AMD

Writing an Ethernet Device Driver for the Alchemy™ Au1000™ Processor from AMD

Application Note

Revision: **003**
Issue Date: **September 2001**

Contacts

www.amd.com pcs.support@amd.com

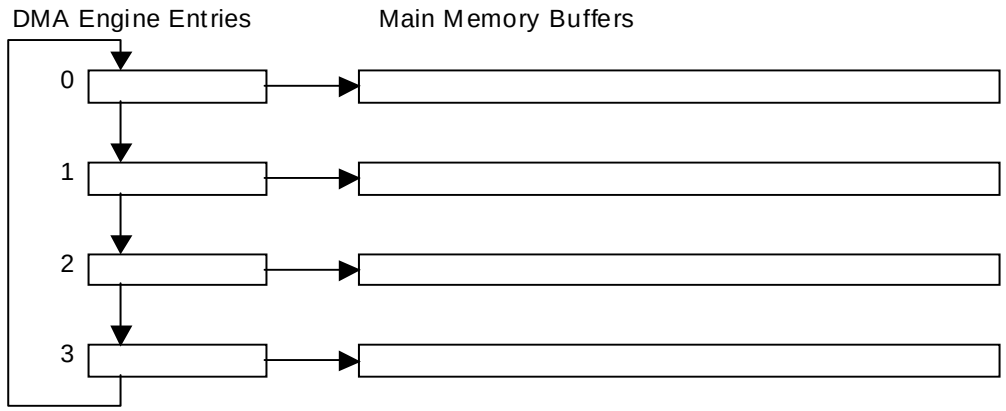
Trademarks

The Au1000™ processor integrates two Ethernet Media Access Controllers, MAC. The MACs are identical and independent of one another. Each MAC communicates to a Media Independent Interface, MII, and is capable of supporting 10/100Mbps Ethernet. This document discusses how to write a MAC device driver for the Au1000 processor.

Overview

The MAC is a simple device to utilize; once configured, the packet receive and transmit operations are a “fire and forget” scenario.

The “fire and forget” scenario is supported by two DMA engines which independently handle receiving and transmitting frames. Each DMA engine processes a 4-entry ring from/into which packets are moved.



There are two rings, one for transmit and one for receive. As the DMA engine completes an operation on an entry, it moves on to the next entry in the ring. Status bits and interrupts indicate when the operation has completed.

A transmit DMA engine entry is composed of the `macdma_txstat`, `macdma_txlen` and `macdma_txaddr` register triplet. A receive DMA engine entry is composed of the `macdma_rxstat` and `macdma_rxaddr` register pair. Note that the Ethernet DMA engines are dedicated engines; separate from the general purpose DMA controller.

The control of the MAC by software is described below. The discussions below apply equally to both MACs, however, where coding examples are given, MAC0 will be used.

Programming Considerations

When programming the MAC, take into consideration the following:

- Before the MAC can be used, the entire MAC module must be enabled via the `macen_mac0` register. An unused MAC should be disabled to reduce power consumption.

- All register accesses must be 32-bits. Furthermore, 32-bit register accesses works for both endian modes. NOTE: This is not true for the buffer contents, which is independently controlled via the `mac_control[EM]` bit.

As with all peripherals integrated into the Au1000 processor, all register accesses should be through the KSEG1 region. This region is un-mapped, and more importantly, non-cached. All non-cached accesses proceed through the write buffer, WB. (Please see the Au1000 Processor Databook[1].) As a result, accesses to the MAC registers must flush the write buffer (before and/or) after a register write. Flushing the write buffer also guarantees that previous writes have been posted to the peripheral (as opposed to sitting indefinitely in the write buffer until a write buffer flush event occurs). The write buffer is flushed with a SYNC instruction (rather than a non-cached read since a read of a peripheral register can have unintended side-effects).

MAC Initialization

Initialization of the MAC is performed using these recommended steps.

