AN724

Using PICmicro® MCUs to Connect to Internet via PPP

SOFTWARE LICENSE AGREEMENT

The software supplied herewith by Microchip Technology Incorporated (the “Company”) for its PICmicro® Microcontroller is intended and supplied to you, the Company’s customer, for use solely and exclusively on Microchip PICmicro Microcontroller products.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

Author: *Myron Loewen*
Microchip Technology Inc.

INTRODUCTION

PICmicro microcontrollers (MCU) are suitable for low-cost connections to the Internet. The desire to connect everything to the Internet focuses the price reduction challenge on the Internet interface. Typically, the interface is an overpowered embedded PC running a bulky operating system and memory intensive applications. For low-cost applications that handle smaller amounts of data, a much better choice is the Microchip family of PICmicro MCUs. This application note will show how these little processors are capable of connecting to the Internet with resources to spare for controlling the original application.

The software will dial into the Internet and try to connect to the server using Point to Point Protocol (PPP). It continues pinging once every 30 seconds to keep the connection open while responding to ping requests. With the Internet Protocol (IP) connection established you can add your own algorithms for traceroute, Trivial File Transfer Protocol (TFTP), Simple Network Management Protocol (SNMP), or get the time and date accurate to a millisecond.

Since there are lots of good books and free Internet resources to describe how the Internet works, this application note will focus on the less publicized details of negotiating PPP. Another common protocol used to connect to the Internet by modem is the Serial Line Internet Protocol (SLIP). PPP was chosen for this application note instead of the simpler SLIP because it is much more versatile. PPP has the advantage of not requiring a unique login script for most servers. Another advantage of PPP is line quality monitoring. Although not implemented in this algorithm, it is useful when reliable communications is a top priority. The most important reason is to maintain compatibility with Internet Service Providers by riding the popularity of desktop operating systems, which use PPP by default.

This Internet interface requires a physical connection to a local Internet Service Provider (ISP) with a serial line or modem. The rest is all software. The algorithm requires about 145 bytes of RAM and 2170 words of ROM. The amount of processor time available for other tasks will depend on the processor's clock speed and baud rate of the serial connection. The algorithm takes time for each received or transmitted character and extra time to process or create a data or control packet.

This algorithm does not include Transmission Control Protocol (TCP) which is required for email, Telnet, web browsing, and File Transfer Protocol (FTP). These algorithms could be added but they require a processor with a lot more RAM and ROM. This algorithm will not connect to every server; it attempts an unscripted PPP login and falls back to a generic script. If it fails, some login tweaking or implementing more Link Control Protocol (LCP) options will usually bring up the connection.

FIGURE 1: PHOTO OF PROTOTYPE CONNECTED TO THE INTERNET

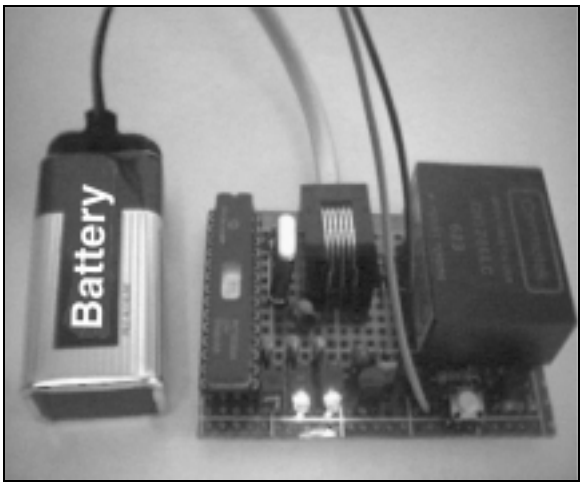
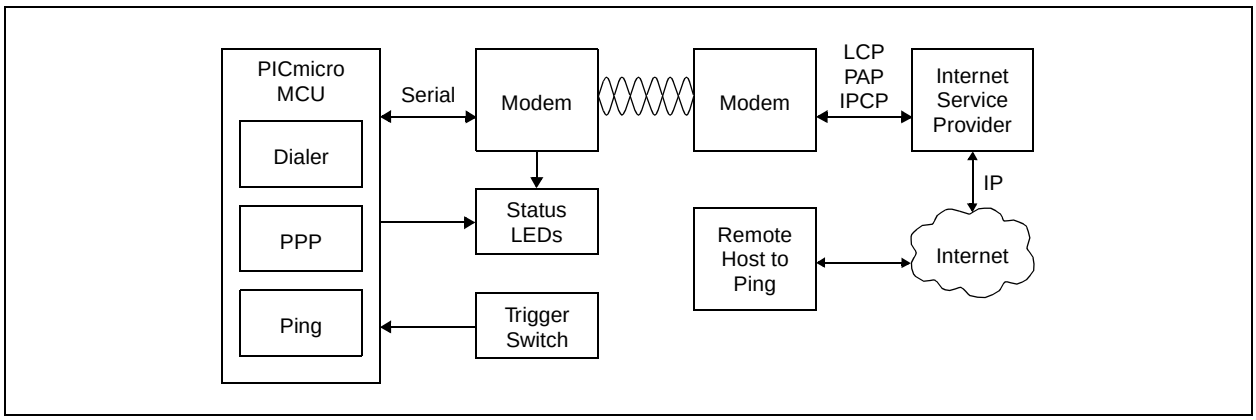


FIGURE 2: BLOCK DIAGRAM



INTERNET PROTOCOLS

The Internet is just a very large bunch of many types of computers connected in a variety of ways. What makes it all work is the thousands of standards and conventions that all computers follow. Most standards are documented and freely available on the Internet. Table 5 lists the standards needed for this algorithm and where to find them.

Data gets around the Internet in specially marked packets that are passed from computer to computer. The packets indicate the type of data they contain such as a part of a web page or email. Each packet gets stuffed in its own envelope specially marked to get it to the right program on the remote computer. This is important because you may be running several web browsing windows simultaneously on one machine. The type of data determines if the envelope is the simpler UDP type or a more robust TCP type. TCP packets generate extra packets to open and close the connection and resend lost packets.

Each computer gets a unique Internet address that looks like 10.241.45.12, and works much like a postal mailing address. The envelope is stuffed in a larger envelope with the source and destination addresses written on the front. This is like international mail, the address will get it anywhere on the Internet.

But the Internet works more like passing notes in class. The larger envelope goes into a bigger envelope with your friend's name on it. Your friend opens the envelope and checks the Internet address. If he is the recipient, he processes it, otherwise he puts it in a new big envelope. From the Internet address he can tell which direction to send the envelope and puts the name of his neighbor, in that direction, on the front. The process repeats until the envelope makes it to the correct Internet address across the class, or gets lost along the way.

In this algorithm only the ping packets travel this way. The other packets have the same format but are just exchanged locally between this algorithm and the Internet server it dials up. These packets are discussed in the next few sections and are used by both ends to configure the serial link options.

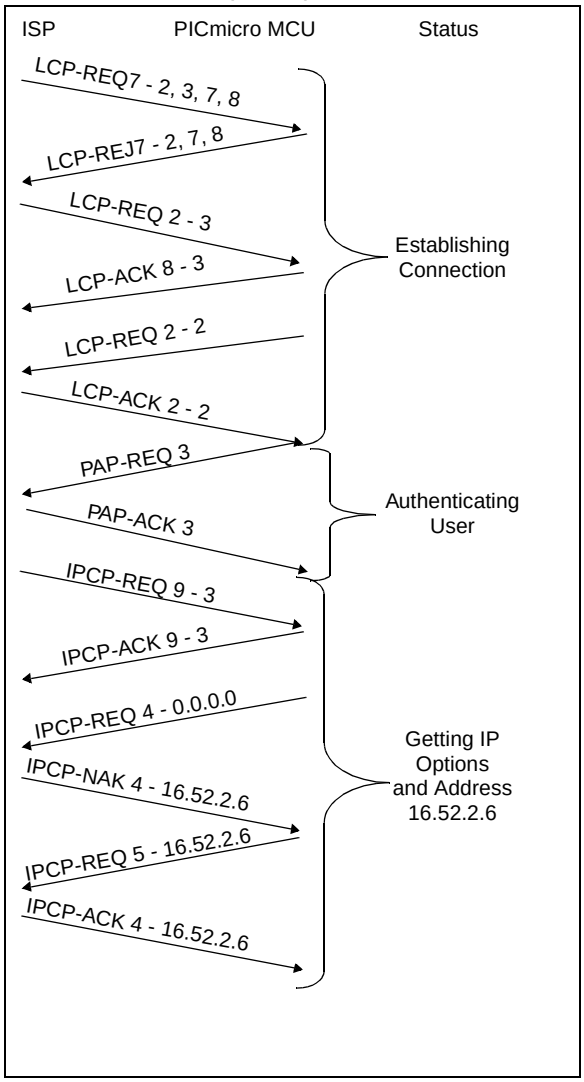
PPP requires the serial data format to be set to eight data bits with no parity. The PPP data is sent as packets that start and stop with the tilde character (~) or in hexadecimal 0x7E. Because ~ has a special meaning, any other instance of ~ is replaced by the }^ escape sequence. The escape sequence works by transmitting two characters instead of the original, first the } and then the original exored with the space character, or in hexadecimal 0x20. Because the } has a special meaning, it too is also escaped in the same way resulting in the 2-character sequence }]. For compatibility with all serial links, the control characters 0x00 to 0x1F can also be optionally mapped into the 2-character escape sequence. For more complete details, read RFC 1662 *PPP in HDLC-like Framing*.

