5v	bt4rpf	1	Video
1390		D1OutF[9]	top	out	5v	bt4rpf	1	Video
1395		VDD	top	pwr		VDD	0	Video
1400		D1_ClkEF	top	out	5v	bt4rpf	1	Video
1405		D1OutE[0]	top	out	5v	bt4rpf	1	Video
1410		D1OutE[1]	top	out	5v	bt4rpf	1	Video
1415		D1OutE[2]	top	out	5v	bt4rpf	1	Video
1420		D1OutE[3]	top	out	5v	bt4rpf	1	Video

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1425		D1OutE[4]	top	out	5v	bt4rpf	1	Video
1430		D1OutE[5]	top	out	5v	bt4rpf	1	Video
1435		D1OutE[6]	top	out	5v	bt4rpf	1	Video
1440		D1OutE[7]	top	out	5v	bt4rpf	1	Video
1445		D1OutE[8]	top	out	5v	bt4rpf	1	Video
1450		D1OutE[9]	top	out	5v	bt4rpf	1	Video
1455		VSS	top	pwr		VSS	0	Video
1460		D1_ClkD	top	in	5v	ibufdf	1	Video
1465		SelC_DN	top	out	5v	bt4rpf	1	Video
1470		D1InCD[0]	top	in	5v	ibufdf	1	Video
1475		D1InCD[1]	top	in	5v	ibufdf	1	Video
1480		D1InCD[2]	top	in	5v	ibufdf	1	Video
1485		D1InCD[3]	top	in	5v	ibufdf	1	Video
1490		D1InCD[4]	top	in	5v	ibufdf	1	Video
1495		CoreVDD	top	pwr		VDD2	0	CorePWR
1505		D1InCD[5]	top	in	5v	ibufdf	1	Video
1510		D1InCD[6]	top	in	5v	ibufdf	1	Video
1515		D1InCD[7]	top	in	5v	ibufdf	1	Video
1500		CoreVSS	top	PWR		VSS2	0	CorePWR
1520		D1InCD[8]	top	in	5v	ibufdf	1	Video
1525		D1InCD[9]	top	in	5v	ibufdf	1	Video
1530		D1_ClkC	top	in	5v	ibufdf	1	Video
1535		VDD	top	pwr		VDD	1	Video
1540		D1_ClkB	top	in	5v	ibufdf	1	Video
1545		SelA_BN	top	out	5v	bt4rpf	1	Video
1550		D1InAB[0]	top	in	5v	ibufdf	1	Video
1555		D1InAB[1]	top	in	5v	ibufdf	1	Video
1560		D1InAB[2]	top	in	5v	ibufdf	1	Video
1565		D1InAB[3]	top	in	5v	ibufdf	1	Video
1570		D1InAB[4]	top	in	5v	ibufdf	1	Video
1575		D1InAB[5]	top	in	5v	ibufdf	1	Video
1580		D1InAB[6]	top	in	5v	ibufdf	1	Video
1585		D1InAB[7]	top	in	5v	ibufdf	1	Video
1590		D1InAB[8]	top	in	5v	ibufdf	1	Video
1595		D1InAB[9]	top	in	5v	ibufdf	1	Video
1605		VSS	top	PWR		VSS	0	Video
480			bottom			PwrCut	0	Cut
1280			top			PwrCut	0	Cut

