Network Working Group
Request for Comments: 914

David J. Farber
Gary S. Delp
Thomas M. Conte
University of Delaware
September 1984

# A Thinwire Protocol for connecting personal computers to the INTERNET

## Status of this Memo

This RFC focuses discussion on the particular problems in the ARPA-Internet of low speed network interconnection with personal computers, and possible methods of solution. None of the proposed solutions in this document are intended as standards for the ARPA-Internet. Rather, it is hoped that a general consensus will emerge as to the appropriate solution to the problems, leading eventually to the adoption of standards. Distribution of this memo unlimited.

## What is the Problem Anyway ?

As we connect workstations and personal computers to the INTERNET, many of the cost/speed communication tradeoffs change. This has made us reconsider the way we juggle the protocol and hardware design tradeoffs. With substantial computing power available in the $3--10K range, it is feasible to locate computers at their point of use, including in buildings, in our homes, and other places remote from the existing high speed connections. Dedicated 56k baud lines are costly, have limited availability, and long lead time for installation. High speed LAN's are not an applicable interconnection solution. These two facts ensure that readily available 1200 / 2400 baud phone modems over dialed or leased telephone lines will be an important part of the interconnection scheme in the near future. This paper will consider some of the problems and possibilities involved with using a "thin" (less than 9600 baud) data path. A trio of "THINWIRE" protocols for connecting a personal computer to the INTERNET are presented for discussion.

Although the cost and flexibility of telephone modems is very attractive, their low speed produces some major problems. As an example, a minimum TCP/IP Telnet packet (one character) is 41 bytes long. At 1200 baud, the transmission time for such a packet would be around 0.3 seconds. This is equivalent to using a 30 baud line for single character transmission. (Throughout the paper, the assumption is made that the transmission speed is limited only by the speed of the communication line. We also assume that the line will act as a synchronous link when calculating speed. In reality, with interrupt, computational, and framing overhead, the times could be 10-50% worse.)

In many cases, local echo and line editing can allow acceptable

Telnet behavior, but many applications will work only with character at a time transmission. In addition, multiple data streams can be very useful for fully taking advantage of the personal computer/Internet link. Thus this proposal.

There are several forms that a solution to this problem can take. Three of these are listed below, followed by descriptions of possible solutions of each form.

- As a non-solution, one can learn to live with the slow communication (possibly a reasonable thing to do for background file transfer and one-time inquiries to time, date, or quote-of-the-day servers).

- Using TCP/IP, one can intercept the link level transmissions, and try various kinds of compression algorithms. This provides for a symmetrical structure on either side of the "Thinwire".

- One could build an "asymmetrical" gateway which takes some of the transport and network communication overhead away from both the serial link and the personal computer. The object would be to make the PC do the local work, and to make the interconnection with the extended network a benefit to the PC and not a drain on the facilities of the PC.

The first form has the advantage of simplicity and ease of implementation. The disadvantages have been discussed above. The second form, compression at link level, can be exploited in two ways.

Thinwire I is a simple robust compressor, which will reduce the 41 byte minimum TCP/IP Telnet packets to a series of 17 byte update packets. This would improve the effective baud rate from 30 baud to 70 baud over a 1200 baud line (for single character packets).

Thinwire II uses a considerably more complex technique, and takes advantage of the storage and processing power on either side of the thinwire link. Thinwire II will compress packets from multiple TCP/IP connections from 41 bytes down to 13 bytes. The increased communication rate is 95 (effective) baud for single character packets.

The third form balances the characteristics of the personal computer, the communications line, the gateway, and the Internet protocols to optimize the utility of the communications and the workstation itself. Instead of running full transport and internet layers on the PC, the PC and the gateway manage a single reliable stream, multiplexing data on this stream with control requests. Without the interneting and flow control structures traveling over the communications line on a per/packet basis, the data flow can be

compressed a great deal. As there is some switching overhead, and a reliable link level protocol is needed on the serial line, the average effective baud rate would be in the 900 baud range.

Each of these Thinwire possibilities will be explored in detail.

## Thinwire I

The simplest technique for the compression of packets which have similar headers is for both the transmitting and receiving host to store the most recent packet and transmit just the changes from one packet to the next. The updated information is transmitted by sending a packet including the updated information along with a description of where the information should be placed. A series of descriptor-data blocks would make up the update packet. The descriptor consists of the offset from the last byte changed to the start of the data to be changed and a count of the number of data bytes to be substituted into the old template. The descriptor is one byte long, with two four bit fields; offsets and counts of up to 15 bytes can be described. In the most pathological case the descriptor adds an extra byte for every 15 bytes (or a 6% expansion).

An example of Thinwire I in action is shown in Appendix A. A sequence of two single character TCP/IP Telnet packets is shown. The "update" packet which would actually be transmitted is shown following them. Each Telnet packet is 41 bytes long; the typical update is 17 bytes. This technique is a useful improvement over sending entire packets. It is also computationally simple. It suffers from two problems: the compression is modest, and, if there is more than one class of packets being handled, the assumption of common header information breaks down, causing the compression of each class to suffer.

## Thinwire II

Both of the problems described above suggest that a more computationally complex protocol may be appropriate. Any major improvement in data compression must depend on knowledge of the protocols being used. Thinwire II uses this knowledge to accomplish two things. First, the packets are sorted into classes. The packets from each TCP connection using the thinwire link, would, because of their header similarities, make up a class of packets. Recognizing these classes and sorting by them is called "matching templates". Second, knowledge of the protocols is used to compress the updates. A bitfield indicating which fields in the header have changed, followed only by the changed fields, is much shorter than the general form of change notices. Simple arithmetic is allowed, so 32 bit

fields can often be updated in 8 or 16 bits. By using the sorting, protocol-specific updating, Thinwire II provides significant compression.

A typical transaction is described in Appendix B. The "template matching" is based on the unchanging fields in each class of packet. A TCP/IP packet would match on the following fields: network type field(IP), version, type of service, protocol(TCP), and source and destination address and port. Note that the 41 bytes have been reduced to 13 bytes. An additional advantage is that multiple classes of packets can be transported across the same line without affecting the compression of each other, just by matching and storing multiple templates.