The PPP connection can be broken into several phases. First, if the link is dead, carrier detect from the modem is one of the stimuli that starts the link establishment phase. This phase uses Link Control Protocol (LCP) to detect and negotiate link options with the remote computer.

Next the authentication phase verifies the User ID and password with Password Authentication Protocol (PAP). Although not one of the phases, this is where ISPs negotiate compression protocols. The final phase is the network layer protocol. Each protocol such as IP, is configured with its control protocol like IPCP.

Control protocols are very similar for LCP, PAP, CCP, and IPCP but the protocol field is different and the options have different meanings. Each packet can request, deny, or accept a list of options. Negotiation starts with either side requesting a list of options in a request (REQ) packet. Each option consists of an option type byte, length byte, and option parameters. If the receiving end likes all the options, it replies with an acknowledge (ACK) packet.

FIGURE 3: PPP NEGOTIATION WITH REQ, ACK, NAK, AND REJ PACKETS



If it doesn't like some parameters, it responds with a not acknowledge (NAK) packet that repeats all the options it rejects and replaces the rejected parameters with acceptable values. If required options are missing those are added to the NAK reject list.

If some options are not recognized or are considered non-negotiable they are rejected with the REJ packet that lists all the bad options. The first side updates its option list and retransmits requests until it gets an ACK reply packet. The other side can start negotiations at any time and the resulting link may have different options for each direction. The terminate, code reject, protocol reject, echo, and discard control packet types are not implemented in this algorithm. The details are broken down into a section for each control protocol.

TABLE 1: ACRONYMS

Acronym	Description
ACK	Acknowledgement
CCP	Compression Control Protocol
CRC	Cyclic Redundancy Check
CHAP	Challenge-Handshake Authentication Protocol
DNS	Domain Name System
DTR	Data Terminal Ready
FTP	File Transfer Protocol
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IPCP	Internet Protocol Control Protocol
ISP	Internet Service Provider
LCP	Link Control Protocol
MRU	Maximum Receive Unit
NAK	Negative Acknowledgement
PAP	Password Authentication Protocol
PPP	Point-to-Point Protocol
REJ	Reject
REQ	Request
RFC	Request For Comment
SLIP	Serial Line Internet Protocol
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
TTL	Time To Live
UDP	User Datagram Protocol

DIAL-UP SCRIPT

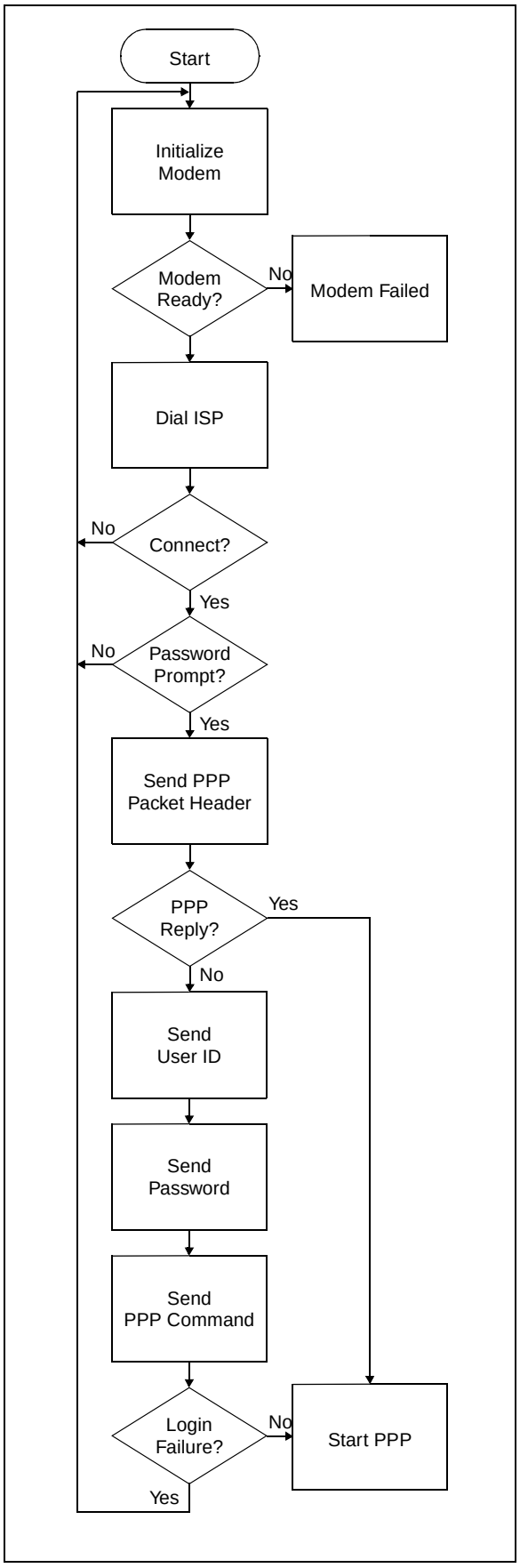
We cannot start link negotiations without a physical connection to the Internet. In this algorithm, a dial-up modem makes the connection and then the link is negotiated. The modem dial script could be removed for circuits with a direct connection to the server's serial port. The `sendwait (command, test, timeout)` routine does most of the work for the script. The script sends the command string and then returns when it receives the test string or the time expires. The timeout units are 10 milliseconds, thus a timeout of 100 is 1000 milliseconds or 1 second.

First the modem is taken into command mode with a pause, then three plus characters (+) and another pause. The | character in the command string causes a 1 second pause; use them where required in any `sendwait()` command string. Then, the `ATH` command hangs up the modem in case it was already off hook. The `ATZ` and `AT&F` commands then reset the modem and restore factory default settings. `ATS11=50` speeds up the tone dialing to reduce connect time. The modem has 3 seconds to echo the command before the algorithm aborts and assumes the modem is not functioning.

The algorithm indicates dialing by flashing the status LED. Then it sends the dial string to the modem and waits 30 seconds for the modem to respond with the connect message before the algorithm gives up and tries the whole process again. If the dial string contains pauses, or the modem is faster than 2400 baud you may need to increase the timeout. When the modem connects, the connection LED stays on steady, otherwise it is turned off.

The script waits 10 seconds for a colon followed by a space to detect when the server is prompting for the password. Whether it gets the prompt or not, it sends the first PPP packet. This makes use of the fact that most servers switch to PPP login instead of further text prompts when they get the first few characters of a PPP packet instead of a valid User ID. If the bait is taken, the script ends and PPP negotiation begins. Otherwise the script continues by sending the User ID, password, and command to enter PPP. The IP address is not captured at this time because the IPCP negotiations will capture it later.

FIGURE 4: PPP LOGIN FLOW CHART



LCP OPTIONS

The LCP options are negotiated first to establish the link. A sample packet is shown in Figure 5. It has the normal PPP header of 0x7E 0xFF 0x03 followed by 0xC0 0x21 to indicate that the protocol is LCP. The LCP packet consists of a code, identification, length, and a list of options to configure followed by the standard 2 byte PPP CRC. The code is a byte to indicate the meaning of the packet. A list of codes is found in Table 2. The identification byte is incremented after each negotiation request, which makes requests unique and connects them to the correct reply. The 16-bit length is the number of bytes in the LCP packet, four for the header plus the sum of the lengths of each option.