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1615			left			blank	0	SystemPLL
1620		SyRefClk	left	in	3v	ddrv	0	SystemPLL
1625		SyPLLVDD	left	pwr		pllvdd	0	SystemPLL
1630		SyPLLAGND	left	pwr		pllagn	0	SystemPLL
1635		SyPLLlp2	left	bidi	3v	plllp2	0	SystemPLL
1640		SyPLLVSS	left	pwr		pllvss	0	SystemPLL
1645			left			blank	0	SystemPLL
1650			left	pwr		PwrCut	0	Cut
1655		VDD	left	pwr		VDD	0	
1660		!ClkBuf133	left			clkc16i	0	
1665		VSS	left	pwr		VSS	0	
1670			left			PwrCut	0	Cut
1675		VDD	left	pwr		VDD	0	
1680		!ClkBuf66	left			clkc16i	0	
1685		VSS	left	pwr		VSS	0	
1690			left			PwrCut	0	Cut
1695		CoreVDD	left	pwr		VDD2	0	CorePwr
1705		VDD	left	pwr		VDD	0	Crime
1715		CrmAD[0]	left	bidi	3v	bd8c	1	Crime
1710		VSS	left	pwr		VSS	0	Crime
1700		CoreVSS	left	pwr		VSS2	0	CorePwr
1720		CrmAD[1]	left	BIDI	3v	bd8c	1	Crime
1725		VDD	left	pwr		VDD	0	Crime
1735		CrmAD[2]	left	BIDI	3v	bd8c	1	Crime
1730		VSS	left	pwr		VSS	0	Crime
1740		CrmAD[3]	left	BIDI	3v	bd8c	1	Crime
1745		VDD	left	pwr		VDD	0	Crime
1755		CrmAD[4]	left	BIDI	3v	bd8c	1	Crime
1750		VSS	left	pwr		VSS	0	Crime
1760		CrmAD[5]	left	BIDI	3v	bd8c	1	Crime
1765		VDD	left	pwr		VDD	0	Crime
1775		CrmAD[6]	left	BIDI	3v	bd8c	1	Crime
1770		VSS	left	pwr		VSS	0	Crime
1780		CrmAD[7]	left	BIDI	3v	bd8c	1	Crime
1785		VDD	left	pwr		VDD	0	Crime
1795		CrmAD[8]	left	BIDI	3v	bd8c	1	Crime
1790		VSS	left	pwr		VSS	0	Crime
1800		CrmAD[9]	left	BIDI	3v	bd8c	1	Crime

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1805		VDD	left	pwr		VDD	0	Crime
1815		CrmAD[10]	left	BIDI	3v	bd8c	1	Crime
1810		VSS	left	pwr		VSS	0	Crime
1820		CrmAD[11]	left	BIDI	3v	bd8c	1	Crime
1825		VDD	left	pwr		VDD	0	Crime
1835		CrmAD[12]	left	BIDI	3v	bd8c	1	Crime
1830		VSS	left	pwr		VSS	0	Crime
1840		CrmAD[13]	left	BIDI	3v	bd8c	1	Crime
1845		VDD	left	pwr		VDD	0	Crime
1855		CrmAD[14]	left	BIDI	3v	bd8c	1	Crime
1850		VSS	left	pwr		VSS	0	Crime
1860		CrmAD[15]	left	BIDI	3v	bd8c	1	Crime
1865		VDD	left	pwr		VDD	0	Crime
1875		CrmAD[16]	left	BIDI	3v	bd8c	1	Crime
1870		VSS	left	pwr		VSS	0	Crime
1880		CrmAD[17]	left	BIDI	3v	bd8c	1	Crime
1885		VDD	left	pwr		VDD	0	Crime
1895		CrmAD[18]	left	BIDI	3v	bd8c	1	Crime
1890		VSS	left	pwr		VSS	0	Crime
1900		CrmAD[19]	left	BIDI	3v	bd8c	1	Crime
1905		VDD	left	pwr		VDD	0	Crime
1915		CrmAD[20]	left	BIDI	3v	bd8c	1	Crime
1910		VSS	left	pwr		VSS	0	Crime
1920		CrmAD[21]	left	BIDI	3v	bd8c	1	Crime
1925		VDD	left	pwr		VDD	0	Crime
1935		CrmAD[22]	left	BIDI	3v	bd8c	1	Crime
1930		VSS	left	pwr		VSS	0	Crime
1940		CrmAD[23]	left	BIDI	3v	bd8c	1	Crime
1945		VDD	left	pwr		VDD	0	Crime
1950		CrmAD[24]	left	BIDI	3v	bd8c	1	Crime
1955		VSS	left	pwr		VSS	0	Crime
1960			left			PwrCut	0	Cut
1965		VDD	left	pwr		VDD	0	
1970		!ClkBuf33	left			clkc16i	0	
1975		VSS	left	pwr		VSS	0	
1980			left			PwrCut	0	Cut
1985		VDD	left	pwr		VDD	0	Crime
1995		CrmAD[25]	left	BIDI	3v	bd8c	1	Crime

