Micriμm

Empowering Embedded Systems

μC/TCPIP

Hardware Optimization

www.Micrium.com

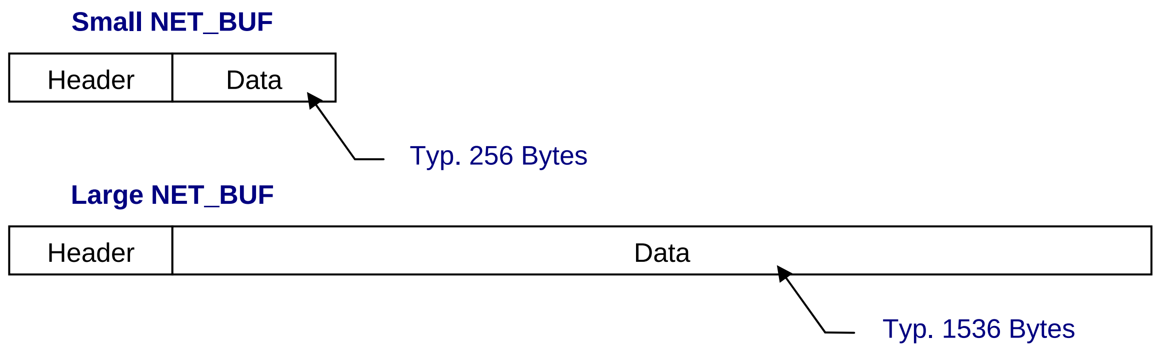
Introduction

µC/TCP-IP is a TCP/IP stack that was written from the ground up to be used for embedded application. The TCP/IP stack assumes that most of the processing be handled in software. However, a number of hardware improvements can be done to the stack in order to speed processing:

1) Byte stuffing from MAC
2) DMA driven copy
3) Computation of checksum

Byte Stuffing from the MAC

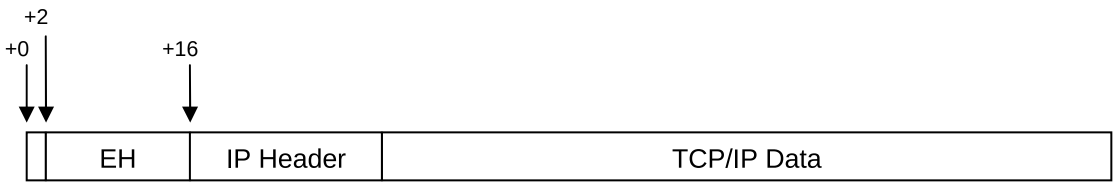
µC/TCP-IP uses network buffers (NET_BUF) when processing packets to and from the MAC. There are two types of µC/TCP-IP buffers, small and large. Small buffers are used when data packets are typically smaller than 256 bytes (actually, the size is user configurable at compile time). Large buffers are typically able to hold packets of up to 1536 bytes (also configurable at compile time). A NET_BUF looks as follows:



Small and large NET_BUFs

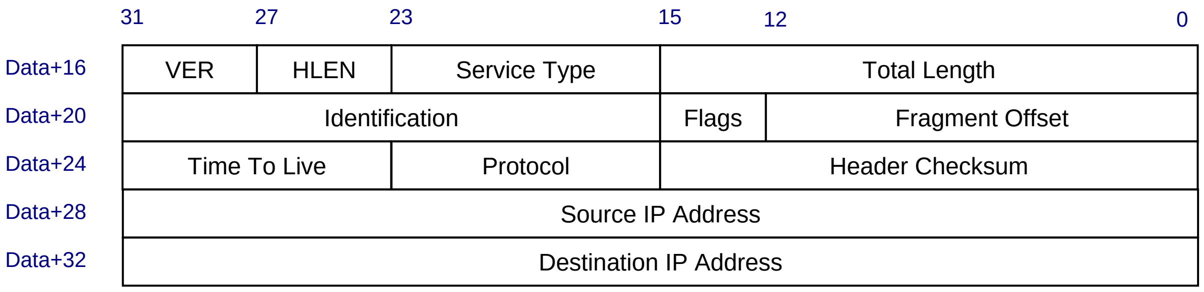
The 'Data' portion of the NET_BUF holds a packet received or sent from/to the MAC. The 'Data' is **ALWAYS** aligned on a 32-bit boundary (i.e. the start address of 'Data' is 0x???????0, 0x???????4, 0x??????8 or 0x???????C).