The list of possible options is found in Table 3. Each option is sent as a one byte option type, followed by a one byte option length, and an optional parameter. The option length is two for the option header plus the number of bytes in the parameter. Here is a brief description of the more common options:

1 **Maximum-Receive-Unit** – The 2 byte parameter is the maximum size of PPP packets. It would be nice to make this value very small to conserve buffer space in the limited PICmicro MCU RAM. However, the minimum allowable value of 576 is much too big to help. Since the MRU option has no benefit and can be safely left at the 1500 default, this algorithm doesn't waste code space to support it. Note that packets longer than the 47 byte buffer size are truncated to fit, and typically the longest packet to handle is about 40 characters. Some ping packets are much longer but they are quite tolerant of losing the extra padding characters.

2 **Async-Control-Character-Map** – The 4-byte parameter in this option adds up to 32 bits: each bit represents one of the ASCII control characters from 0 to 31. Starting with the most significant bit as character 31 and the least significant as character 0. If the bit for the character is a one then that character must be transmitted as a } sequence. This way the server and client can decide to escape only the characters which may cause problems instead of wasting bandwidth escaping all control characters. Even characters that do not need to be escaped may be, this algorithm exploits that to simplify the software and to transmit all control characters as two bytes.

3 **Authentication-Protocol** – This option chooses the method of sending the password. Unless you have already logged in with the script, this option will be required. A parameter value of 0xC023 selects Password Authentication Protocol (PAP) which sends a packet containing the User ID and password in plain text. A parameter value of 0xC223 selects the Challenge Handshake Authentication Protocol (CHAP) in which the User ID is sent in plain text and the password encrypts and returns a random string from the server. On the server, the password encrypts the same string; if the two results match, the user is logged in. For simplicity, this algorithm only supports the PAP method. So far no ISP has forced it to use the CHAP method.

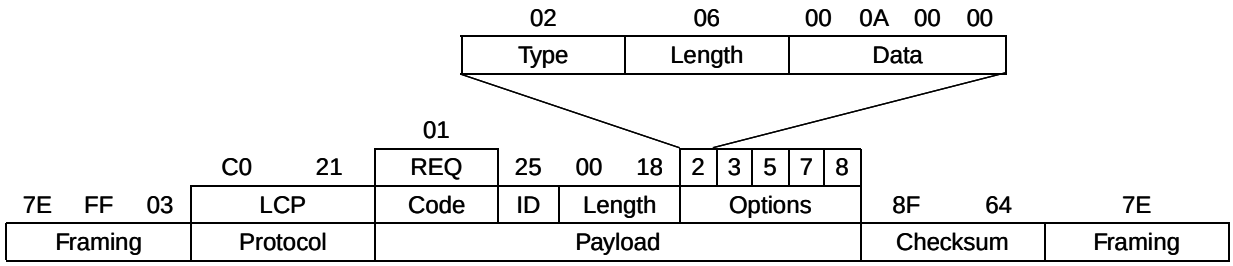
5 **Magic-Number** – This option did not need to be implemented for the PPP negotiations to converge. The 4 byte parameter is a random number; if identical to the server's, then both ends choose another random number. Assuming good randomness, the chance of random numbers not being unique after three iterations is so low that the transmission path is assumed to be looped back, just echoing packets sent.

7 **Protocol-Field-Compression** – This option has no parameters. If requested, the acknowledging side may transmit future PPP packets with the first byte of the 2-byte protocol field left out. This is meant to save bandwidth. It is easy to uncompress - if an odd byte arrives at the start of the protocol field it must be the second byte since the first byte is always even, and the second is always odd. A zero is inserted for the missing first character.

8 **Address-and-Control-Field-Compression** – This option has no parameters. However, if requested, the acknowledging side may transmit future PPP packets with the second and third bytes, 0xFF and 0x03, left out. This is also meant to save bandwidth. It is also easy to decompress because if the first character in the packet is not 0xFF, a 0xFF is inserted first, if the second character is not 0x03, a 0x03 is inserted first.

The other options didn't need to be implemented to make the Internet connection, but as standards evolve in the future a missing option could prevent login. Note that only options up to number 16 can be negotiated without modifying the `TestOptions()` routine. It has a one word parameter called option in which each of the 16 bits indicate acceptance of an option. For example, if the Most Significant Bit (MSB) is set, then option 16 is accepted; if bit 0 is cleared, then option 1 is rejected. LCP is complete when both sides of the connection have their list acknowledged by the other side.

FIGURE 5: A SAMPLE LCP REQUEST PACKET



The complete packet with more option details:

```
0000: 7E FF 03 C0 21 01 25 00 18 02 06 00 0A 00 00 03
0010: 04 C0 23 05 06 71 AE BE D2 07 02 08 02 8F 64 7E
```

Same LCP packet with } escape sequences:

```
0000: 7E FF 7D 23 C0 21 7D 21 25 7D 20 7D 38 7D 22 7D
0010: 26 7D 20 7D 2A 7D 2A 7D 20 7D 23 7D 24 C0 23 7D
0020: 25 7D 26 71 AE BE D2 7D 27 7D 22 7D 28 7D 22 8F
0030: 64 7E
```

TABLE 2: LCP NEGOTIATION CODES

Type	Packet Type	Details
0	Vendor Specific	RFC2153
1	Configure-Request	RFC1661
2	Configure-Ack	RFC1661
3	Configure-Nak	RFC1661
4	Configure-Reject	RFC1661
5	Terminate-Request	RFC1661
6	Terminate-Ack	RFC1661
7	Code-Reject	RFC1661
8	Protocol-Reject	RFC1661
9	Echo-Request	RFC1661
10	Echo-Reply	RFC1661
11	Discard-Request	RFC1661
12	Identification	RFC1570
13	Time-Remaining	RFC1570

TABLE 3: LCP OPTIONS

Type	Configuration Option	Details
0	Vendor Specific	RFC2153
1	Maximum-Receive-Unit	RFC1661
2	Async-Control-Character-Map	RFC1662
3	Authentication-Protocol	RFC1661
4	Quality-Protocol	RFC1661
5	Magic-Number	RFC1661
6	Quality-Protocol	Deprecated
7	Protocol-Field-Compression	RFC1661
8	Address-and-Control-Field-Compression	RFC1661
9	FCS-Alternatives	RFC1570
10	Self-Describing-Pad	RFC1570
11	Numbered-Mode	RFC1663
12	Multi-Link-Procedure	Deprecated
13	Callback	RFC1570
14	Connect-Time	Deprecated
15	Compound-Frames	Deprecated
16	Nominal-Data-Encapsulation	Deprecated
17	Multilink-MRRU	RFC1990
18	Multilink-Short-Sequence-Number-Header	RFC1990
19	Multilink-Endpoint-Discriminator	RFC1990
20	Proprietary	
21	DCE-Identifier	RFC1976
22	Multi-Link-Plus-Procedure	RFC1934
23	Link Discriminator for BACP	RFC2125
24	LCP-Authentication-Option	
25	Consistent Overhead Byte Stuffing (COBS)	
26	Prefix elision	
27	Multilink header format	
28	Internationalization	RFC2484
29	Simple Data Link on SONET/SDH	

PAP OPTIONS

The PAP details can be found in RFC 1334. For this algorithm they were simplified to one packet exchange. The PAP packet is similar to LCP with 0xC023 instead of 0xC021 in the protocol field. Instead of negotiating options, only the User ID and password are sent as a request. If the server acknowledges, then the user is logged in. A NAK reply would mean that the User ID or password is incorrect. The format can be seen in Figure 6. The first payload byte is the length of the User ID, followed by the User ID. The password is appended in the same way: Length first followed by password.