1. Pre-allocate four buffers for the receiver buffer DMA ring. See discussion on buffer management for more information.
2. Pre-allocate four buffers for the transmitter buffer DMA ring. See discussion on buffer management for more information.
3. Place the MAC in reset. This is accomplished by writing the value `0x00000040` to the `macen_mac0` register. This disables the entire MAC module and its DMA engines.
4. Initialize the receive DMA ring. For each of the four DMA entries, write the value `0x00000000` to the `macdma_rxstat` register, and write the address of a receive buffer along with EN=1 bit to the `macdma_rxaddr` register. NOTE that the receive buffers must be cache line (32-byte) aligned.
5. Initialize the transmit DMA ring. For each of the four DMA entries, write the value 0x00000000 to the `macdma_txaddr`, `macdma_txstat` and `macdma_txlen` registers. NOTE that the transmit buffers must be cache line (32-byte) aligned. Note that the `macdma_txaddr[EN]` bit must not be set until a frame is actually ready to be transmitted.
6. Enable the MAC module by performing this sequence:
    a. Write `0x00000041` to `mac0_enable`. This enable clocks to the MAC.
    b. Write `0x00000033` to `mac0_enable`. This takes the MAC out of reset, enables coherent transactions and only passes valid frames to memory. Steps 6a and 6b must not be combined, but performed in sequence.
7. Configure the MAC module.
    a. Write the `mac_control` register. The exact value will vary depending upon the application, but the value `0x00800000` will work in most circumstances. NOTE:

That EM (bit 30) bit of this register indicates if the buffer contents are written by the processor in big endian or little endian mode. If the processor is running in big endian, then the value `0x40800000` is desired.

b. Write the `mac_addrhigh` and `mac_addrlow` registers with the 48-bit Ethernet MAC address for this controller. For example, if the MAC address is 01:23:45:67:89:AB, then `mac_addrhigh` is written with `0x0000AB89` and `mac_addrlow` is written with `0x67452301`.

c. Write the `mac_hashhigh` and `mac_hashlow` registers with `0xFFFFFFFF`. This value accepts all multicast frames (if enabled by `mac_control[PM,HO,HP]` bits).

8. Configure the MII. Consult the documentation for the PHY device for any special settings. NOTE: Must poll the `mac_miictrl[MB]` bit before each MII access. NOTE: The MAC and the PHY must agree on duplex, else collisions between the MAC and the PHY occur on the MII interface and significantly degrade performance.
9. Initialize a driver global variable named `NextRxBuffer`. This variable is initialized from the `(macdma_rxaddr[CB] >> 2)` of any receive DMA entry. The CB field points to the entry that will be processed next by the receive DMA engine. It's value out of reset is not guaranteed to be zero.
10. Initialize a driver global variable named `NextTxBuffer` and `LastTxBuffer`. Both variables are initialized from the `(macdma_txaddr[CB] >> 2)` of any transmit DMA entry. The CB field points to the entry that will be processed next by the transmit DMA engine. Its value out of reset is not guaranteed to be zero.
11. Enable the transmit and receive DMA engines. Utilizing a read-modify-write operation, set the `mac_control[TE,RE]` bits. Now enabled, Ethernet packet reception and transmission can occur.

The MAC should now be operational.

MAC Interrupts

There are two possible options in configuring the interrupt controller, IC0, for MAC interrupts: rising edge or high level. The reason for the two options is a result of the interrupt logic. The MAC interrupt signal is a logical OR'ing of all the DN bits in the macdma_rxaddr and `macdma_txaddr` DMA registers. When a DMA entry is consumed, the DN bit transitions from a 0 to 1, thus a rising edge interrupt is possible. Similarly, as long as a DN bit is set, high level interrupts are also possible. The DN bit is cleared only by software when handling a completed DMA entry.

The interrupt controller, IC0, for MAC interrupts (interrupt 28 for MAC0 and interrupt 29 for MAC1) should be configured as rising-edge or high-level. If the MAC interrupt is configured for rising-edge, then it must be acknowledged by writing 1 to the corresponding bit in `ic0_risingclr` in IC0. The receive and transmit algorithms described below work with either interrupt type. However, the use of

rising edge interrupts requires a mature driver (to avoid the race condition of taking the interrupt, acknowledging the interrupt and receiving additional interrupts while handling the interrupt). Thus, many drivers may (and should) choose to utilize high-level interrupts, which is more forgiving during driver development, and generally more robust as interrupts are not easily lost.