Some of the implications of this system are:

- The necessity of saving several templates (one for each TCP/IP connection ) means that there will be a relatively large memory requirement. This requirement for current personal computers is reasonable. In addition, the gateway must keep tables for several connections at a time.

- The Thinwire links are slow (that's why we call them thin); much slower than normal disk access. There is no reason that inactive templates cannot be swapped out to disk and retrieved when needed if memory is limited. (Note that as memory density increases, this is less and less of a problem.)

- There is state information in the connections. If the two sides get out of synchronization with each other, data flow stops. This means that some method of error detection and recovery must be provided.

- To minimize the problem described above, the protocol used on the serial line must be reliable. See Appendix D for details of SLIP, Serial Line Interface Protocol, as an example of such a protocol. There must also be periodic resynchronization. (For example, every Nth packet would be transmitted in full).

- The asynchronous link is not, by its nature, a packet oriented system; a packet structure will need to be layered on the character at a time transfer. However, if the protocol layer below thinwire (SLIP) can be trusted, the formation of packets is a simple matter.

- Thinwire II will need to be enhanced for each new protocol

(TCP, UDP, TP4) it is called upon to service. Any packet type not recognized by the Thinwire connection will be transmitted in full.

For maintaining full network service, Thinwire II or a close variant seems to be the solution.

Thinwire III

When transmissions at the local network (link) level are not required, if only the available services are desired, then a solution based on Thinwire III may be appropriate. A star network with a gateway in the center serving as the connection between a number of Personal Computers and the Internet is the key of Thinwire III. Rather than providing connections at the network/link level, Thinwire III assumes that there is a reliable serial link (SLIP or equivalent) beneath it and that the workstation/personal computer has better things to do than manage TCP state tables, timeouts, etc. It also assumes that the gateway supporting the Thinwire III connections is powerful enough to run many TCP connections and several SLIP's at the same time. The gateway fills in for the limitations of the communications line and the personal computer. It provides a gateway to the INTERNET, managing the transport and network functions, providing both reliable stream and datagram service.

In Thinwire III, the gateway starts an interpreter for each SLIP connection from a personal computer. The gateway will open TCP, UDP, and later TP4 connections on the request of the personal computer. Acting as the agent for the personal computer, it will manage the remote negotiations and the data flow to and from the personal computer. Multiple connections can be opened, with inline logical switches in the reliable data flow indicating which connection the data is destined for. Additional escaped sequences will send error and informational data between the two Thinwire III communicators.

This protocol is not symmetric. The gateway will open connections to the INTERNET world as an agent for the personal computer, but the gateway will not be able to open inbound connections to the personal computer, as the personal computer is perceived as a stub host. The personal computer may however passively open connections on the gateway to act as a server. Extended control sequences are specified to handle the multiple connection negotiation that this server ability will entail.

This protocol seems to ignore the problem of flow control. Our thought is that the processing on either side of the communication link will be much speedier than the link itself. The buffering for the communication line and the user process blocking for this will

provide most of the flow control. For the rare instances that this is not sufficient, there are control messages to delay the flow to a port or all data flow.

A tentative specification for Thinwire III is attached as Appendix C.

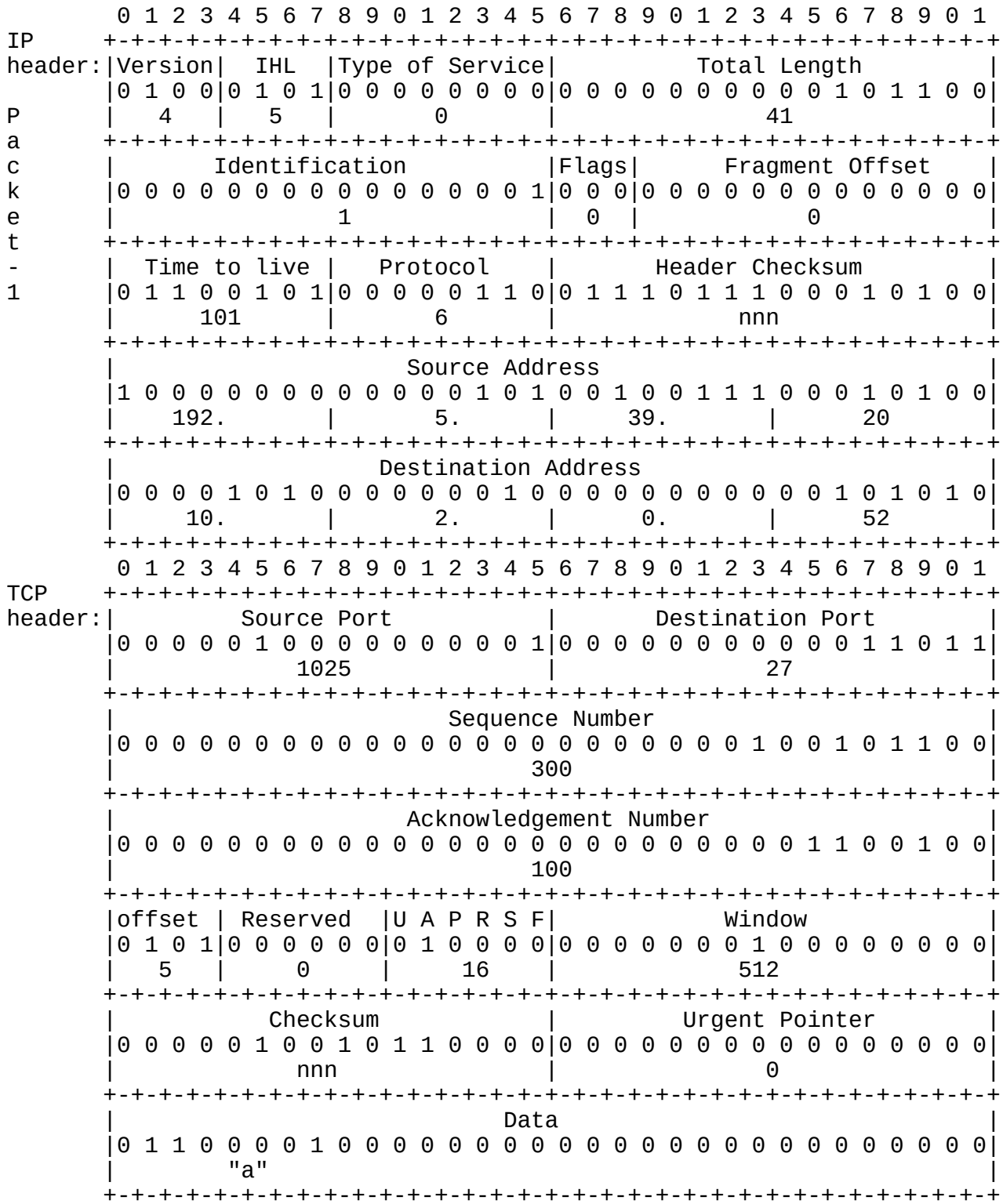
The authors acknowledge the shoulders upon which they stand, and apologize for the toes they step on. Ongoing work is being done by Eric Thayer, Guru Parulkar, and John Jaggers. Special thanks are extended to Peter vonGlahn, Jon Postel and Helen Delp for their helpful comments on earlier drafts. Responses will be greatly appreciated at the following addresses:

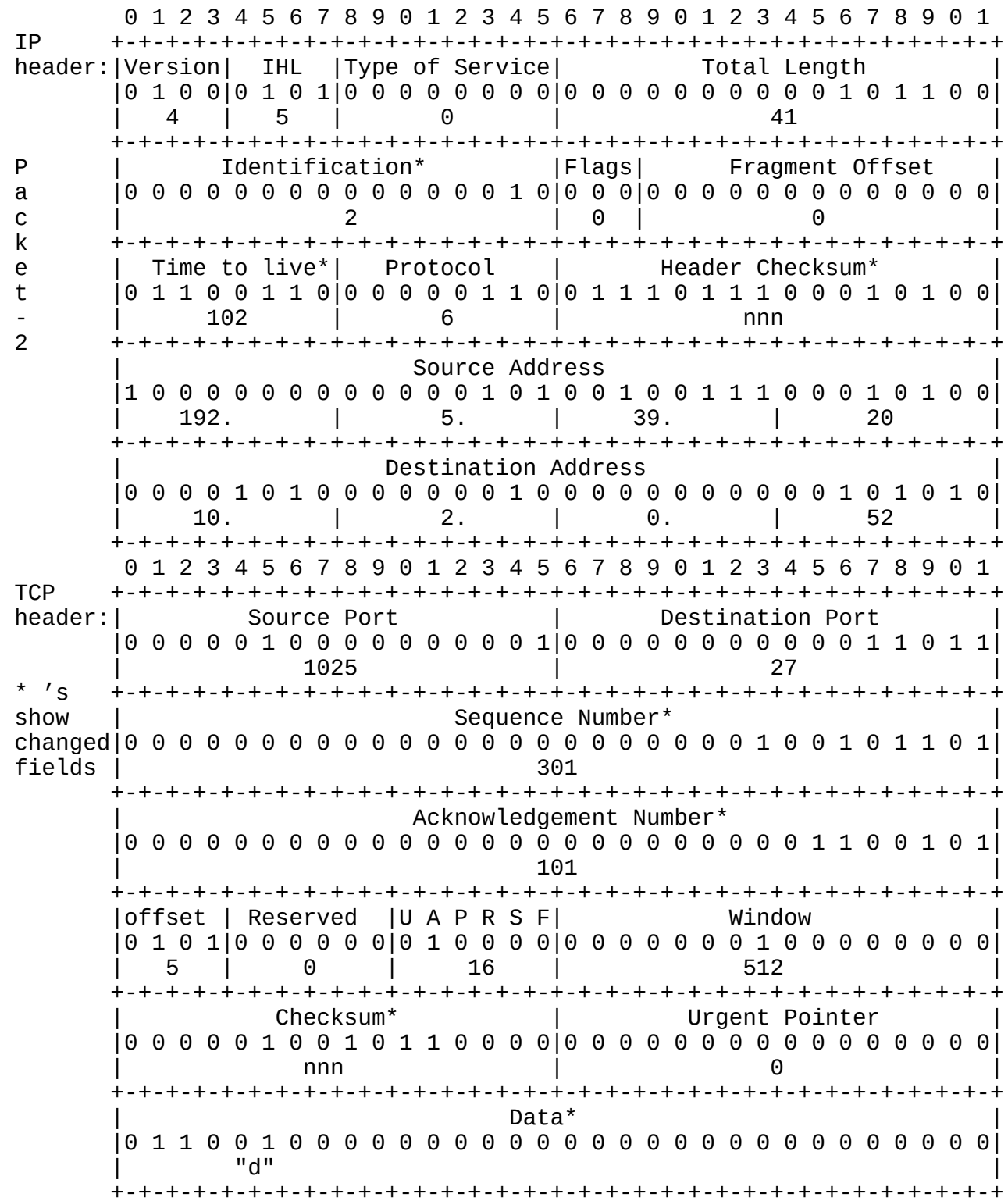
Dave Farber <Farber@udel-ee>
Gary Delp <Delp@udel-ee>
Tom Conte <Conte@udel-ee>

