The Point to Point Protocol in Forth

2 Nov 2000

Howard Oakford
Inventio Software Ltd
www.inventio.co.uk

The Point to Point Protocol is used by PC programs to connect to an Internet Service Provider (ISP) so that applications can surf the web, or get email.

In this paper I describe an implementation of a PPP Peer in Forth.

English is not a good language to describe complex computer programs, so I have given some examples of Forth source.

Paper is not a good medium to present Forth source - it is better to try out the program interactively.

PPP is complicated. I have done everything I can to simplify it:

1. I captured the actual packets transferred between my PC and ISP when checking my email using a well known email program. This converted the enormous range of possible formats and options to the ones that are actually used.
2. I use a "flat architecture" and avoid the concept of an "IP-stack" or hierarchical layer of protocols. This follows the Forth-like "direct action" approach, where Forth words actually do something, rather than the C-like or operating system approach where functions process and pass data structures to other functions.
3. The current implementation has four tasks (apart from the Operator task). Again the tasks are used in a Forth-like way - each one defines an activity which is inherently asynchronous to the other tasks. The C-like approach to tasks is to use them to isolate functions written by different teams of people, so that each part of the software can operate asynchronously to each other part without interfering with each other.

The Point to Point Protocol Peer has the properties of both a Client (issuing requests) and Server (responding to requests). Each end of a PPP connection is identical to the other.

However, the Client and Server parts of PPP are not independent, as the PPP Peer must step through three levels to achieve an open PPP connection:

LCP Link Control Protocol level sets up the Maximum Transmission Unit (MTU) and other parameters required to pass PPP packets between the two PPP Peers.

The PAP/CHAP level authenticates the other PPP Peer.

The IPCP sets up the Internet Protocol parameters such as IP addresses and compression algorithms.

When all three levels have been achieved by both the Client and Server side, the PPP link is open, and IP packets may be sent in either direction.

IP packets carry a payload of data in one of the higher level formats such as UDP and TCP. A simple example of a UDP packet is given. TCP is more difficult and is my next project!

This is "work in progress", and the source listed below is only a small part of the current system. I intend to publish a working system as soon as it can get my email…
First the easy bit, PPP Packet Output:

The basic unit of currency on a PPP connection is the packet - a string of characters separated by a PPP Flag character: hex 7E.

The format of a PPP packet is well defined, and includes a PPP protocol field and CRC.

When sending a packet nothing except the payload is stored in an array, just as data on the stack or as literals in the program. For example, executing IP actually sends the payload as an IP packet, complete with PPP Flag, PPP header, IP header and checksum and PPP CRC. PPP "escape" characters (hex 7D) are sent before certain characters, these characters are XORd with hex 20. See the code below for details.

In a conventional IP stack, the payload would be sent to each layer in turn to have another header added to it, before being passed to an output function. Here we calculate the byte stream as we need it and send it immediately. This is what is meant by a "flat architecture".

```plaintext
( 720 )
( 0 ) (Output) HEX
( 1 ) 7D CONSTANT PPP_ESC 7E CONSTANT PPP_FLAG
( 2 )
( 3 ) VARIABLE ACCM : \ACCM 00000000. ACCM 2! \ACCM
( 4 ) CREATE [2**] 1 C, 2 C, 4 C, 8 C, 10 C, 20 C, 40 C, 80 C, 100 C,
( 5 ) \ACCM (c - 6) DUP 07 AND [2**] + C0
( 6 ) SWAP 8 / \ACCM + C0 AND ;
( 7 )
( 8 ) : ESC? (c - f) \ACCM OR THEN \ACCM ELSE DROP ;
( 9 )
( 10 ) DUP ESC? IF \ACCM \ACCM REMIT 20 XOR REMIT ;
( 11 ) \ACCM DUP IF \ACCM THEN \ACCM ELSE DROP ;
( 12 ) : ?ACCM (c - f) DUP 07 AND \ACCM + C0 AND ;
( 13 ) CREATE \ACCM 1 C, 2 C, 4 C, 8 C, 10 C, 20 C, 40 C, 80 C, 100 C,
( 14 ) SWAP 8 / \ACCM + C0 AND ;
( 15 ):
( 721 )
( 0 ) (Output) HEX
( 1 ) B0 DUP OFF AND HEMIT OFF AND HEMIT +1CS ;
( 2 )
( 3 ) : 2C> (c1 c2) SWAP \ACCM OR \ACCM ;
( 4 )
( 5 ) : D>> (d) \ACCM OR \ACCM ;
( 6 )
( 7 ) : \ACCM \ACCM OR \ACCM ;
( 8 )
( 9 ) : 2B> (a) \ACCM OR \ACCM ;
( 10 )
( 11 ) : PPP< SERIAL_OUT GET PPP_FLAG REMIT \CRC-OUT
( 12 ) FF HEMIT 03 HEMIT ;
( 13 )
( 14 ) : PPP< \CRC-OUT \ACCM + XOR \ACCM PPP_FLAG REMIT
( 15 ) SERIAL_OUT RELEASE ;
( 722 )
( 0 ) (LCP Output) HEX
( 1 ) 70 USER SO.UL.CE.IP 7A USER DESTINPRT
( 2 ) 7E USER DE.ST.IN.IP 7A USER DESTINPRT
( 3 )
( 4 ) : LCP (a n code seq) PPP< C021 \ACCM + (code seq) \ACCM
( 5 ) (Length) DUP (n) + \ACCM (a n) HEMIT >PPP ;
( 6 )
( 7 ) : LCP_MTU (MTU) PPP< C021 \ACCM (REQ code) 1 (seg) \ACCM
( 8 ) (Length) 0A \ACCM (MTU) \ACCM >PPP ;
```