FIGURE 6: A SAMPLE PAP REQUEST PACKET

	C0 23	01	04	0014	06 userid	08 password		
7E FF 03	PAP	Code	ID	Length	User ID	Password	58 3D	7E
Framing	Protocol	Payload					Checksum	Framing

The complete packet:

0000:	7E	FF	03	C0	23	01	04	00	14	06	75	73	65	72	69	64
0010:	08	70	61	73	73	77	6F	72	64	58	3D	7E				

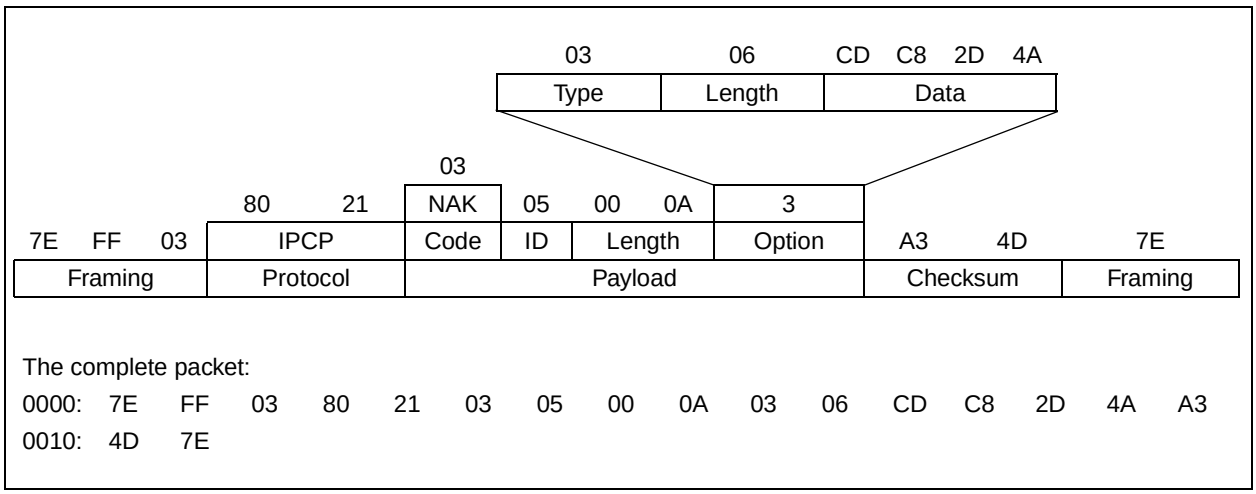
IPCP OPTIONS

After LCP is negotiated and PAP is accepted, the Internet Protocol must be configured. The options are for IP address and IP Compression with more details in RFC 1332. IP address is option three and its 4-byte parameter is the Internet address of this node. The server typically sends a request with option three followed by the IP address. Otherwise, the address is found by requesting an invalid choice like 0.0.0.0 and the server replies with a NAK and option three with the correct address. A sample packet is shown in Figure 7. Some server implementations may request IP Compression Protocol option type two. These requests are rejected because TCP is not implemented in this algorithm. Table 4 shows the IPCP configuration option types.

TABLE 4: IPCP CONFIGURATION OPTION TYPES

Type	Configuration Option	Details
1	IP-addresses	Deprecated
2	IP-Compression-Protocol	RFC1332
3	IP-address	RFC1332
4	Mobile-IPv4	RFC2290
129	Primary DNS Server address	RFC1877
130	Primary NBNS Server address	RFC1877
131	Secondary DNS Server address	RFC1877
132	Secondary NBNS Server address	RFC1877

FIGURE 7: A SAMPLE IPCP NAK PACKET



CCP OPTIONS

Some servers will try to negotiate compression, but since this algorithm is optimized for size instead of speed, these requests are rejected. The compression algorithms are complex and in some cases proprietary, yet have little benefit on the short packets used in this algorithm. Choosing the puddle jumper option type 3 means that no compression or decompression is required.

ICMP COMMUNICATIONS

The Internet Control Message Protocol messages are sent with full IP packets, an example is shown in Figure 8. This protocol is used to implement ping, but it has many other uses that you can read about in RFCs 792 and 950. Ping works by sending a packet to a remote Internet address and waiting for the reply, like radar or sonar from which it gets the name. This algorithm pings a fixed address once every 30 seconds to maintain an ISP connection. During this time it also responds to pings initiated by remote devices on the Internet.

The description of this packet is better understood as two parts, the IP header and the ICMP message. The packets discussed up to this point were just for setting up the serial link and never made it past the server. A lot more information is packed into the IP header: Its 20 bytes containing the instructions to take it anywhere on the Internet.

The first byte is broken into two nibbles, the first 4 bits are the IP version which is currently still four. The next 4 bits are the header length, which is the number of 32 bit words in the header, 5 in this case. The second byte is the type of service to optimize for: minimize delay, maximize throughput, maximize reliability, or minimize monetary cost. The recommended value for ping is 0x00 which means normal service with no optimization. The third and fourth bytes are the 16 bit total length of the IP header plus the following data such as the ICMP message.

The next 4 bytes are for fragmented packets and since these packets are so small, this algorithm ignores fragmentation. The ninth byte is the time to live (TTL) flag and it sets the maximum number of routers a packet can pass through before it is discarded. This is important to keep the Internet from getting clogged with lost packets. The TTL flag is usually set to 32 or 64 and decremented by each router the packet passes through. The tenth byte is a protocol field which says what type of information the IP header is attached to.

Bytes 11 and 12 are called the header checksum, which is a 16 bit one's complement sum of the 20 header bytes. For implementation details check RFC 1071 which has a very good description and sample C code. Basically, a 16-bit one's complement sum is the 16 bit sum of 16 bit data where overflow carries into bit 16 are wrapped around and added to bit 0. The next 4 bytes are the source IP address and the last four bytes are the destination IP address.

The ICMP message follows with a type byte, code byte, and checksum word, see Figure 9. The type byte is 0 for a ping reply or 8 for a ping request. The code byte is zero in both cases and the checksum is again a one's complement sum. This time the checksum is the sum of the ICMP header plus the following data. The amount of data is the IP header total length minus the IP header length. In the case of a ping the originator stuffs in some data to see if it is properly echoed by the ping reply. This arbitrary data could just as well be your collected data or other information you wish to send. This algorithm responds to ping requests without echoing back any of the arbitrary data and causes some ping programs to report an error.

FIGURE 8: INTERNET PROTOCOL PACKET SHOWING MEMORY LOCATIONS IN RX BUFFER

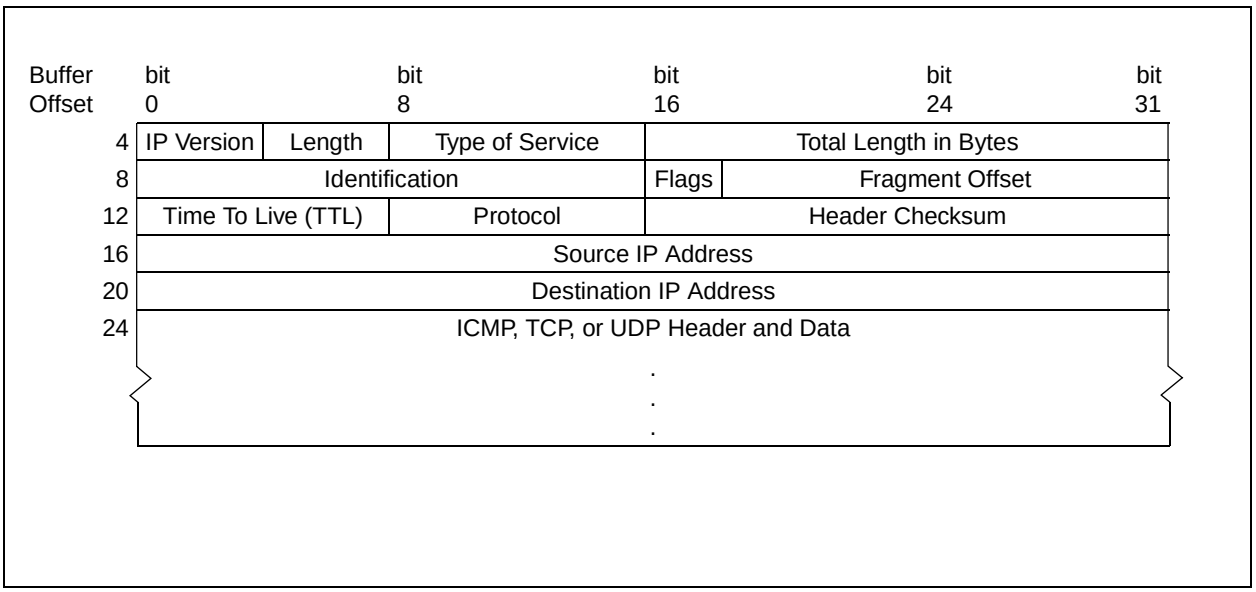


FIGURE 9: A SAMPLE OF A PING WITH NO OPTIONAL DATA

Buffer Offset	bit 0	bit 8	bit 16	bit 24	bit 31	PPP Packet
						0000 : FF 03 00 21
4	Version 0100	Length 0101	Service 0000 0000	Total Length 00000000 00011100		0004 : 45 00 00 1C
8	Identification 1000 1000 0001 0000		Flags 010	Fragment Offset 00000 00000000		0008 : 88 10 40 00
12	TTL 0111 1111		Protocol 0000 0001	Header Checksum 00110011 10100111		000C : 7F 01 33 A7
16	Source IP Address 11001101 11001000 00101101 01111100					0010 : CD C8 2D 7C
20	Destination IP Address 11001111 10100001 01110101 01000011					0014 : CF A1 75 43
24	ICMP Type 0000 1000		ICMP Code 0000 0000	ICMP Checksum 11110111 11111110		0018 : 08 00 F7 FE
28	PING Identifier 00000000 00000001			PING Sequence Number 00000000 00000000		001C : 00 01 00 00
						0020 : 22 7C 7E