## Appendix A -- Example of Thinwire I Compression

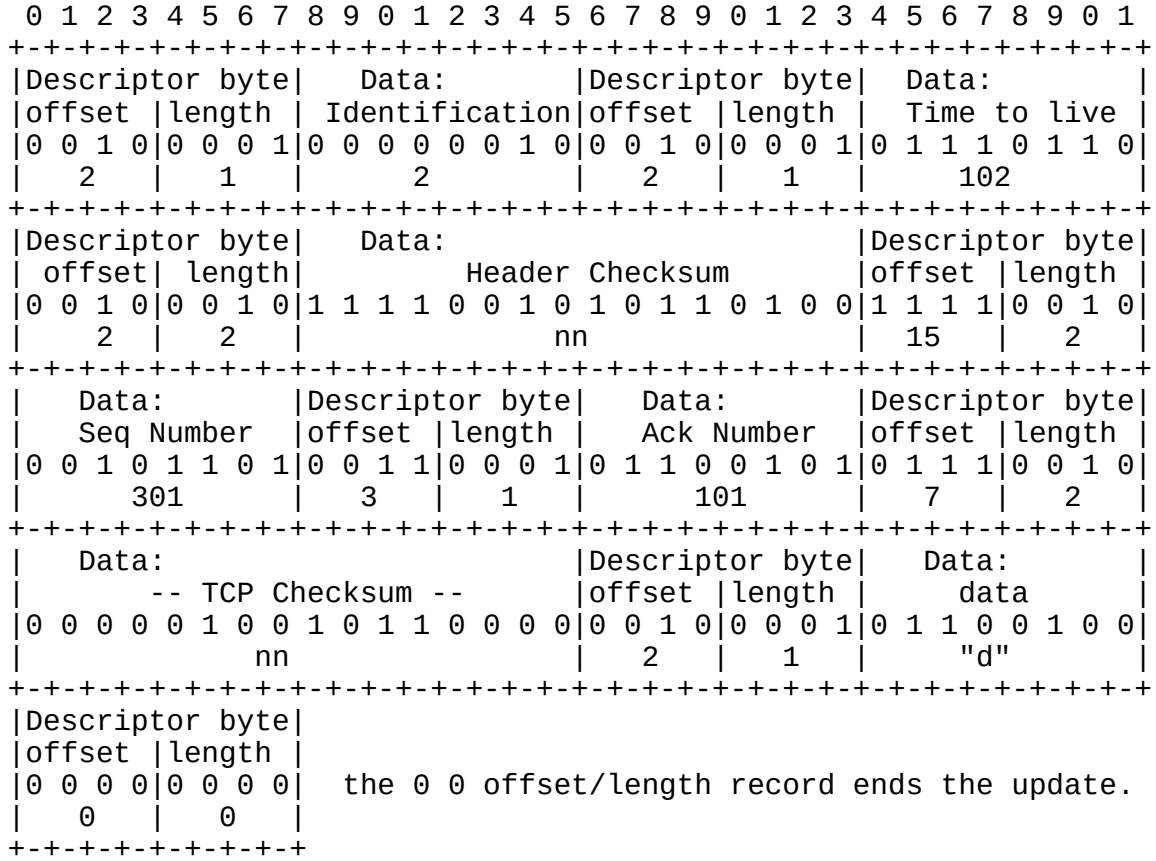
Here is an example of how Thinwire I would operate in a common situation. The connection is a TCP/IP Telnet connection. The first TCP/IP Telnet packet is on the next page; it simulates the typing of the character "a". The second packet would be produced by typing "d"; it is shown on the following page. The compressed version is on the third page following.

[NOTE: The checksums pictured have not been calculated. Binary in MSB to LSB format]


IP header:
Version
IHL
Type of Service
Total Length
4
5
0
41
Identification
Flags
Fragment Offset
1
0
0
Time to live
Protocol
Header Checksum
101
6
nnn
Source Address
192.
5.
39.
20
Destination Address
10.
2.
0.
52
TCP header:
Source Port
Destination Port
1025
27
Sequence Number
300
Acknowledgement Number
100
offset
Reserved
Window
5
0
16
512
Checksum
Urgent Pointer
nnn
0
Data
"a"


IP header:
Version
IHL
Type of Service
Total Length
Identification*
Flags
Fragment Offset
Time to live*
Protocol
Header Checksum*
Source Address
Destination Address
Packet-2
TCP header:
Source Port
Destination Port
Sequence Number*
Acknowledgement Number*
offset
Reserved
Window
Checksum*
Urgent Pointer
Data*
"d"
* 's show changed fields

The Thinwire Driver finds the template (which is the previous packet sent), compares the template to the packet and creates a change message (field names of change record data have been added for comparison):

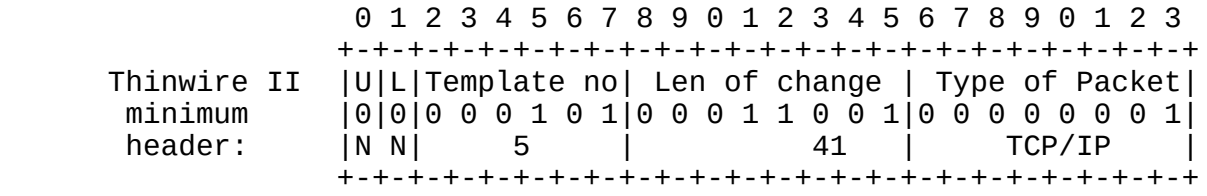



Thinwire I then sends this message over the line where the previous packet is updated to form the new packet. Note: One can see that a series of null descriptor bytes will reset the connection.

## Appendix B -- Examples of Thinwire II Compression

This Appendix provides an example of how the Thinwire II would operate in a common situation. The same original packets are used as in Appendix A, so only the updates are shown.

As the later field definitions depend on the contents of earlier fields, a field by field analysis of the update packets will be useful.




The first bit is the UPDATE bit. If it is a 0 this packet describes a new template, and the entire new packet is included, following the header. If there was a previous template with the same number, it will be cleared and replaced by the new template. If the UPDATE bit is a 1, then this packet should be used to update the template with the number given in the template number field.

The second bit is the LONG bit. If it is a 1 it indicates a LONG packet. This means that the update length field will be 16 bits instead of 8 bits.

The remaining 6 bits in the first byte indicate the template number that this packet is an update to.

The template number is followed by 1 or 2 bytes (depending on the value of the LONG bit) which give the length of the packet. This is the number of data bytes following the variable length header.