\PPP_ESC\ is the PPP "escape" character
\PPP_FLAG\ is the PPP flag character
\ACCM = ASCII Control Character Map
\ACCM\ has one bit set for each character from 0 to hex 1F if that character must be escaped.
\ACCM\ looks up the bit for character c in the ACCM and returns true if it is set (meaning it must be escaped)
\ACCM\ returns true if c must be escaped.
The PPP escape character and the PPP flag character must also be escaped.
\ACCM\ sends c as one or two characters. Certain characters must not be sent in a PPP packet - they are XOR'd with 20 and preceded by a PPP escape character = 7D.
\ACCM\ also accumulates the CRC of them-escaped character
\ACCM\ sends an "escaped" string

\ACCM\ sends a 16 bit Word in Big Endian format
\ACCM\ sends two 8 bit Characters, c1 first then c2.
\ACCM\ sends a 32 bit Word in Big Endian format
\ACCM\ sends the 16 bit Word at a in Big Endian format
\ACCM\ sends the 32 bit Word at a in Big Endian format
\ACCM< .... >PPP send a formatted PPP packet
\ACCM< sends the initial Flag, address and control bytes
\ACCM< sends the accumulated CRC and terminating Flag
\ACCM< Note that since the checksum is 16 bits it is more efficient to send bytes two at a time, hence 2C> and no C>
\ACCM< The 1CS word does take care of odd length strings though.

\SO.UL.CE.IP\ the source IP address
\SOURCEPORT\ the source port number
\DE.ST.IN.IP\ the destination IP address
\DESTINPORT\ the destination port number
\LCP\ sends an LCP packet of code and sequence number
A bit more difficult is PPP input:

Again "direct action" is taken, rather than buffering and passing of pointers.

First the start of a PPP packet is detected, then the protocol type is used to calculate the header size, and the header is directed to the Header array and the payload ( the rest of the packet ) is directed to the Buffer array. The Header array is a small 64 byte array used to save the current header and any other non-payload data. It is destroyed when the next PPP packet arrives. The Buffer is a large ( 16K byte ) circular linked list which retains the most recent payloads. Only the oldest payloads are destroyed, and only when the Buffer is full.

The CRC is accumulated during this process, and is checked when the end of the packet is detected.

Detection of the start of packet of a PPP packet is tricky, because there may be one or more PPP Flag characters between any two packets.

---

LCP_MTU sends an LCP request with MTU setup option.

IPCPOpt sends a six byte IPCP option. Most options consist of an option type, length and 4 data bytes. Only one option is sent here: the IP addresses, code 3.
The first character of a packet is the first non-Flag character after one or more Flag characters have been received.

```
{ 733 }
{ 0 } ( PPP input) HEX
{ 1 } :PPPstart (- n) 0
{ 2 } IF OEC7: URCP
{ 3 } HKEY DUP PPP_FLAG = 0= ( this one is not a Flag )
{ 4 } FLAG_RECVD @ AND ( but we have had a Flag )
{ 5 } UNTIL 0 FLAG_RECVD !
{ 6 } DUP FF = IF ( an address byte, so remove the control byte )
{ 7 } DROP HKEY DROP ( and get the first protocol byte )
{ 8 } THEN ( otherwise it was the first protocol byte )
{ 9 } DUP 1 AND 0= IF :: HKEY OR THEN
{ 10 } SAVE-BUF ;
{ 11 }
{ 12 }
{ 13 }

{ 734 }
{ 0 } ( Peer state)
{ 1 } :CALC_PPP_PEER_STATE
{ 2 } PPP_CLIENT_STATE @ 3 = PPP_SERVER_STATE @ 1 = AND
{ 3 } IF ( LCP ) 1 PPP_PEER_STATE = .PPP_STATE THEN
{ 4 } PPP_CLIENT_STATE @ 2 = PPP_SERVER_STATE @ 2 = AND
{ 5 } IF ( PAP/CHAP ) 3 PPP_PEER_STATE = .PPP_STATE THEN
{ 6 } PPP_CLIENT_STATE @ 3 = PPP_SERVER_STATE @ 3 = AND
{ 7 } IF ( IPCP ) 3 PPP_PEER_STATE = .PPP_STATE THEN
{ 8 } PPP_CLIENT_STATE @ 4 = PPP_SERVER_STATE @ 4 = AND
{ 9 } IF ( Open ) 4 PPP_PEER_STATE = .PPP_STATE THEN
{ 10 } ;

{ 744 }
{ 0 } ( Protocols ) HEX
{ 1 } :PPPpacket PPPstart DUP PPP_PROT !
{ 2 } CASE
{ 3 } 0021 OF PPP_IP ENDOF
{ 4 } 8021 OF PPP_IPV6 ENDOF
{ 5 } C021 OF PPP_LCP ENDOF
{ 6 } C023 OF PPP_PAP ENDOF
{ 7 } C223 OF PPP_CHAP ENDOF
{ 8 } PPP_UNDEFINED ENDOF
{ 9 } ENDCASE ;
{ 10 } 
```

