Buffer Proposal

Version 1.01

Allen Baker

Hewlett-Packard Company
INSO R&D Lab

July 1, 1996

Table of Contents

1. Patents 3
2. Introduction 4
3. API 5
  3.1 Allocation and Deallocation 5
    3.1.1 CreateBuf 5
    3.1.2 DestroyBuf 5
  3.2 Manipulation 5
    3.2.1 ReinitializeBuf 5
    3.2.2 PrependCopyToBuf 5
    3.2.3 PrependBuf 6
    3.2.4 UnprependBuf 6
  3.3 Information 6
    3.3.1 BufDataSpace 6
    3.3.2 BufFreeSpace 7
    3.3.3 BufDataAddr 7
4. Reference Implementation 8
5. Example Usage 11
  5.1 Protocol Layer N 11
  5.2 Protocol Layer N-1 12
  5.3 Protocol Layer N-2 12

1. Patents

The protocol described in this document is protected under the following international patents which are owned by the Hewlett-Packard Company:

1. DE69802792D1
2. DE69802792T2
3. DE69835314D1
4. DE69835314T2
5. EP0872991A2
6. EP0872991A3
7. EP0872991B1
8. EP0886411A2
9. EP0886411A3
10. EP0976058A1
11. EP0976058B1
12. US6202096 B1
13. US6502000
14. US6721286
15. WO1998047076A1

2. Introduction

This proposal is for a buffer service for TOPS. Although this service is generally useful, its motivation is to avoid excessive data copying during packet assembly and disassembly.

As the TOPS protocols pass a send-data packet down from one protocol layer to the next, each successive layer prepends its own header to the packet. For performance reasons, at each of the successive layers we don't want to allocate a new memory buffer, copy the layer's header to the new buffer, copy the data into it, and then pass the resulting buffer down to the next layer where the process repeats. We also don't want to accumulate headers in a list as the data flows down through the layers. Accumulating headers means the final layer has to allocate a contiguous buffer into which it can assemble all the pieces prior to transmission to the network. The reverse situation - receiving data and passing it up the stack - has similar problems with stripping off headers at each successive layer.

We prefer to define a buffer mechanism that allows us to add headers at each successive layer without copying data from the previous layer. The layer that originates the send-data allocates a contiguous buffer large enough to contain the send-data and all the headers that will accumulate. It places the send-data at the end of the buffer, leaving free space at the front, and passes the buffer to the next layer. The next layer places its header at the end of the free space immediately before the send-data then passes the buffer to the next layer. This process repeats down through the TOPS stack. The buffer passes out of the final layer to the network for transmission as contiguous memory containing the send-data and headers prepended in the proper order.

The buffer service described in this proposal makes this possible while hiding the implementation details behind an easy-to-use API.

3. API

All the functions of the API create or operate upon buffer control blocks, which are named by a typedef to **Tbcb**.

3.1 Allocation and Deallocation

3.1.1 CreateBuf

Allocate an initialized buffer with the specified size. In an initialized state, the entire buffer is free space and the size of the buffer's valid data space is zero.

```
Tbcb*  CreateBuf
     (
     int  size
     )
```

All functions that operate on a buffer use the Tbcb* that CreateBuf() returns to identify the buffer upon which to operate.

If CreateBuf() is successful it returns a valid Tbcb*.

If CreateBuf() fails it returns NULL.

3.1.2 DestroyBuf

Deallocate the specified buffer.

```
void  DestroyBuf
     (
     Tbcb*  buf
     )
```

After DestroyBuf() is called, the Tbcb* that was passed to it is no longer valid.

3.2 Manipulation

3.2.1 ReinitializeBuf

Set a buffer back to an initialized state. In an initialized state, the entire buffer is free space and the size of the buffer's valid data space is zero.

```
void  ReinitializeBuf
     (
     Tbcb*  buf
     )
```

Any data that was contained in the buffer referenced by Tbcb* before the call to ReinitializeBuf() is unavailable after the call.

3.2.2 PrependCopyToBuf

Remove size bytes from the end of the buffer's free space and add them to the beginning of the buffer's valid data space. Then copy size bytes from data to the first size bytes of buffer's valid data space.

```
int  PrependCopyToBuf
     (
     Tbcb*    buf,
     void*  data,
     int    size
     )
```

If PrependCopyToBuf() is successful it returns BUFOK, the buffer's free space is reduced by size bytes, and the buffer's valid data space is increased by size bytes.

If size is larger than the buffer's available free space, PrependCopyToBuf() returns BUFERR and the buffer's free space is not modified.

3.2.3 PrependBuf

Remove size bytes from the end of the buffer's free space and add them to the beginning of the buffer's valid data space.

```
int  PrependBuf
     (
     Tbcb*    buf,
     int    size
     )
```

If PrependBuf() is successful it returns BUFOK, the buffer's free space is reduced by size bytes, and the buffer's valid data space is increased by size bytes.