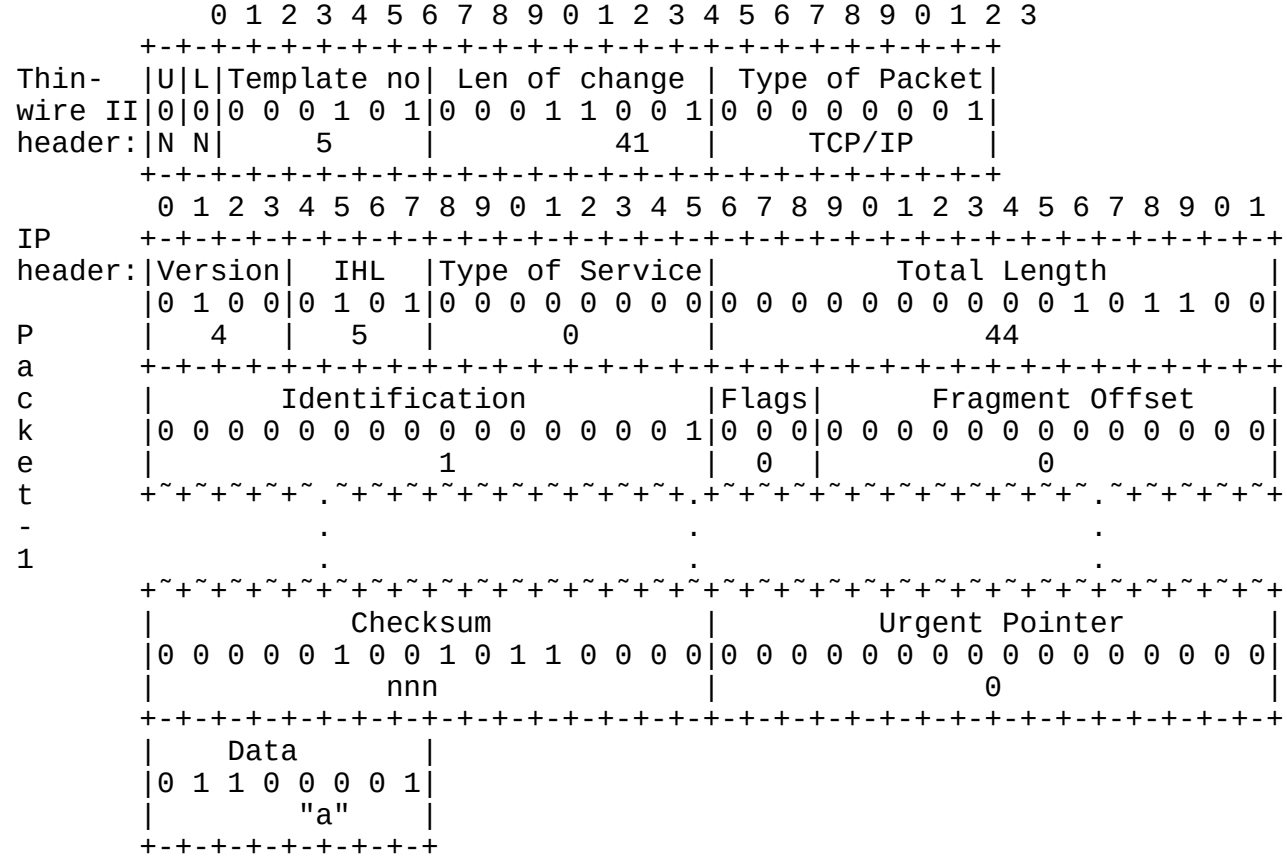
If the UPDATE bit is 0 on this packet, the next byte will be a flag telling what type of packet the sender thinks this packet is. The flag will be saved by the receiver to interpret the update packets. Type 0 is for unknown types. If the type 0 flag is set, there will be no updates to this template number. Type 1 is TCP/IP; the method of updating will be described below. Type 2 is UDP/IP; the method of update is not described at this time.

At this time we have enough information to encode packet 1 of the example. Assuming for the moment that this is the first packet for this connection, the UPDATE bit would be set to 0. As the packet has a length of 41 and so can be described in 8 bits, the LONG bit would be set to 0. A template number not in use (or the oldest in use

template number) would be assigned to this packet. The number 5 is illustrated. The Length of Packet would be given as 41, and the Type Flag set to TCP/IP (1). The 41 bytes of the packet would follow.

The transmission of packet 2 requires the specification of Type 1 (TCP/IP) updating. There are portions of the packets which will always be the same; these are described in the body of the paper, and are used to match the template. These do not need to be transmitted for an update. There are portions of the packet which will always (well almost always) change. These are the IP Header checksum, the IP Identification number, and the TCP checksum. These are transmitted, in that order, with each template update immediately after the packet length byte/bytes. Following the invariant portion of the header are updates to the fields which change some of the time. Which fields are different is indicated with a bitfield describing the changes.

The Bitfield is used to indicate which fields (of those that may stay the same) have changed. The technique for updating the field varies with the field description. The specifications for TCP/IP are shown in Table B-1.