The Ethernet packet is placed into the 'Data' portion of the `NET_BUF` starting at offset `+2` as shown below.



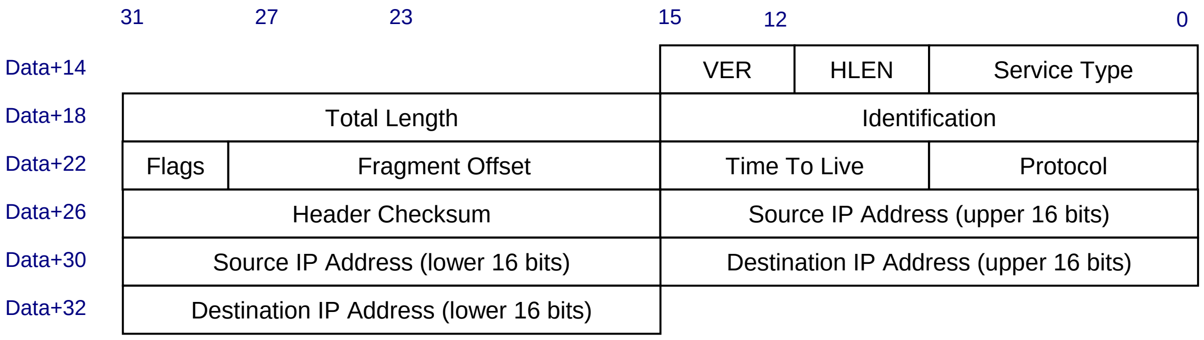
Data portion of a NET_BUFs

You will note that the Ethernet Header (EH) is typically 14 bytes in length. When we designed µC/TCP-IP, we decided to align the IP header on a 32-bit boundary because this gives us faster access to the contents of the IP header on a 32-bit processor. A typical IP header (as found in the 'Data' portion of a `NET_BUF`) looks as shown below:



µC/TCP-IP's IP Header aligned on 32-bit boundary

If we had ligned up the Ethernet header at offset `+0` in the 'Data' portion of the NET_BUF, the IP header (and UDP and TCP headers) would aligned on a 16-bit and thus would have required some run-time processing to extract the Source and Destination addresses (as well as other fields) using two 32-bit word accesses.



IP Header aligned on 16-bit boundary

With a 16-bit aligned IP header, the software has to perform the following operation to obtain the Source IP address:

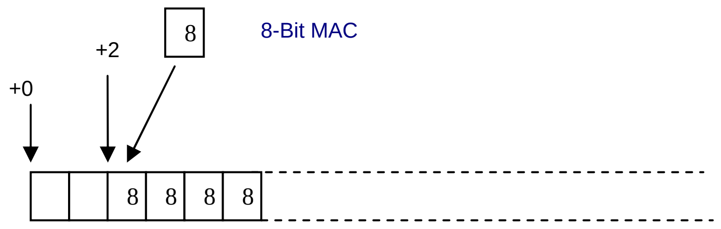
```
Source IP address = ((Data+28) << 16) + (((Data+30) >> 16) & 0x0000FFFF)
```

Similarly, for the Destination IP address:

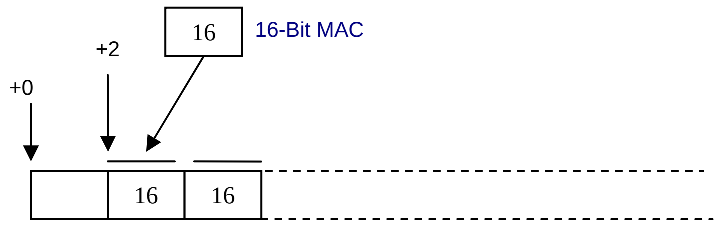
```
Destination IP address = ((Data+30) << 16) + (((Data+32) >> 16) & 0x0000FFFF)
```

Although on most 32-bit architectures, shifting and adding is not a huge performance hit, the misalignment of the header does not only affect the IP header but the other headers as well (UDP and TCP) which would also be non 32-bit aligned.