Having directed the incoming data stream to its rightful places, the data can now be processed. Four tasks are currently used to handle the Point to Point Protocol. The tasks allow the functionality of the protocol to be expressed clearly, without getting bogged down in the details of packet parsing and creation.

```
{ 773 }
{ 0 } ( Peer state)
{ 1 } :CALC_PPP_PEER_STATE
{ 2 } PPP_CLIENT_STATE @ 3 = PPP_SERVER_STATE @ 1 = AND
{ 3 } IF ( LCP ) 1 PPP_PEER_STATE = .PPP_STATE THEN
{ 4 } PPP_CLIENT_STATE @ 2 = PPP_SERVER_STATE @ 2 = AND
{ 5 } IF ( PAP/CHAP ) 3 PPP_PEER_STATE = .PPP_STATE THEN
{ 6 } PPP_CLIENT_STATE @ 3 = PPP_SERVER_STATE @ 3 = AND
{ 7 } IF ( IPCP ) 3 PPP_PEER_STATE = .PPP_STATE THEN
{ 8 } PPP_CLIENT_STATE @ 4 = PPP_SERVER_STATE @ 4 = AND
{ 9 } IF ( Open ) 4 PPP_PEER_STATE = .PPP_STATE THEN
{ 10 } ;

{ 774 }
{ 0 } ( Protocols ) HEX
{ 1 } :PPPpacket PPPstart DUP PPP_PROT !
{ 2 } CASE
{ 3 } 0021 OF PPP_IP ENDOF
{ 4 } 8021 OF PPP_IPV6 ENDOF
{ 5 } C021 OF PPP_LCP ENDOF
{ 6 } C023 OF PPP_PAP ENDOF
{ 7 } C223 OF PPP_CHAP ENDOF
{ 8 } PPP_UNDEFINED ENDOF
{ 9 } ENDCASE ;
{ 10 } 
```
PPP_MAIN repeatedly receives PPP packets and parses them for the other tasks to pick up from the Header and Buffer. It also updates the display of our PPP state.

PPP_CLIENT initiates requests to the remote PPP peer.

PPP_SERVER receives PPP packets and parses them.

PPP_KICK performs timeout retries.
And finally the top level user interface:

{ 779 )
( 0 ) ( Protocols )
( 1 ) : RUN COM1 9600 BAUD ( 115200 .BAUD ) HEX PAGE
( 2 ) 0, MyIPaddr 2! HisIPaddr 2TALLY
( 3 ) 23 0 TAB 20 24 WIN PPP_STATE .PPP_STATE
( 4 ) \BUF PPP_MAIN PPP_CLIENT PPP_SERVER PPP_KICK
( 5 ) 24 0 TAB DECIMAL ;
( 6 )
( 7 ) : VIEW RUN \PPP_KICK ;
( 8 )
( 9 ) : GO 1 PPP_CLIENT_STATE ! 1 PPP_SERVER_STATE ! ;
(10 )
(11 ) : \GO -1 PPP_CLIENT_STATE ! ;
(12 )
(13 ) : run RUN ; : go GO ; : view VIEW ; : \tasks \TASKS ;
(14 ) : K1 CR ." F1 = Help, F2 = Go, F3 = Reset, F4 = Stop tasks ";
(15 ) : K2 GO ; : K3 RUN ; : K4 \TASKS ;
\ RUN starts the PPP peer
\ VIEW starts the PPP peer in display only mode ( no Client actions )
\ GO opens a PPP link
\ Note that only one PPP peer needs to issue a GO command, as
\ the other PPP peer will be started by the receipt of an LCP configure request.
\ GO closes the PPP link
\ run is a lower case alias for RUN etc...
\ Press F1 for Help

The Forth program is written in 8086 polyForth. This is a 16 bit LittleEndian Forth with built in multitasking and uses blocks for the Forth source.

The current state of the program is that two PCs connected together with a null modem cable, and both running the program will open a PPP connection. PAP and CHAP are not supported yet, although they are there in outline form and the MD5 algorithm ( for CHAP ) is complete and tested.

The LCP and IPCP options are defined at compile time, except for the IPCP IP address option which handles an IPCP configure request with a zero IP address option field by returning a NAK and new IP address.

The Buffer array circular queue does not support linked lists yet. This is only relevant to IP and higher protocols.

The PPP packet parsing and display functions correctly show the packets captured from the PC during fetching of email, and correctly calculate the PPP packet CRC and IP and UDP header checksums.

Howerd Oakford   2 Nov 2000