UDP DETAILS

UDP is the protocol required to transfer files with TFTP, convert host names to IP addresses with DNS, or status and event reporting with SNMP. Its simplicity and bandwidth efficiency make it an important part of some multimedia Internet protocols. The official specification is found in RFC 768.

This algorithm doesn't support UDP protocols but this section will give you a bit of background and make it easier for you to add it to the algorithm. First of all UDP is an unreliable protocol, not that it should be avoided, but rather that packets can get lost without warning and may require retransmission. It is deterministic in the sense that each packet, or timeout, triggers the next event without regard to what state the connection is in. This simplifies programming and makes debugging much easier.

The format of UDP is shown in Figure 10. There are 20 bytes of IP header, then 8 bytes of UDP header, and the UDP data. The first two 2 byte fields are the source and destination port numbers. The port numbers are important to identify what process gets the UDP data. An example is port 69 which is always used for TFTP.

The next two bytes are the UDP length, eight bytes of UDP header plus the length of the UDP data. This value is redundant because it can be calculated from the IP header by subtracting the header length from the total length.

The last 2 bytes of the UDP header are the 16 bit one's complement checksum of the pseudo header, the UDP header, and the UDP data. The pseudo header is not transmitted but the following 12 bytes are added to the checksum anyway. The 32 bit source and destination addresses, the 16-bit UDP address, and the 8-bit protocol field are extended to 16 bits to ensure that the UDP data is going to the correct IP address.

The checksum is optional and set to zero if not used. Since zero means no checksum, then a valid checksum that adds up to zero must be inverted to 0xFFFF. If the UDP data is an odd number of bytes, your 16 bit checksum routine will need to pretend that there is an extra byte 0x00 at the end.

The format of the UDP data will depend on which port you are connecting to and which protocol is using the data. A good example is the Trivial File Transfer Protocol (TFTP) which is well documented in RFC 1350.

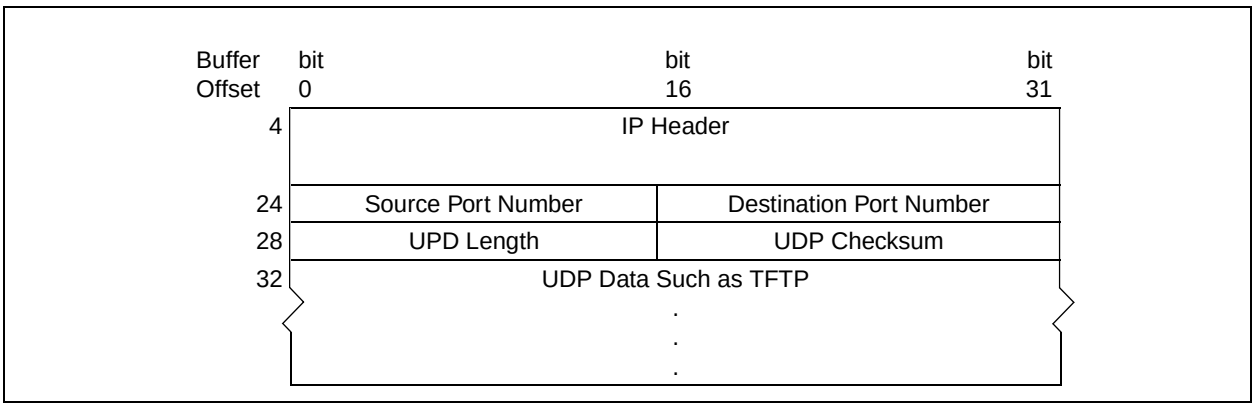
TCP DETAILS

TCP is the protocol required to transfer files with FTP, communicate by email with SMTP, login remotely with Telnet, or serve web pages with HTTP. The original specification is found in RFC 793; however, it has been improved by the Host Requirements RFCs 1122 and 1123.

This algorithm doesn't even pretend to support TCP packets because of the larger RAM and ROM requirements. Parts of the protocol may fit in a PICmicro MCUs with larger program memory size, so here is a little information for those brave enough to try. TCP is considered a reliable protocol because it hides lost and missing packets from the running applications: it tracks and retransmits them in the background.

This is really no different than UDP for the purpose of this algorithm since there is no distinction of software levels. In both cases the same process is responsible to retransmit missing packets. The difference is in the complexity and size of the packets. Another difference is that the other end of the connection is expecting this algorithm to keep track of packet timing, retransmit previous packets, and remember the state of multiple simultaneous connections.

FIGURE 10: THE UDP PACKET FORMAT



HARDWARE IMPLEMENTATION

This application note was designed for the PIC16C63A to demonstrate how compact the PPP connection algorithm can be shrunk. The algorithm uses 151 of the 192 file registers and 2.2k of the 4k ROM and only six I/O pins. There is still plenty of space for your own code and the code is portable enough to move it into a smaller or larger PICmicro MCU. The 4 MHz crystal is fast enough and could be slowed down, unless you need a faster modem or run additional CPU intensive tasks.

The modem is a Ceremtek CH1786LC, running at 2400 baud, but with packet sizes under 50 bytes speed is not much of an issue. For higher traffic connections or large data transfers you may want to upgrade to the larger but still pin-compatible 14.4 kilobaud CH1794. Be sure to check with the modem manufacturer what external circuitry is required before connecting to the telephone line. Your circuit must be designed and tested to meet the telecom standards of the country in which you wish to use it.

Only the telephone line, power, serial transmit, receive, and DTR line need to be connected to use this modem. The DTR line must be tied low for the modem to operate properly. The modem can be very sensitive to power supply noise so be sure to keep it close to a bypass capacitor. You could also change the software a little and replace the modem with an RS232 driver to go directly to the server's serial port.

One desirable characteristic of this modem is that it draws a maximum of 50 mA. Since it is only on for brief periods and the entire circuit never draws more than a total of 65 mA, we can easily power it off a 9V battery. A typical 9V alkaline battery with a 560mA hour rating would give us about 9 hours of power. The modem off hook to ping time is under a minute, so if we just send one ping and hang up, the battery would last for more than 500 pings. This works out to be about a year and a half at one ping per day.

This requires the power supply to turn off completely after the ping is successful. The power supply design uses a NPN transistor to turn on a PNP transistor which supplies the current to the voltage regulator. The NPN transistor can be turned on by the processor or by a momentary switch. When the momentary switch closes it turns on the power, and as the microcontroller initializes it too turns on the NPN transistor. By this time the user has released the switch and the microcontroller keeps the power on. The switch could be almost anything like a magnetic burglar sensor or even a thermostat. When the ping is complete it releases the power and turns itself off. If you need to ping the device, just keep the manual switch closed until you want the power off. With the switch released, it will power down after the first successful ping or an automatic timeout if there is no modem connection, password fails, or no ping replies.

The PNP transistor is also a benefit to reducing power consumption because no voltage drop is lost to a reverse protection diode. You may also upgrade to a low dropout (LDO) voltage regulator to get a little extra life out of the battery before the 5 volt regulator stops regulating. Choose a regulator that includes a power switch, or change the PNP to a MOSFET to reduce the current draw by one mA and add another two percent to the battery life. Rather than improving efficiency with a DC-DC converter, I would choose a lower-voltage battery pack with a flat discharge curve that barely maintains the minimum LDO regulating voltage. If the device will be inaccessible or needs a long shelf life, then go with a lithium-ion battery pack.