If size is larger than the buffer's available free space, PrependBuf() returns BUFERR and the buffer's free space is not modified.

3.2.4 UnprependBuf

Remove size bytes from the beginning of the buffer's valid data space and add them to the end of the buffer's free space.

```
int  UnprependBuf
     (
     Tbcb*  buf,
     int  size
     )
```

If UnprependBuf() is successful it returns BUFOK, the buffer's valid data space is reduced by size bytes, and the buffer's free space is increased by size bytes.

If size is larger than the buffer's valid data space, UnprependBuf() returns BUFERR and the buffer's valid data space is not modified.

3.3 Information

3.3.1 BufDataSpace

Return the size of the buffer's valid data space. (A byte count of the amount of valid data contained in the buffer.)

```
int  BufDataSpace
     (
```

```
      Tbcb*  buf
      )
```

3.3.2 BufFreeSpace

Return the size of the buffer's free space. (A byte count of the amount of free space remaining in the buffer.)

```
int  BufFreeSpace
      (
      Tbcb*  buf
      )
```

3.3.3 BufDataAddr

Return a pointer to the beginning of the buffer's valid data.

```
void*  BufDataAddr
      (
      Tbcb*  buf
      )
```

4. Reference Implementation

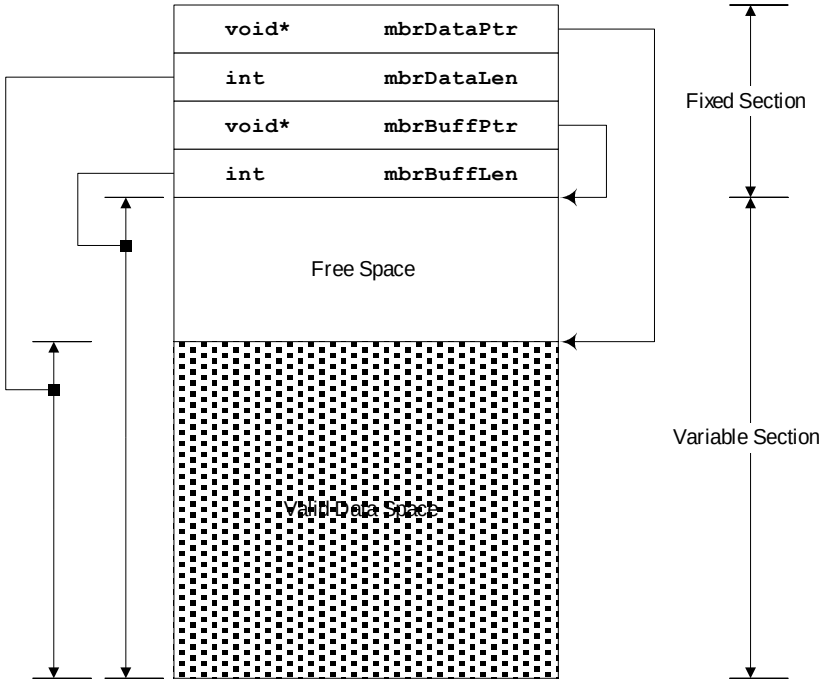


Figure 1

This reference implementation is based upon the data structure illustrated in Figure 1. Each buffer has two sections, the fixed size section and the variable size section. The fixed section is the size of one Tbcb struct and contains the buffer's state information. The variable section is the size specified in the CreateBuf() call and contains the buffer's free space / valid data space.

```
#define    BUFOK  0
#define    BUFERR 1

typedef    struct  bcb
    {
    Tu8*   mbrDataPtr;
    int    mbrDataLen;
    Tu8*   mbrBuffPtr;
    int    mbrBuffLen;
    }      Tbcb;

Tbcb*      CreateBuf
    (
    int    prmSize
    )
    {
    Tbcb*  lclTbcbPtr;

    lclTbcbPtr = (Tbcb*)malloc(sizeof(Tbcb) + prmSize);
    if (lclTbcbPtr)
           {
           lclTbcbPtr->mbrBuffPtr = ((Tu8*)lclTbcbPtr) + sizeof(Tbcb);
           lclTbcbPtr->mbrBuffLen = prmSize;
```

```
            ReinitializeBuf(lclTbcbPtr);
            }
        return(lclTbcbPtr);
        }

    void        DestroyBuf
        (
        Tbcb*  prmTbcbPtr
        )
        {
        free(prmTbcbPtr);
        }

    void        ReinitializeBuf
        (
        Tbcb*  prmTbcbPtr
        )
        {
        prmTbcbPtr->mbrDataPtr = prmTbcbPtr->mbrBuffPtr + prmTbcbPtr->mbrBuffLen;
        prmTbcbPtr->mbrDataLen = 0;
        }

    int PrependCopyToBuf
        (
        Tbcb*  prmTbcbPtr,
        void*  prmData,
        int    prmSize
        )
        {
        if (PrependBuf(prmTbcbPtr,prmSize) == BUFERR) return(BUFERR);
        memcpy(BufDataAddr(prmTbcbPtr),prmData,prmSize);
        return(BUFOK);
        }

    int PrependBuf
        (
        Tbcb*  prmTbcbPtr,
        int    prmSize
        )
        {
        if (BufFreeSpace(prmTbcbPtr) < prmSize) return(BUFERR);
        prmTbcbPtr->mbrDataPtr -= prmSize;
        prmTbcbPtr->mbrDataLen += prmSize;
        return(BUFOK);
        }

    int UnprependBuf
        (
        Tbcb*  prmTbcbPtr,
        int    prmSize
        )
        {
        if (BufDataSpace(prmTbcbPtr) < prmSize) return(BUFERR);
        prmTbcbPtr->mbrDataPtr += prmSize;
        prmTbcbPtr->mbrDataLen -= prmSize;
