



JBM TCP/IP Header Processing Methods in C/C++

Header Information

JBM Electronics Co. has developed a set of headers to provide a mechanism for bridging the different characteristics between the "frame-type" data of a polled legacy protocol and the "stream-type" data of a TCP/IP connection. The headers pass information between a customer-developed application and the Gateway. The header is stripped before the data is sent to the serial device.

We provide an expanded theory of operation, examples of header (status) processing logic and example code, which can be used to as a guide when adding support for our headers to a TCP application. This code will simplify the effort necessary to support the headers. The available information on our home page is:

Headers	Headers.pdf	Smspecs.pdf
Status Message Examples	Smexex1.pdf	Smexex2.pdf
Header Example Code	This document	hdtst.zip

Background

The primary function of a JBM Electronics Gateway is to extract data from one protocol and repackage it within another. Depending on how dissimilar the capabilities and behaviors of the two protocols are, the repackaging process may exhibit some undesirable side effects.

Consider an example configuration where a stream based protocol, such as IP, is being converted to a block mode serial protocol, such as Bisync or SNA, and vice versa. Data exchanges through SNA or Bisync are structured as well defined blocks. The rules governing the block structure are strict and unforgiving, and for this reason the applications reading and writing to these kinds of interfaces can depend on a guaranteed data presentation. Data exchanges through IP, by definition, are an unstructured stream of bytes. This stream of bytes is allowed to be fragmented, by intervening routers due to buffer space constraints and/or network congestion or by the IP protocol stacks themselves, for reasons ranging from physical interface constraints, operating systems interactions and just basic design/implementation. An application has no control over these mechanisms, and for this reason it cannot assume that data will be presented in any particular manner. The net effect of all this is that data which originated in nice blocks from the SNA or Bisync device may get broken into numerous pieces, or combined into one big piece as it travels through the IP interface. From the opposite direction, data originating from the IP device may consists of one, a few or many fragments, and each fragment may or may not require reassembly into a larger block or division into smaller blocks upon delivery to the SNA or Bisync device. In fact, there's no IP based mechanism available to even make the block boundary determination.

Whether or not this poses problems on either end depends upon the end point applications receiving the data. Since there are no inherent mechanisms in either protocol to convert to and from stream and block modes, there is no generic solution to any problems which might be encountered.

We have had to deal with these very problems on several occasions. In every case, the solution has involved the addition of some 'header' information to the stream based protocol to allow applications reading and writing it to determine block boundaries. This solution requires both intelligence in the IP application and special processing in the Gateway device. The solutions have ranged from the very simple to the very complex, requiring several customized versions of the Gateway software to work with these existing IP applications. From all of these custom solutions we have also been able to derive some general forms of processing which can be configured, and the intelligence in a new IP application can be coded accordingly.

In its simplest form, the header information consists of nothing more than a simple length indication at the beginning of every 'block' that the IP application wishes to send. When the Gateway receives its initial packet from an IP device, it first parses out the length information and then waits to receive that many bytes before releasing the data to the remote interface and looking for the next length attribute. The Gateway will then read as many fragments of the data (or portions thereof) as it takes to assemble the block.

When a block is received from the remote (serial) device, it will prefix the length to the data and the IP application can use that information to decide when all of the data in the block has been received, no matter how it may have been fragmented during its journey.

More complex headers are also available which can be used to convey other information about the nature of the data, end-to-end statuses or other kinds of 'out of band' data. Detailed information on the specifications of the configurable, standard header types supported by the Gateways may be found by clicking [here](#). If you have a unique application and want to develop a new header type, contact JBM for assistance.

[Application programming implementations for Standard and Extended Headers](#)

Writing application code to deal with Standard Headers is usually a fairly straightforward matter. There are several possible approaches:

- Create a buffer sized for the largest expected block and read the stream until all the data in the block (according to the header length attribute) has been received and then release the buffer to the application. We can either set aside two buffers, one for the header and one for the data; or we may use one buffer and 'overlay' the header image onto it and pass the data to the application via a pointer to the data (thereby 'skipping' the header).
- Create a resizable ('growable') buffer class, append each fragment of the block to it until all the data in the block (according to the header length attribute) has been received and then release the data to the application.