There are three LEDs: one indicates the modem status and the other two indicate serial data transmitted and received. The modem status LED is off while the software initalizes the modem, flashes quickly while dialing, and goes on steady when connected. If it goes off after flashing then it didn't connect and it will try again in a couple seconds. After connecting and negotiating PPP the status light will go off for a second and then flash out its 32 bit IP address. A long flash is a one and a short flash is a zero. Write down each bit as it flashes and then convert the binary to hexadecimal. Make it easier by grouping the bits in fours, each a hexadecimal character. Insert three decimals spaced every 8 bits, convert the four numbers to decimal, and you have your IP address, see Figure 11. This address is usually dynamically assigned resulting in a different address every time it logs on the Internet.

Note:	For first-time developers of PICmicro MCUs, using the Microchip PICDEM Demonstration boards (DV163002) may be useful.

FIGURE 11: CALCULATION OF IP ADDRESS FROM LED PULSES

Record long flashes as 1 bit.
Record short flashes as 0 bit.
There will be a pause every eight flashes.
Example:

	1100	1101		1100	1000		0010	1101		0100	1010
Hexadecimal:	C	D	•	C	8	•	2	D	•	4	A
Decimal:	205		•	200		•	45		•	74	

SOFTWARE IMPLEMENTATION

The software is written in C to keep it portable. This way the algorithm can be developed on a PC and tested with a variety of low-cost compilers and debuggers or simply print all relevant data to the screen. Then, just press PrintScreen or use DOS to pipe the screen output to a file for analysis. To compile for a PC, replace the serial functions with COM port routines and use the PC tick counter at address '0040:006C' instead of TMR0.

There are a number of excellent C compilers for the PICmicro MCU. This code was started with the free compiler CC5X from B. Knudsen Data in Norway and completed with the PICmicro MCU C Compiler from HiTech in Australia. The code shown will compile with the HiTech demo available at http://www.htsoft.com. It should work with all the other C compilers for the PICmicro MCU if you do all compiler specific modifications required to the code.

The code consists of a main routine that does the two main tasks of modem control scripting and the protocol state machine. There are a couple of support routines for calculating the CRC checksums, creating packets, checking configuration options, and controlling the modem.

When you press the power switch, the microcontroller powers up, does a short time delay loop, and asserts RB3 to keep the power on. The 250 millisecond time delay is meant to prevent false triggering. As long as either the user is pressing the power button or RB3 is asserted the power will stay on. The software will release RB3 to turn off the power after it successfully pings a remote host, times out trying, or fails 20 dial attempts. If the power button is still pressed, the software continues to dial or attempt more pings until the button is released. For example, if you want to ping it from another computer you will have to hold in the button until you complete your ping tests.

The software will attempt to phone the number programmed in the source code up to 20 times at about 30-second intervals. When it connects it tries going from a script login to a PPP login by sending a PPP packet instead of a User ID. If that fails, it falls back to a script login; otherwise it goes into the main loop of the algorithm. The protocol state machine loop does all the serial I/O, packet processing, packet creation, and timing to negotiate PPP and complete a ping.

The state machine starts in state 0. When the Internet server acknowledges the LCP configuration packet state, bit 0 is set. When the algorithm acknowledges the server's LCP configuration, bit 1 is set. As long as bit 0 is clear, the algorithm will send an LCP request once every second. When both bits are set the algorithm moves into state 4.

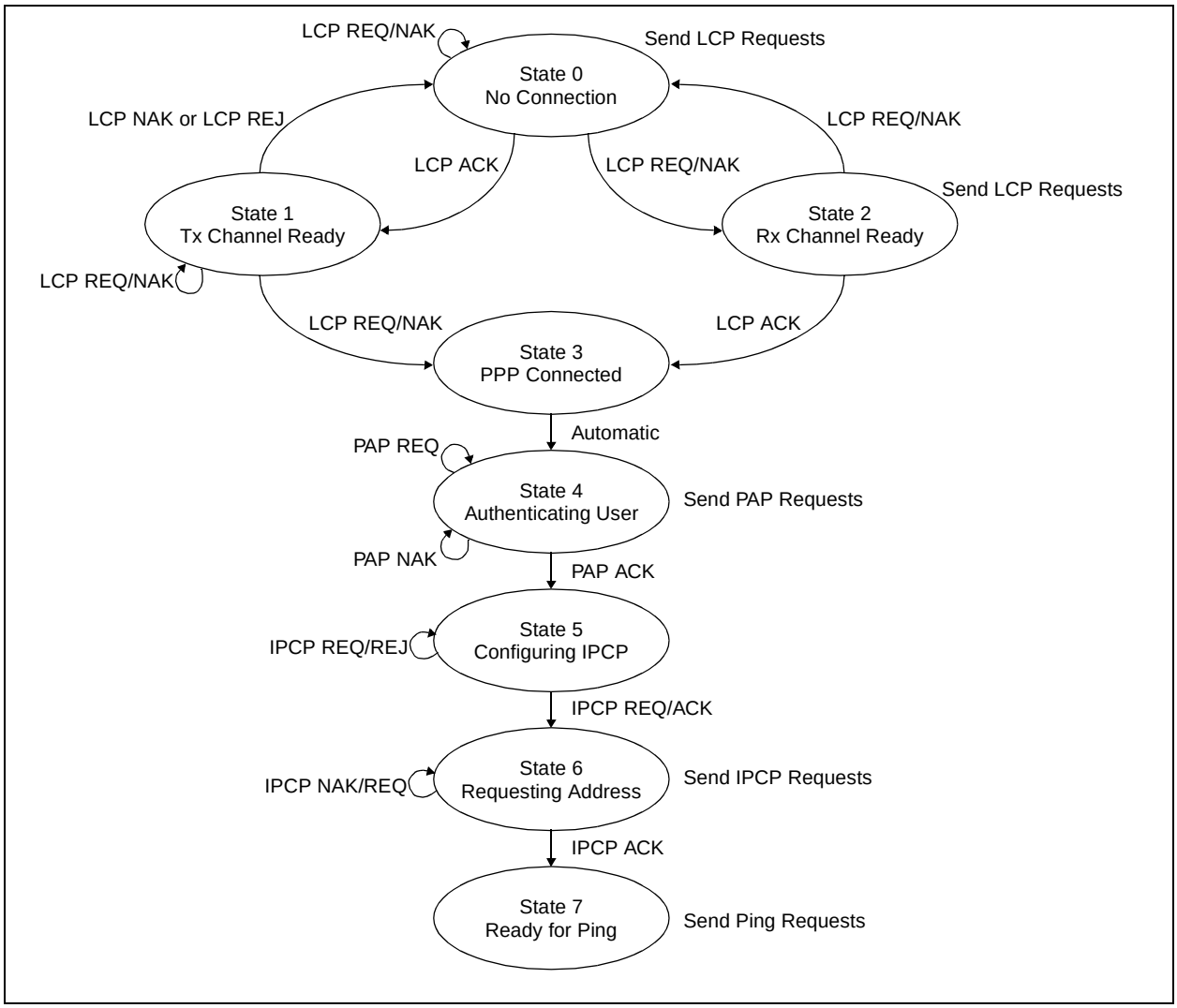
In state four a PAP request with the User ID and password is sent once a second. The acknowledgement of the password moves the algorithm into state five. When the algorithm acknowledges the server's IPCP options it moves into state six. In state six the algorithm requests IP address 0.0.0.0 once a second. The server should reply with a NAK packet containing the correct address to move the algorithm into it final state 7. Here it flashes out the IP address on the status LED and then pings the hardcoded host IP address every 30 seconds. After the first good ping reply, it turns off the power unless the power button is still pressed.

The `MakePacket` routine creates an outgoing packet in the transmit buffer. Every loop of the state machine checks if the serial transmitter is ready for another character. If the transmit buffer is empty, it sends the next character. On the last character it marks the buffer empty and sends an extra 0x7E to mark the end of the packet.

Every loop of the state machine also checks the serial receiver for characters from the modem or Internet server. Characters that are sent using the previously described `}` escape sequence are immediately converted back to the original character. The CRC checksum is also calculated as the bytes come in so that packets longer than the buffer can still pass the CRC.