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
1990		VSS	left	pwr		VSS	0	Crime
2000		CrmAD[26]	left	BIDI	3v	bd8c	1	Crime
2005		VDD	left	pwr		VDD	0	Crime
2015		CrmAD[27]	left	BIDI	3v	bd8c	1	Crime
2010		VSS	left	pwr		VSS	0	Crime
2020		CrmAD[28]	left	BIDI	3v	bd8c	1	Crime
2025		VDD	left	pwr		VDD	0	Crime
2035		CrmAD[29]	left	BIDI	3v	bd8c	1	Crime
2030		VSS	left	pwr		VSS	0	Crime
2040		CrmAD[30]	left	BIDI	3v	bd8c	1	Crime
2045		VDD	left	pwr		VDD	0	Crime
2050		CoreVDD	left	pwr		VDD2	0	CorePwr
2065		CrmAD[31]	left	BIDI	3v	bd8c	1	Crime
2060		VSS	left	pwr		VSS	0	Crime
2055		CoreVSS	left	pwr		VSS2	0	CorePwr
2070		Tin	left	in	3v	ibuf	1	Crime
2075		VDD	left	pwr		VDD	0	Crime
2085		Tout	left	out	3v	bt8	1	Crime
2080		VSS	left	pwr		VSS	0	Crime
2090		Pack	left	out	3v	bt8	1	Crime
2095		VDD	left	pwr		VDD	0	Crime
2100		Mack	left	in	3v	ibuf	1	Crime
2105		SysReset	left	in	3v	ibuf	1	Crime
2110			left			PwrCut	0	Cut
2135		JTDI	left	in	5v	ibufdf	0	Test
2140		JTDO	left	out	5v	bt4rpf	0	Test
2145		JTCK	left	in	5v	ibufdf	0	Test
2150		VSS	left	pwr		VSS	0	Test
2155		JTMS	left	in	5v	ibufdf	0	Test
2160		JTRST_N	left	in	5v	ibufdf	0	Test
2165		IOoff_N	left	in	5v	ibufuf	0	Test
2170		TDClk	left	in	5v	ibufdf	0	Test
2175		TClkSelB	left	in	5v	ibufdf	0	Test
2180		TClkSelA	left	in	5v	ibufdf	0	Test
2185		VDD	left	pwr		VDD	0	Test
2115		PromClrEn_N	left	in	5v	ibufuf	1	Test
2120		CMITrig	left	out	5v	bt4rpf	1	Test
2125		SerNumDat	left	bidi	5v	bd4crpf	1	Test

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
2130		TE	left	in	5v	ibufdf	0	Test
2190		IDDTN	left	in		iddtn	0	Test
2195		PMONtest	left	in			0	Test
2200		PMONout	left	out			0	Test
1610						CrnrPwr Cut	0	Cut
50		!TxClkBuf	bottom	clkbuf		clkc16i	0	Ethernet
55		VSS	bottom	pwr		VSS	0	Ethernet
60		!RxClkBuf	bottom	clkbuf		clkc16i	0	Ethernet
65		VDD	bottom	pwr		VDD	0	Ethernet
550		VSS	bottom	pwr		VSS	0	PCI
555						CrnrPwr Cut	0	Cut
560			right			blank	0	PCIPLL
565		PCIRefClk	right	in	3v	ddrv	0	PCIPLL
570		PCIPLLVDD	right	pwr		pllvdd	0	PCIPLL
575		PCIPLLAGND	right	pwr		pllagn	0	PCIPLL
580		PCIPLLlp2	right	in		plllp2	0	PCIPLL
585		PCIPLLVSS	right	pwr		pllvss	0	PCIPLL
590			right			blank	0	PCIPLL
595			right			PwrCut	0	Cut
635		VDD	right	pwr		VDD	0	PCI
1600		D1_ClkA	top	in	5v	ibufdf	1	Video
1321		ExtLokH	top	in	5v	ibuff	1	Video
1322		ExtLokV	top	in	5v	ibuff	1	Video
1323		ExtLokF	top	in	5v	ibuff	1	Video
1324		ExtLokClk	top	in	5v	ibuff	1	Video
1681		!ClkBuf66	left			clkc16i	0	
1971		!ClkBuf33	left			clkc16i	0	
1691		SysClkSync	left	in	3v	ibufn	1	SystemPLL
347		GPCS_N	bottom	out	5v	bt4rpf	1	ISA
382		DMATC	bottom	out	5v	bt4rpf	1	ISA
417		RTCIrq_N	bottom	in	5v	schmitcnf	1	ISA
471		SCLI_N	bottom	in	5v	schmitcnf	1	ISA
2166		PCIPllOut	left	out	5v	bt4rpf	0	Test
2167		SysPllOut	left	out	5v	bt4rpf	0	Test
2168		PllCntClr	left	in	5v	ibufnuf	0	Test
2126		VDNSW	left	in	5v	schmitcf	1	Test
2127		VUPSW	left	in	5v	schmitcf	1	Test

slot number	pin number	signal name	side	io type	5v/3v	I/O Cell type	Jtag	group
2128		spare1	left	in	5v	schmitcf	1	Test
2129		spare2	left	in	5v	schmitcf	0	Test