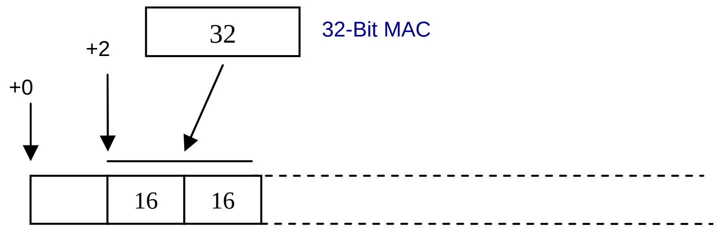
µC/TCP-IP's choice of aligning the IP header on a 32-bit boundary is more efficient (when processing the TCP/IP stack) when we use an 8-bit or 16-bit Ethernet controller (MAC) because we can move the MAC data 8 or 16 bits at a time but, doesn't work when using a 32-bit MAC. The problem is that most 32-bit MACs must be read in a single 32-bit operation and thus presents a problem when writing a 32-bit value at a 16-bit aligned address. In other words, if we read 32 bits from the MAC, we'd want to write 32 bits into the `NET_BUF` but, because the first 32 bits read contain a portion of the Ethernet header, we need to store at offset `+2` in the 'Data' portion of the `NET_BUF`.



No alignment issues with 8-bit MACs



No alignment issues with 16-bit MACs



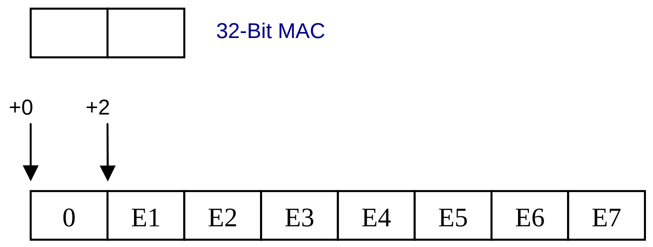
Can't move 32-bit data from MAC on 2-byte aligned!

This problem could be solved if the MAC was able to 'prepend' 2 bytes of 'stuffing' at the beginning of the packet as shown below:

MAC	**`NET_BUF`'s 'Data' offset**
`0x0000_E1E2`	`+0x00`
`0xE3E4_E5E6`	`+0x04`
`0xE7E8_E9EA`	`+0x08`
`0xEBEC_EDEE`	`+0x0C`
`0xI1I2_I3I4`	`+0x10`
`0xI5I6_I7I8`	`+0x14`
Etc.	

Where **E1** is the first Ethernet Header byte, **E2** is the second Ethernet Header byte … **I1** is the first IP Header byte, **I2** is the second IP Header byte, etc.

In other words, having the MAC stuff two bytes of `0x00` at the beginning of the packet would allow us to move 32 bits of data from the MAC starting at 'Data+0' of our `NET_BUF` and thus benefit from the performance improvement of doing 32 bit reads and writes. Of course, a similar process occurs when transmitting (the MAC needs to ignore the first TWO stuffed bytes and thus not send those out).



Stuffing the first 2 bytes of a packet

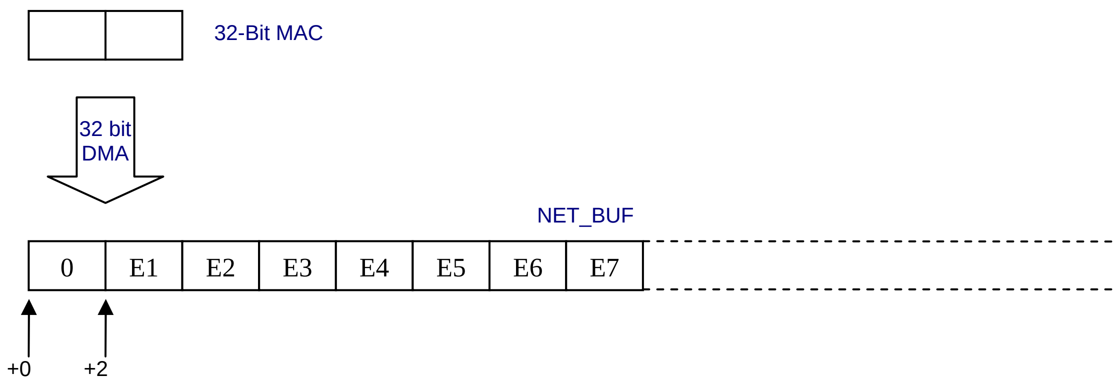
DMA driven copies

If the target used with µC/TCP-IP contains DMA controllers then we can move MAC packets to and from the NET_BUF using DMA.