The `OptionTest` routine is used to test the receive buffer for whatever options the server is requesting. It takes a 16-bit option parameter, where each bit represents an option from 1 to 16 - with 16 being the MSB. If a bit is set, then its corresponding option can be accepted. If the server requests options that are not allowed by the option parameter, then the subroutine returns a zero and deletes the options that were allowed. This way a REJ packet can be sent to tell the server which options to drop. If it is an LCP packet with option three set to CHAP then the subroutine returns a one and deletes the options that passed. This way a NAK packet can be sent to tell the server to switch to PAP. In all other cases the subroutine returns a value greater than one and leaves the receive buffer unchanged.

FIGURE 12: PPP NEGOTIATION STATE MACHINE



CONCLUSION

This algorithm is a little taste of what is possible with a PICmicro MCU, you will likely use this information as a basis for even more powerful Internet applications. Just remember to only make small changes to working code and test them before making the next change.

This information is quite technical, so don't give up if you are not already TCP/IP savvy. Remember that all the so called experts had to learn it at some time too. Read a book like TCP/IP Illustrated Volume 1 by Richard Stevens and the referenced RFCs, then compile the code and analyze lots of packets. Start your experimenting slowly with a relatively easy task like adding support for replying to Traceroute requests. Here's a hint: test the TTL on valid IP packets received to trigger sending an ICMP error packet.

This tutorial was meant to encourage the development of tiny Internet interfaces and not to replace or override the established Internet standards documents.

The prototype does what I needed it to do but there are many areas in which it could be improved upon, such as size, speed, power requirements, RAM usage, supported protocols, and more universal PPP negotiations. The possibilities are only limited by your imagination and creativity in overcoming the obstacles.

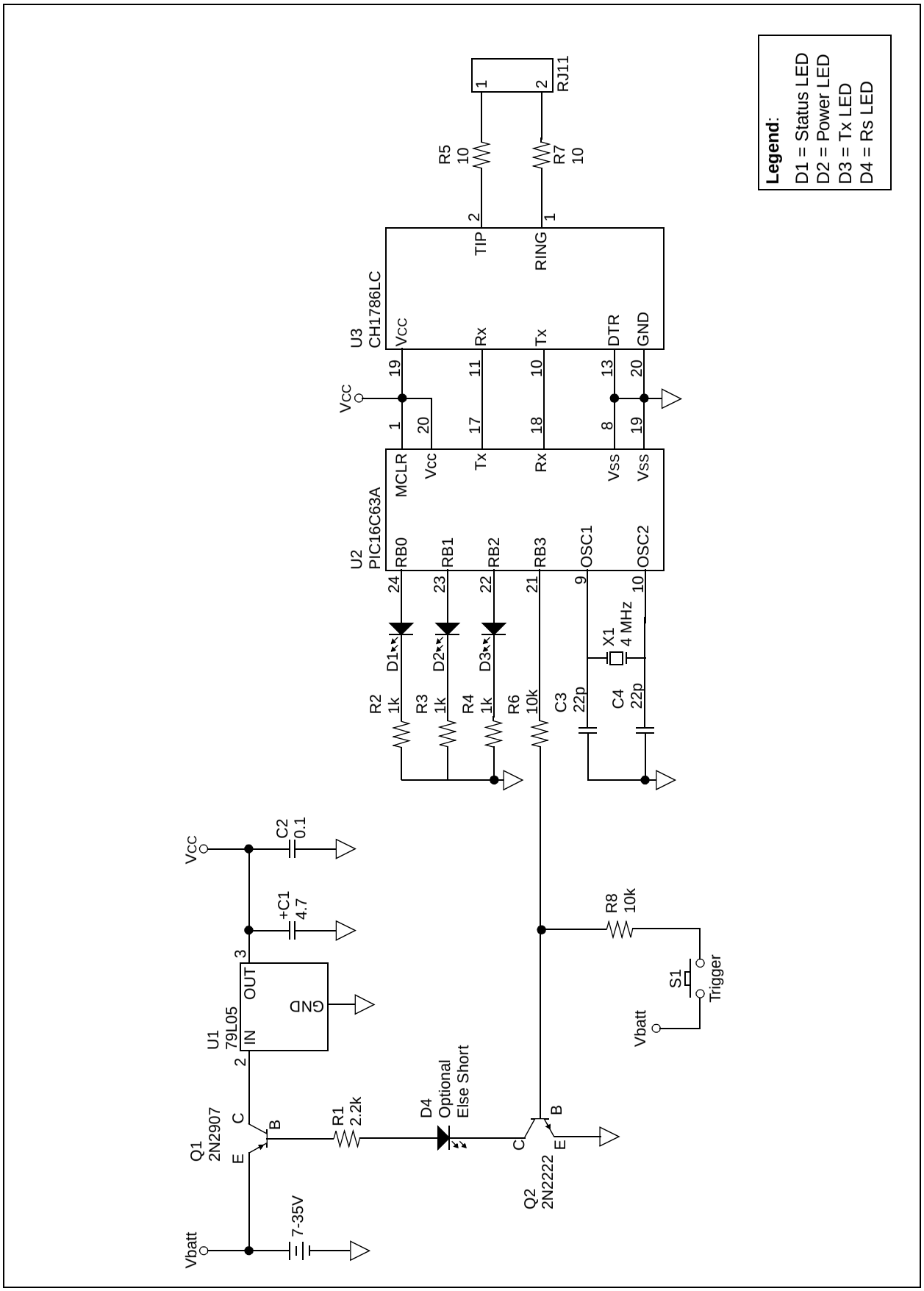
TABLE 5: REFERENCES

W.R. Stevens, *TCP/IP Illustrated*, Vol. 1, Addison Wesley, Reading, MA, 1994
RFC 0768 User Datagram Protocol. J. Postel. Aug-28-1980.
RFC 0791 Internet Protocol. J. Postel. Sep-01-1981.
RFC 0792 Internet Control Message Protocol. J. Postel. Sep-01-1981.
RFC 0793 Transmission Control Protocol. J. Postel. Sep-01-1981.
RFC 0867 Daytime Protocol. J. Postel. May-01-1983.
RFC 0950 Internet Standard Subnetting Procedure. J.C. Mogul, J. Postel. Aug-01-1985.
RFC 1055 Nonstandard for transmission of IP datagrams over serial lines: SLIP. J.L. Romkey. Jun-01-1988.
RFC 1071 Computing the Internet checksum. R.T. Braden, D.A. Borman, C. Partridge. Sep-01-1988.
RFC 1122 Requirements for Internet hosts - communication layers. R.T. Braden. Oct-01-1989.
RFC 1123 Requirements for Internet hosts - application and support. R.T. Braden. Oct-01-1989.
RFC 1144 Compressing TCP/IP headers for low-speed serial links. V. Jacobson. Feb-01-1990.
RFC 1157 Simple Network Management Protocol (SNMP). J.D. Case, M. Fedor, M.L. Schoffstall, C. Davin. May-01-1990.
RFC 1332 The PPP Internet Protocol Control Protocol (IPCP). G. McGregor.May 1992.
RFC 1334 PPP Authentication Protocols. B. Lloyd, W. Simpson. October 1992.
RFC 1350 The TFTP Protocol (Revision 2). K. Sollins. July 1992.
RFC 1547 Requirements for an Internet Standard Point-to-Point Protocol. D. Perkins. December 1993.
RFC 1570 PPP LCP Extensions. W. Simpson. January 1994.
RFC 1661 The Point-to-Point Protocol (PPP). W. Simpson, Editor. July 1994.
RFC 1662 PPP in HDLC-like Framing. W. Simpson, Editor. July 1994.
RFC 1663 PPP Reliable Transmission. D. Rand. July 1994.
RFC 1700 Assigned Numbers. J. Reynolds, J. Postel. October 1994.
RFC 1962 The PPP Compression Control Protocol (CCP). D. Rand. June 1996.
RFC 1989 PPP Link Quality Monitoring. W. Simpson. August 1996.
RFC 1994 PPP Challenge Handshake Authentication Protocol (CHAP). W. Simpson. August 1996.
James Carlson, PPP Design and Debugging, Addison Wesley, Reading, MA, 1997

RFCs available online from sites like: http://www.cis.ohio-state.edu/hypertext/information/rfc.html

Assigned PPP Numbers: ftp://ftp.isi.edu/in-notes/iana/assignments/ppp-numbers

FIGURE 13: SCHEMATIC



APPENDIX A: SOURCE CODE

SOFTWARE LICENSE AGREEMENT

The software supplied herewith by Microchip Technology Incorporated (the “Company”) for its PICmicro® Microcontroller is intended and supplied to you, the Company’s customer, for use solely and exclusively on Microchip PICmicro Microcontroller products.