When a MAC interrupt does occur, software must examine the DMA entries; there is no interrupt status register to check. Since the receiver can not throttle traffic, for most applications it is best to handle the receive buffers first, and then the transmit buffers.

In other words, the interrupt service routine for the MAC should call the MAC receive algorithm and then the MAC transmit algorithm. The algorithms are designed to process zero or more completed buffers.

If the application is polled, rather than interrupt driven, then it too should call the MAC receive and transmit algorithms described below, since they are designed to process zero or more completed buffers.

MAC Frame Receive

The algorithm for processing buffers which the receive DMA engine has filled is simple and robust. Its robustness is due to the fact that it can handle zero or more completed buffers, independent of whether or not the driver is interrupt driven or polled. The algorithm pseudo-code is:

```
while (macdma_rxaddr[NextRxBuffer].DN == 1)
{
     Examine macdma_rxstat[NextRxBuffer].FA for errors
     If no errors, process buffer contents
     Zero macdma_rxstat[NextRxBuffer]
     Write macdma_rxaddr[NextRxBuffer] with memory buffer address
     and DN=0,EN=1 // clear IRQ
     Advance NextRxBuffer: if (++NextRxBuffer == 4) NextRxBuffer = 0
}
```

This algorithm assumes that the same memory buffer is used for the same DMA entry. In the discussion on Buffer Management , the receive algorithm is changed to accommodate managing multiple memory buffers for the DMA entries. It is necessary to write the address of the memory buffer every time to `macdma_rxaddr` since the DMA engine modifies the value in the register.

In the event that the DMA engine discovers that the next buffer entry is not yet available (macdma_rxaddr[EN]=0), the engine stalls until the EN bit is set by software. In other words, frames

are dropped and the CB field does not advance until the buffer entry becomes available again. There is no mechanism for reporting that frames have been.

MAC Frame Transmit

The algorithm for transmitting frames is also straightforward. This code to transmit a frame is independent of whether or not the driver is interrupt driven or polled. The algorithm pseudo-code is:

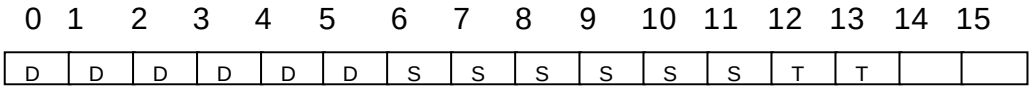
```
// Handle previously transmitted frames
while (1)
{
     if (macdma_txaddr[LastTxBuffer].DN == 1)
     {
          Examine macdma_txstat[LastTxBuffer] for errors
          Write macdma_txaddr[LastTxBuffer].DN=0 // clear IRQ
          Advance LastTxBuffer: if (++LastTxBuffer == 4)
          LastTxBuffer = 0
          If (LastTxBuffer == NextTxBuffer) BREAK;
     }
     else BREAK;
}

// Ensure previous transmit of this entry completed
While (macdma_txaddr[NextTxBuffer].EN == 1)
     ;
Copy data into transmit buffer
Write macdma_txstat[NextTxBuffer] with 0x00000000
Write macdma_txlen[NextTxBuffer] with packet length
Write macdma_txaddr[NextTxBuffer] with address of memory buffer and
DN=0,EN=1
Advance NextTxBuffer: if (++NextTxBuffer == 4) NextTxBuffer = 0
```

This algorithm assumes that the same memory buffer is used for the DMA entry. The transmit algorithm can be changed to accommodate multiple memory buffers for the DMA entries, as discussed in the Buffer Management section. It is necessary to write the address of the memory buffer every time to `macdma_txaddr` since the DMA engine modifies the value in the register.

Processing MAC Frames

When preparing a frame for transmission, or processing a received frame, it is important to understand the layout of the frame in memory.



The diagram above illustrates that the destination (D) Ethernet mac address occupies the first 6 bytes, the source (S) Ethernet mac address occupies the next 6 bytes, and the frame type/length (T) occupies the next two bytes. The data payload follows, starting at byte offset 14.