- Create a container manager class (one which perhaps manages a buffer comprised of nodes from a linked-list) into which each fragment of the block may be received until all the data in the block (according to the header length attribute) has been received and pass a reference or pointer to the container manager to the application.

The approach chosen will depend on many factors such as the complexity of the application, the OS/API interface to the IP stack services, data structures and manipulation methods available for buffer resources, etc. But in any case, there are some common operations which must be performed, including:

- Extracting and assembling headers and packets from IP message fragments.
- Managing two modes, one where we are assembling a header and one in which we are assembling a packet.

We have provided the following examples of various methods of receiving data with header information prepended.

Method One - Separate buffers for header and data

In this method, we set aside two buffers, one for header assembly and one for data packet assembly. We use a simple state engine to manage the modes where we are processing a header versus processing a data packet.

Method Two - Separate buffers for header and data

In this method, we only set aside one buffer which will contain both the header and data packet, which assumes we can anticipate the size of the largest block we can receive. We use a simple state engine to manage the modes where we are processing a header versus processing a data packet.

Method Three - Single buffer with header 'overlay' class

In this method, we only set aside one buffer which will contain both the header and data packet. We will use an overlay message class to decompose the header. As in the first method we will use a simple state engine to manage the modes where we are processing a header versus processing a data packet. Two examples have been provided, one using the standard headers and one with the extended headers.

Example Code

The example code has been written in C/C++ for a Linux environment using a BSD sockets style interface to the TCP/IP stack layer. The code runs from console/command line interface, TTY or Telnet. The code may be obtained in compressed form by clicking [here](#).

The archive should contain the following files:

File Name	Description	Required by 1 2 3 4
ex1.hpp	Header file for Example 1.	Y N N N
ex2.hpp	Header file for Example 2.	N Y N N
ex3.hpp	Header file for Example 3.	N N Y N
ex4.hpp	Header file for Example 4.	N N N Y
jbmhdef.h	Definitions of static values declared in jbmhdrs.hpp.	N N Y Y
jbmhdrs.hpp	Header file for JBM Header 'overlay' class.	N N Y Y
jbmhdrs.hpp	Inlinable code for jbm Header 'overlay class.	N N Y Y
lnxtcpdf.hpp	Linux specific TCP/IP interface Header file.	Y Y Y Y
ex1.cpp	Source code for example 1.	Y N N N
ex2.cpp	Source code for example 2.	N Y N N
ex3.cpp	Source code for example 3.	N N Y N
ex4.cpp	Source code for example 4.	N N N Y
jbmhdrs.cpp	Source code for jbm Header 'overlay' class.	N N Y Y
filelst1.txt	File list for example 1 (used by make file).	Y N N N
filelst2.txt	File list for example 2 (used by make file).	N Y N N
filelst3.txt	File list for example 3 (used by make file).	N N Y N
filelst4.txt	File list for example 4 (used by make file).	N N N Y
makfile1	Make file for example 1.	Y N N N
makfile2	Make file for example 2.	N Y N N
makfile3	Make file for example 3.	N N Y N
makfile4	Make file for example 4.	N N N Y

Each example has been provide with its own make file, for example, to build example 1, you would type:

```
make -f makfile1
```

Which would produce an executable named ex1.

Example One requires two command line parameters, the IP address and IP port number of a host acting as a server transmitting data using jbm standard headers. As an example:

```
ex1 192.168.10.24 4433
```

Examples Two - Four require three command line parameters, the IP address and IP port number of a host acting as a server transmitting data using JBM standard headers (or extended headers in the case of example 4) and the maximum expected block size; however, if the block size is omitted, the program assumes a value of 512. As an example:

```
ex3 192.168.10.33 10010 420
```

Of course these examples are not intended to be used as the basis for a complete TCP/IP implementation. The code fails to properly handle with all of the possible IP related error conditions, which might occur. All of that would have tended to clutter the basic idea here, which was to illustrate the mechanics of assembling headers and data blocks from the dribbles and drabs of an IP stream.

The code was kept deliberately simple and should be pretty self explanatory as-is; however, feel free to contact us if you have questions or comments about the code or problems with building or running the examples.