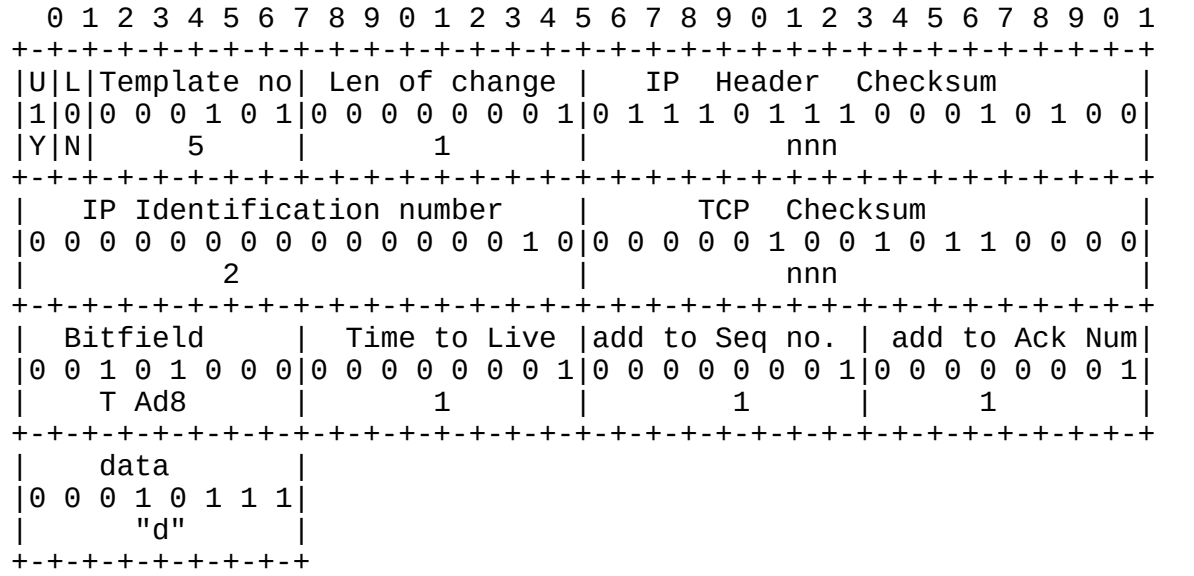


The changed field update information is added to the update header in the order that the bits appear in the field. That is, if both the IP packet length bit and the Time to Live bit are set, the 2 new bytes of the IP Packet length will precede the 1 new byte of the Time to Live field.

The update for packet 2 is shown below. Note that this is an update to template 5, the length of update is 8 bits with a value of 1. The new checksums and IP Identification Number are included, and the flags are set to indicate changes to the following fields: Time to Live, Add 8 bits to Sequence and Acknowledgement Numbers. The new data is one byte following the header.

Thinwire II would send this message over the line where it would be reassembled into the correct packet.

Note: For purposes of synchronization, if three 0 length, template 0, type 0 packets are received, the next non-zero byte should be treated as a start of packet, and the template tables cleared.




Packet 2. Thinwire II update

## Appendix C -- Tentative Specification for Thinwire III

Thinwire III, as stated in the body of this paper, provides multiple virtual connections over a single physical connection. As Thinwire III is based on a single point to point connection, much of the per/datagram information (routing and sequencing) of other transport systems can be eliminated. In the steady state any bytes received by thinwire III are sent to the default higher level protocol connection. There are escaped control sequences which allow the creation of additional connections, the switching of the default connection, the packetizing of datagrams, and the passing of information between the gateway and the personal computer. The gateway and the personal computer manage a single full duplex stream, multiplexing control requests and streams of data through the use of embedded logical switches.

The ascii character "z" (binary 01011011 ) is used as the escape character. The byte following the "z" is interpreted to determine the command. Table C-1 shows the general classes the bytes (Request codes) can fall into.

In order to transmit the character "z", two "z"'s are transmitted. The first is interpreted as an escape, the second as the lower case letter "z" to be transmitted to the default connection. The letter z was chosen as the escape for its low occurrence in text and control data streams, because it should pass easily through any lower level protocols, and for its generally innocuous behavior.

Descriptions of specifications of each of the Request codes are below.

Starting with the range 0-31; these Request codes change the default connection. After a connection has been established, any characters which come across the line that are not part of a Request code sequence are transmitted to one of the connections. To begin with this connection defaults to Zero, but when the "Switch Default Connection" command is received, characters are sent to the connection named in the request until a new request is received. Zero is a special diagnostic connection; anything received on connection number Zero should be echoed back to the sender on connection number One. Anything received on connection number One should be placed on the diagnostic output of the receiving host. Any other connection number indicates data which should be sent out the numbered connection. If the numbered connection has not been opened, the data can be thrown away, and an Error Control Message returned to the sender.

Escapes followed by numbers 32 through 255 are for new connections, requests for information, and error messages. The escape will be

followed by a Request code, a one byte Request Sequence Number (so that the Reply to Request can be asynchronously associated with the Request), and the arguments for the specific request. (The length of the argument field will be determined by the Request code.) The format of the request will be as pictured below:

"z" <Request Code> <Request Sequence Number> [ <Arguments> ... ]

At this time the Request codes 32-63 are reserved.

The Request codes 64-127 are for stream server open requests. For the purposes of compression, many of the common servers are assigned single byte codes. See Table C-2.

Request code 68 is to a connection to the default hostname server used by the gateway. It takes 3 bytes for this request. It has the form:

"z" < 68 > < Request Sequence Number >

Request code 95 is to open any specified TCP Port at the specified address. It takes 9 bytes for this request. It has the form:

"z" < 95 > < Request Sequence Number > < 4 bytes of IP address> < 2 bytes of TCP Port >

Request codes 96-127 are RESERVED for alternate transport protocols.

The Request codes 128-191 are used for framing Datagrams and opening new Datagram connections. The code 128 is the Start of Datagram code. The format is:

"z" <128> <Length of Datagram (2 bytes)> <Socket> Data ...

As with the Stream opens, there are a number of assigned ports with codes for them. They are listed in Table C-3.

The Request Codes 192-254 are control, status and informational requests. These are still under development, but will include:

-flow control
-get host/server/protocol by entry/name/number.
-additional error messages
-overall reset
-open passive connection

The Request Code 252 is the request to close a connection. This Code, followed by the connection number, indicates that no more data

will be sent out this connection number. A second request with the same connection number will indicate that no more data will be accepted on this connection.

The Request Code 253 is the information request for a connection. The protocol, status, and port number of the connection should be returned. The format of this reply has yet to be specified.

The Request code 254 is an error notification. These are to be acknowledged with their Request Sequence Numbers. Error codes are under development.

The Request code 255 is the Reply to Request. The Request Sequence Number identifies the request being replied to. The format is:

> "z" <255> <Request Sequence Number (in reply to)> <Length of reply (1 byte)> Reply...

The Thinwire Drivers on each side will wait at their inbound sockets, and relay across the thinwire link character-by-character/packet-by-packet for the stream/datagram connections.

Thinwire III is labeled as a tentative specification, because at this time, in order to publish this RFC in a timely fashion, several minor issues are still unresolved. An example is the scheduling of serial line use. Short messages could be given priority over long packets, or priority schemes could be changed during the session, depending upon the interactive desire of the user. Addition issues will be resolved in the future.