32-bit MACs moving 32-bit data with 2 byte stuffing

If the 32-bit MAC adds two extra bytes at the beginning of a packet, we can move 32-bit data from the MAC directly into the NET_BUF. This is the most efficient way to move data to/from NET_BUFs. In fact, this operation can be performed as the packet is being received by the MAC and the processor can be notified only once the packet is placed in the NET_BUF. The processor can then do further processing (i.e. TCP/IP) on the packet. This method assumes that the NET_BUF can hold the largest possible packet (1536 bytes) because we don't know how big the packet will be as it's being received. However, this is generally not a problem on most 32-bit based applications.

Of course, a similar process occurs when transmitting (the MAC needs to ignore the first TWO stuffed bytes and thus not send those out).



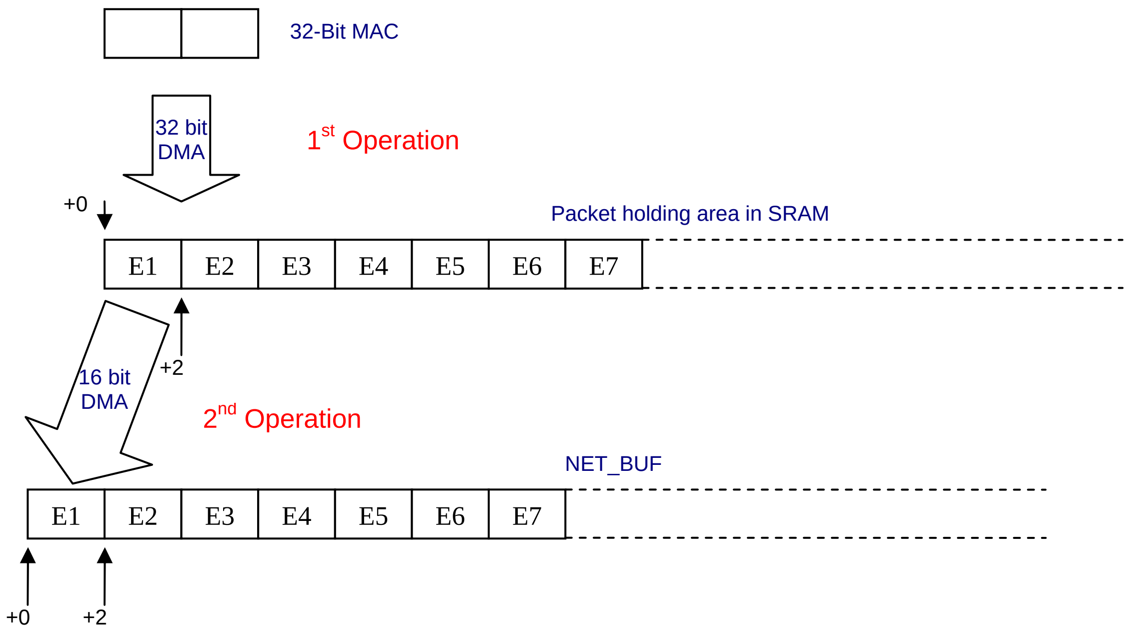
32-bit MACs using 32-Bit DMA transfers

32-bit MACs moving 32-bit data without 2 byte stuffing

If the 32-bit MAC does not add the two extra bytes at the beginning of a packet then, if we want to use a DMA controller, we can use the DMA in a two step operation:

1) Move the MAC packet into a holding area (preferably in fast SRAM) using 32-bit transfers
2) Move the holding area to the `NET_BUF` using 16-bit transfers

This method is clearly not very efficient as it means moving the data twice. Also, it assumes that we allocate sufficient storage in the holding memory to hold the largest possible packet (1536 bytes).



32-bit MACs using TWO DMA transfers

It might actually be faster to perform the second operation using the CPU especially if it supports data bursting and allows to load multiple CPU registers with a single instruction. In other words, we can load many CPU registers all at once from the holding area in SRAM and then move the data out of the CPU registers using 16-bit moves:

1) DMA the packet into high speed SRAM
2) Have the CPU loop through the received packet and load as many CPU registers as possible
3) Move the packet out of the CPU registers 16 bits at a time