Since the DMA engine buffers are on cache line boundaries, an important side effect occurs with the payload. The packet is aligned to a 32-bit boundary, but the payload, starting at offset 14, is NOT aligned on a 32-bit boundary. In the case of a TCP/IP packet, the IP header begins at byte offset 14, and is mis-aligned. Upper layer software must take care in accessing 32-bit values in the packet since mis-aligned accesses are not supported in hardware.

Buffer Allocation

The memory buffers accessed by the DMA engines must be cache-line (32-byte) aligned, and must 2048 bytes in size. A 2048 byte buffer is large enough to hold the largest possible Ethernet packet, 1518 (6 + 6 + 2 + 1500 + 4) bytes and, under specific circumstances, maximum jabber timeout.

The addresses programmed into the DMA entries are physical addresses. The addresses utilized by the DMA engine do not pass through translation, and therefore must be physical addresses. The buffers can be referenced in KSEG0 or KSEG1 regions since the data cache on the Au1000 maintains coherency.

In the absence of a special memory allocator which is capable of providing 32-byte aligned buffers, do the following:

- Request/allocate all receiver buffers in one chunk.
- The value 2048 is an integer multiple of the cache-line size (32 bytes), and need not be adjusted.
- Request an additional 32 bytes which can be used to guarantee cache-line alignment.

A code example demonstrates these principles.

```
static char rx_buffer_pool[(2048 * MAX_RX_BUFS) + 32];
```

or

```
char *rx_buffer_pool = malloc((2048 * MAX_RX_BUFS) + 32);
```

The first cache-aligned buffer is then guaranteed to be at this address:

```
char *first_buffer = (char *)(((unsigned long)rx_buffer_pool + 32) &
~0x1F);
```

Subsequent buffers are guaranteed to be cache-aligned, each 2048 bytes in length.

```
for (i = 0; i < MAX_RX_BUFS; ++i)
{
     rx_buffer_addr[i] = &first_buffer[i * 2048];
}
```

Likewise, the same can be done for the transmit buffers.

Buffer Management

In applications where the Au1000 processor is to process lots of Ethernet traffic (e.g. on a heavily-loaded network or in promiscuous mode), the simple DMA buffer management strategy of one fixed buffer per DMA entry previously suggested becomes inadequate. A more aggressive buffering strategy is necessary.

At 100Mbps, with back-to-back [Ethernet legal] minimum frames of 64 bytes, an entry is consumed every 6.72 microseconds.

$$\left(\left(\frac{1}{\left(\frac{100Mbits}{s}\right)*\left(\frac{1byte}{8bits}\right)}\right)*\left(\frac{7+1+64byte}{1\,frame}\right)\right)+0.96us=\frac{6.72us}{frame}$$

Note: *The 7+1 term is the preamble and start frame delimiter. 0.96us is the inter-frame minimum delay.*

With 4 DMA entries, the CPU must, at worst case, service the receive DMA entries within 26.88 microseconds, else further Ethernet receive frames will be dropped.

At best case (100% cache hit rate), 26.88 microseconds equates to these number of instructions before the next packet arrives (and thus requires a valid DMA entry):

Frequency (MHz)	# of Instructions per frame	# of Instructions for four frames
266	1787	7150
400	2688	10752
500	3360	13440

In the real world, the number of instructions is likely to be less due to a less-than perfect cache hit rate and memory latencies of load/stores. In the heavily loaded environment, a different buffering scheme is needed. The following is one possible solution.

For each the transmitter and the receiver, preallocate a large number of buffers (see Buffer Allocation above). Depending upon available memory, at least 32 for the receiver would be a comfortable start.

When an interrupt arrives, immediately start to examine the receiver DMA ring (see MAC Frame Receive). For all the entries that have been filled by the DMA, place on a “ToDo” list, and immediately update macdma_rxaddr with a new/different buffer from the receive buffer pool.

Having serviced the receiver DMA engine, now process the buffer(s) that are on the “ToDo” list. It is important to provide the DMA engine entries with new buffers **prior** to processing the packet(s). In doing so, the DMA engine will have a higher probability of a valid buffer(s) to which it can transfer Ethernet frame(s).

References:

[1] Au1000™ Processor Data Book.