## Appendix D -- Serial Line Interface Protocol (SLIP)

### Initial Specifications and Implementation Suggestions

#### PHILOSOPHY

The world is a dangerous place for bits. Data transmission can be an time consuming business when one has to make sure that bits don't get lost, destroyed, or forgotten. To reduce such problems, the Serial Line Interface Protocol (SLIP) maintains an attitude toward the world that includes both a mistrust of serial lines and a margin of laziness. Examples of this approach include how SLIP recovers from errors and how SLIP handles the problem of resequencing (see PROTOCOL SPECIFICATIONS and IMPLEMENTATION SUGGESTIONS).

#### THE MESSAGE FORMAT

Both the Sender Task and the Receiver Task communicate using a standard message format and the Sender and Receiver Task of one machine's SLIP communicate using a shared buffer. The message begins with a 1 byte Start of Header token (StH, 11111111) and is followed by a sequence number of four bits (SEQ) and an acknowledgement number of four bits (ACK). Following the StH, SEQ and ACK, is a 5 bit length field which specifies the length of the data contained in the message. Following the length is a three bit field of flags. The first bit is used to indicate that the a receive error has occurred, and the ACK is actually a repeat of the Last Acknowledged message (a LACK). The second bit is used to indicate a Synchronize Sequence Numbers message (SSNM), and the third bit is used to indicate a Start of Control Message (SOCM); all three of these flags are explained below. Finally, at the end of the message is an exclusive-or checksum. The message format is shown in figure D-1.

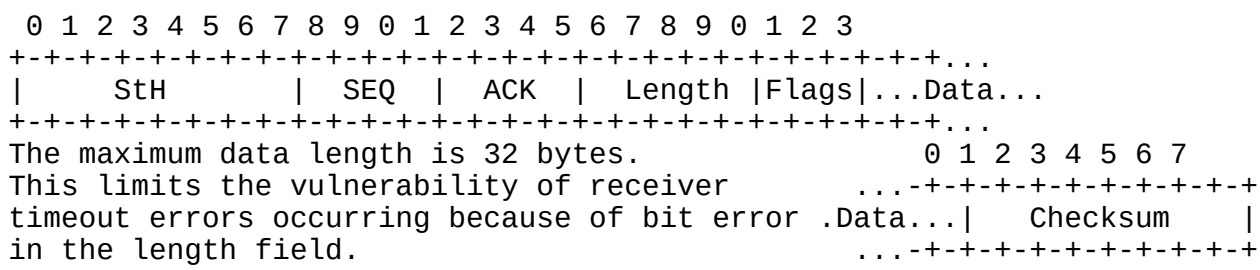



Figure D-1. SLIP Message Format

The Sender, when idle but needing to acknowledge, will send out short messages of the same format as a regular message but with the SOCM flag set and the data field omitted. ( This short message is called a SOCM, and is used instead of a zero length message to avoid the problem of continually ACK'ing ACK's ). The Sender Task, when originating a connection (see STARTING UP AND FINISHING OFF COMMUNICATIONS), will send out another short message but with the SSNM flag set and the data omitted. This message (a SSNM) used for a TCP-style 3 way startup handshake.

PROTOCOL SPECIFICATIONS and SUGGESTIONS

The SLIP module, when called with data to send, prepends its header (SEE ABOVE) to the data, calculates a checksum and appends the checksum at the end. (This creates a message.) The message has a sequence number associated with it which represents the position of the message in the Sender SLIP's buffers. The sequence number for the message can range from 0 to 15 and is returned in the ACK field of the other machine's Sender SLIP messages to acknowledge receipt.

There are two scenarios for transmission. In the first, both SLIP's will be transmitting to each other. To send an acknowledgement, the Receiver SLIP uses the ACK field in its next outgoing message. To receive an acknowledgement, the Sender checks the ACK field of its Receiver's incoming messages. In the second scenario, one SLIP may have no data to transmit for a long time. Then, as stated above, to acknowledge a received message, the Receiver has its Sender send out a short message, the SOCM (SEE ABOVE) which specifies the message it is acknowledging. The SOCM includes a checksum of its total contents. If there is a checksum error, THE SOCM IS IGNORED.

When there is a checksum error on a received normal message, the Receiver asks its Sender to send out a SOCM with the LACK flag set, or set the LACK flag on its next message. The Sender sends this flag ONCE then ceases to increment the acknowledgement number (the ACK) while the Receiver continues to check incoming messages for the sequence number of the message with a checksum error. (Note that it continues to react to the acknowledgement field in the incoming messages.) When it finds the needed message, it resumes accepting the data in new messages and increments the acknowledgement number transmitted accordingly.

The sending SLIP must never send a message greater than four past the last message for which it has received an acknowledgement (effectively a window size of four). Under normal processing loads, a window size greater than four should not be needed, and this decreases the probability of random errors creating valid

acknowledgement or sequence numbers. If the Sender has four unacknowledged messages outstanding, it will retransmit the old messages, starting from the oldest unacknowledged message. If it receives an acknowledgement with the LACK flag set, it transmits the message following the LACK number and continues to transmit the messages from that one on. Thus a LACK is a message asking the Sender to please the Receiver. If the Sender times out on any message not logically greater than four past the last acknowledged message, it should retransmit the message that timed out and then continues to transmit messages following the timed out message.

The following describes a partial implementation of SLIP. System dependent subjects like buffer management, timer handling and calling conventions are discussed.

The SLIP implementation is subdivided into four modules and two sets of input/output interfaces. The four modules are: The Sender Task, The Receiver Task, the buffer Manager, and SLIPTIME (the timer). The two interfaces are to the higher protocol and to the lower protocol (the UARTian, an interrupt driven device driver for the serial lines).