        return(BUFOK);
        }

    int BufDataSpace
        (
        Tbcb*  prmTbcbPtr
        )
        {
```

```
    return(prmTbcbPtr->mbrDataLen);
    }

int BufFreeSpace
    (
    Tbcb*  prmTbcbPtr
    )
    {
    return(prmTbcbPtr->mbrBuffLen - prmTbcbPtr->mbrDataLen);
    }

void*      BufDataAddr
    (
    Tbcb*  prmTbcbPtr
    )
    {
    return((void*)(prmTbcbPtr->mbrDataPtr));
    }
```

5. Example Usage

This example is just for illustrating how protocol developers can use the buffer API. It leaves out error checking and protocol details that would obscure the basic illustration.

The example is based on a three layer protocol stack. Layer N is at the top, layer N-1 is immediately below it, and layer N-2 is the bottom layer.

5.1 Protocol Layer N

This is the topmost layer in a protocol stack. It is the originator of send-data and the destination of receive-data.

For its send-data function, it allocates a buffer large enough for the send-data plus the headers that layers N-1 and N-2 will prepend. It then copies the send-data to the buffer and passes it to the N-1 layer.

```
#define  N2_HEADER_SIZE  16
#define  N1_HEADER_SIZE  8

{
:
sendTbcbPtr = CreateBuf(dataSize + N1_HEADER_SIZE + N2_HEADER_SIZE);
PrependCopyToBuf(sendTbcbPtr,sendDataPtr,dataSize);
N1Send(sendTbcbPtr);
:
}
```

For its initial receive data posting, it allocates a buffer large enough for the maximum receive-data plus the headers that layers N-1 and N-2 will need. It then passes the receive request down through the layers so that the bottom-most layer has someplace to put data that it receives.

```
{
:
receiveTbcbPtr = CreateBuf(maxSize + N1_HEADER_SIZE + N2_HEADER_SIZE);
N1Receive(receiveTbcbPtr);
:
}
```

For it receive-data callback function, it processes and removes the data it receives and then recycles the buffer for another receive.

```
NReceiveCallBack(Tbcb* receiveTbcbPtr)
       {
       dataPtr = BufDataAddr(receiveTbcbPtr);
       ProcessReceivedData(dataPtr);
       ReinitailizeBuf(receiveTbcbPtr);
       N1Receive(receiveTbcbPtr);
       }
```

At the time of program termination or whenever a buffer is no longer needed, the layer currently in possession of the buffer's Tbcb (the buffer owner) will free the buffer.

```
{
:
DestroyBuf(unneededTbcbPtr);
:
}
```

5.2 Protocol Layer N-1

This is an intermediate layer that adds some value as data passes through it.

For its send-data function it prepends its header to the send-data and passes it to the N-2 layer.

```
N1Send(Tbcb* sendTbcbPtr)
       {
       :
       PrependBuf(sendTbcbPtr,N1_HEADER_SIZE);
       headerPtr = BufDataAddr(sendTbcbPtr);
       FillInMyHeader(headerPtr);
       N2Send(sendTbcbPtr);
       :
       }
```

For its receive-data request function it simply passes the request through to the N-2 layer.

```
N1Receive(Tbcb* receiveTbcbPtr)
       {
       :
       N2Receive(receiveTbcbPtr);
       :
       }
```

For its receive data call back function it strips off its header and passes the receive data up to the N layer.

```
N1ReceiveCallBack(Tbcb* receiveTbcbPtr)
       {
       :
       UnprependBuf(receiveTbcbPtr,N1_HEADER_SIZE);
       NReceiveCallBack(receiveTbcbPtr);
       :
       }
```

5.3 Protocol Layer N-2

This is the bottom layer. Its implementation would look much like layer N-1 with the exception that it interfaces with the network (or a network service).