## OPERATIONS OF THE SENDER TASK

The Sender Task takes a relatively noncomplex approach to transmitting. It sends message zero, sets a timer (using the SLIPTIME Task) on the message, and proceeds to send and set timers for messages one, two, and three. When the Receiver Task tells the Sender Task that a message has been acknowledged, the Sender Task then clears the timer for that message, and marks it acknowledged. When the Sender Task has finished sending a message, it checks several conditions to decide what to do next. It first checks to see if a LACK has been received. If it has then it clears all the timers, and begins retransmitting messages (updating the acknowledgement field and checksum) starting from the one after the LACK'ed message. If there is not a LACK waiting for the Sender Task, it checks to see if any messages have timed out. If a message has timed out, the Sender Task again will clear the timers and begin retransmitting from the message number which timed out. If neither of these conditions are true, the Sender Task checks to see if, because it has looped back to retransmit, it has any previously formulated messages to send. If so, it send the first of these messages. If it does not have previously formulated messages, it checks to see if it is more than three past the last acknowledged message. If so, it restarts from the message after the last acknowledged message. If none of these are true, then it checks to see if there is more data waiting to be transmitted. If there is more data available, it forms the largest packet it can, and begins to transmit it. If there is no

more data to transmit, it checks to see if it needs to acknowledge a message received from the other side. If so then it sends a SOCM. If none of the above conditions create work for the Sender Task, the task suspends itself.

Note that the Sender Task uses the Receiver Task to find out about acknowledgements and the Receiver Task uses the Sender Task to send acknowledgements to the other SLIP on the other side (via the ACK field in the Sender Task's message). The two tasks on one machine communicate through a small buffer. Because acknowledgements need to be passed back to the Sender Task quickly, the Receiver Task can wake up the Sender Task (unblock it).

OPERATIONS OF THE RECEIVER TASK

The Receiver Task checks the checksums of the messages coming into it. When it gets a checksum error, it tells the Sender Task to mark the next acknowledgement as a LACK. It then throws away all messages coming into it that don't match the message it wants and continues to acknowledge with the last ACK until it gets the message it wants. As a checksum error could be the result of a crashed packet, and the StH character can occur within the packet, when a checksum error does occur, the recovery includes scanning forward from the last StH character for the next StH character then attempting to verify a packet beginning from it. A valid message includes a valid checksum, and sequence and acknowledgement numbers within the active window of numbers. This eliminates the need for the resequencing of messages, because the Receiver Task throws away anything that would make information in its buffers out of sequence.

OPERATIONS OF SLIPTIME

The timer task will maintain and update a table of timers for each request. Its functions should be called with the timer length and the sequence number to associate with the timer. Its functions can also be called with a request to delete a timer. An interrupt-driven mechanism is used to update the running timers and to wake up the Sender when an alarm goes off.

## THE INPUT AND OUTPUT INTERFACES

To force SLIP to do something, the higher protocol should create a buffer and then call SLIP, passing it a pointer to the buffer. SLIP will then read the buffer and begin sending it. The call to SLIP will return the number of bytes written, negative number indicates to the caller that SLIP could not do the request. Exact error numbers will be assigned in the future. To ask SLIP to receive something, one would call SLIP and SLIP would immediately return the number of bytes received or a negative number for an error (nothing ready to receive, for example).

SLIP, when it wants to talk to the underworld of the serial interface, will do much the same thing only through a buffer written to by the UARTian (for received data) and read from by the UARTian (for sent data).

## OPERATIONS OF THE BUFFER/WINDOW MANAGER

The Manager tends a continuous, circular buffer for the Sender Task in which data to be sent (from the downcalling protocol) is stored. This buffer is called the INPUT-DATA BUFFER (IDBuff). The Manager also manages a SENDER TASK'S OUTPUT-DATA BUFFER (SODBuff), which is its output buffer to the UARTian.

The IDBuff has associated with it some parameters. These parameters include: START OF MEMORY (SOM), the start of memory reserved for the IDBuff; END OF MEMORY (EOM), the end of memory reserved; START OF DATA (SOD), the beginning of the used portion of the IDBuff; and END OF DATA (EOD), the end of data in the IDBuff. The SOM and EOM are constants whereas the SOD and EOD are variables.

The SODBuff is composed of four buffers for four outbound messages (less the checksum). The buffers can be freed up to be overwritten when the message that they contain is acknowledged by the SLIP on the other side of the line. When a message is in the SODBuff, it has associated with it a sequence number (which is the message's sequence number). The Sender Task can reference the data in the SODBuff and reference acknowledgements via this sequence number.

When the application has data to be transmitted, it is placed in the IDBuff by the application using functions from the Manager and the EOD is incremented. If the data the application wants to send won't fit in the buffer, no data is written, and the application can either sleep, or continue to attempt to write data until the

data will fit. The Sender Task calls a Manager function to fill a message slot in the SODBuff. The Sender Task then sends its message from the SODBuff.

The Manager also maintains a buffer set for the Receiver Task. The buffers are similar to those of the Sender Task. There is a CHECKSUMMED OUTPUT-DATA BUFFER (CODBuff), which is the final output from SLIP that the higher level protocol may read. The CODBuff is also controlled by the four parameters START OF MEMORY, END OF MEMORY, START OF DATA, and END OF DATA (SOM, EOM, SOD, and EOD).

There is also an inbound circular buffer the analog of the SODBuff, called the RECEIVER TASK'S INPUT-DATA BUFFER (RIDBuff).

When the UARTian gets data, it places the data in the RIDBuff. After this, the Receiver Task checksums the data. If the checksum is good and the Receiver Task opts to acknowledge the message, it moves the data to the CODBuff, increments EOD, and frees up space in the RIDBuff. The higher level application can then take data off on the CODBuff, incrementing SOD as it does so.

STARTING UP AND FINISHING OFF COMMUNICATIONS

The problem is that the SLIP's on either side need to know (and keep knowing) the sequence number of the other SLIP. The easiest way to solve most of these problems is to have the SLIP check the Request to Send and Clear to Send Lines to see if the other SLIP is active. On startup, or if it has reason to believe the other side has died, the SLIP assumes: all connections are closed, no data from any connection has been sent, and both its SEQ and the SEQ of the other SLIP are zero. To start up a connection, the instigating SLIP sends a SSNM with its starting sequence number in it. The receiving SLIP acknowledges this SSNM and replies with its starting sequence number (combined into one message). Then the sending SLIP acknowledges the receiving SLIP's starting sequence number and the transmission commences. This is the three way handshake taken from TCP, After which data transmission can begin.