The software is owned by the Company and/or its supplier, and is protected under applicable copyright laws. All rights are reserved. Any use in violation of the foregoing restrictions may subject the user to criminal sanctions under applicable laws, as well as to civil liability for the breach of the terms and conditions of this license.

THIS SOFTWARE IS PROVIDED IN AN “AS IS” CONDITION. NO WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE APPLY TO THIS SOFTWARE. THE COMPANY SHALL NOT, IN ANY CIRCUMSTANCES, BE LIABLE FOR SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES, FOR ANY REASON WHATSOEVER.

APPENDIX A: SOURCE CODE

```
////////////////////////////////////////////////////////////////////////////////
//
//    PING.C    version 1.10    July 29/99     (C)opyright by Microchip Technology Inc.
//
////////////////////////////////////////////////////////////////////////////////
//
// For more documentation read the Microchip Application Note 724
// This code is ready to compile with the HiTech C compiler demo for the PIC16C63A.
//
// You will need these additional things to make this code work:
//
//    - the simple hardware described in application note
//
//    - an Internet account with PPP dialup access (not compatible with all ISPs)
//
//    - replace 5551234 with your ISP's phone number in the line like this
//         if (sendwait("5551234\r","NNECT",3000)) {
//
//    - replace userid with your account userid in the line like this:
//         if (sendwait("userid\r","word:",200))
//
//    - replace password with your account password in the line like this:
//         if (sendwait("password\r","tion:",1000))
//
//    - replace the entire string in the line like this:
//         MakePacket(PAP,REQ,number,"\x14\x06userid\x08password");
//
//         C converts the \x## in the string to a character with that ASCII value
//         ## is a hexadecimal value, so the following character cannot be
//         if the next character is 0-9 or A-F or a-f then it will confuse the compiler
//         the solution is to convert the next characters to \x## until a non hex char
//         if in doubt look at the assembly output from the compiler
//         replace the userid with yours and the \x06 with your userid length
//         replace the password with yours and the \x08 with your password length
//         replace the first value in the string, it must be the string length plus 4
//
// Once login is working you should also change the IP address of the Internet host to ping
// if you can not ping 207.161.117.67 with your PC this code will not work either
//    It is CF.A1.75.43, the characters 2 to 5, in the string in the line like this:
//       MakePacket(IP,0,1,"\x10\xCF\xA1\x75\x43\x8\x0\xF7\xFE\x0\x1\x0\x0");
//    Convert the address you want to hexadecimal and replace the four values.
//
// Make sure the power-on reset and brownout detect config bits are enabled
//
////////////////////////////////////////////////////////////////////////////////

// Defines for Internet constants
#define  REQ        1                  // Request options list for PPP negotiations
#define  ACK        2                  // Acknowledge options list for PPP negotiations
#define  NAK        3                  // Not acknowledged options list for PPP negotiations
#define  REJ        4                  // Reject options list for PPP negotiations
#define  TERM       5                  // Termination packet for LCP to close connection
#define  IP         0x0021             // Internet Protocol packet
#define  IPCP       0x8021             // Internet Protocol Configure Protocol packet
#define  CCP        0x80FD             // Compression Configure Protocol packet
#define  LCP        0xC021             // Link Configure Protocol packet
#define  PAP        0xC023             // Password Authentication Protocol packet

#define  MaxRx      46                 // Maximum size of receive buffer
#define  MaxTx      46                 // Maximum size of transmit buffer

unsigned char addr1, addr2, addr3, addr4;// Assigned IP address
unsigned int rx_ptr, tx_ptr, tx_end;     // pointers into buffers
unsigned int checksum1, checksum2;       // Rx and Tx checksums
```

```
unsigned char number;                   // Unique packet id

#include <pic1663.h>                    // Defines specific to this processor

#define serial_init()         RCSTA=0x90;TXSTA=0x24;SPBRG=103// Set up serial port
#define serial_tx_ready()     TXIF                    // Transmitter empty
#define serial_send(a)        TXREG=a                 // Transmit char a
#define serial_rx_ready()     RCIF                    // Receiver full
#define serial_get()          RCREG                   // Receive char
#define serial_error()        OERR                    // USART error
#define serial_fix()          {CREN=0;CREN=1;}        // Clear error

unsigned int TIME;                                    // 10 milliseconds counter
#define TIME_SET(a) TIME=a                            // Set 10 millisecond counter to value ‘a’
bank1 unsigned char tx_str[MaxRx+1];                  // Transmitter buffer
bank1 unsigned char rx_str[MaxTx+1];                  // Receiver buffer

// Process all the interrupts in the PIC here
static void interrupt isr(void) {
   if (T0IF) {                          // Timer overflow interrupt?
      TMR0 = 100;                       // Set to overflow again in 10ms @ 4MHz
      T0IF = 0;                         // Clear overflow interrupt flag
      TIME++;                           // Increment 10 ms counter
   }
}

// Add next character to the CRC checksum for PPP packets
unsigned int calc(unsigned int c) {
   char i;                              // Just a loop index
   c &= 0xFF;                           // Only calculate CRC on low byte
   for (i=0;i<8;i++) {                  // Loop eight times, once for each bit
      if (c&1) {                        // Is bit high?
         c /= 2;                        // Position for next bit
         c ^= 0x8408;                   // Toggle the feedback bits
      } else c /= 2;                    // Just position for next bit
   }                                    // This routine would be best optimized in assembly
   return c;                            // Return the 16 bit checksum
}

// Add character to the new packet
void add(unsigned char c) {
   checksum2 = calc(c^checksum2) ^ (checksum2/256);  // Add CRC from this char to running total
   tx_str[tx_ptr] = c;                               // Store character in the transmit buffer
   tx_ptr++;                                         // Point to next empty spot in buffer
}

// Create packet of type, code, length, and data string specified
//   packet is the type, like LCP or IP
//   code is the LCP type of packet like REQ, not used for IP packets
//   num is the packet ID for LCP, or the IP data type for IP packets
//   *str is the packet data to be added after the header
//   returns the packet as a string in tx_str
void MakePacket(unsigned int packet, unsigned char code, unsigned char num, const unsigned char
*str) {
   unsigned int length;                        // Just a dual use temp variable
   tx_ptr = 1;                                 // Point to second character in transmit buffer
   tx_str[0] = ‘ ‘;                            // Set first character to a space for now
   checksum2 = 0xFFFF;                         // Initialize checksum
   add(0xFF);                                  // Insert PPP header OxFF
   add(3);                                     // Insert PPP header 0x03
   add(packet/256);                            // Insert high byte of protocol field
   add(packet&255);                            // Insert low byte of protocol field
   if (packet==IP) {                           // If Internet Protocol
      add(0x45);                               // Insert header version and length
      add(0);                                  // Insert type of service
      add(0);                                  // Insert total packet length high byte
```

```
      add((*str)+12);                          // Insert total packet length low byte
      add(0x88);                               // Insert identification high byte
      add(0x10);                               // Insert identification low byte
      add(0x40);                               // Insert flags and fragment offset
      add(0);                                  // Insert rest of fragment offset
      add(127);                                // Insert time to live countdown
      add(num);                                // insert the protocol field
      length = 0x45+0x88+0x40+127+addr1+addr3+str[1]+str[3];  // high byte checksum
      packet = *str + 12 + 0x10 + num + addr2 + addr4 + str[2] + str[4];
                                               // low byte checksum
      packet += length/256;                    // make 1's complement
      length = (length&255) + packet/256;      // by adding low carry to high byte
      packet = (packet&255) + length/256;      // and adding high carry to low byte
      length += packet/256;                    // fix new adding carries
      add(~length);                            // Insert 1's complement checksum high byte
      add(~packet);                            // Insert 1's complement checksum low byte
      add(addr1);                              // Insert the 4 bytes of this login's IP address
      add(addr2);
      add(addr3);
      add(addr4);
      length = *str - 4;                       // save the number of following data bytes
      str++;                                   // point to the first data byte
   } else {
      add(code);                               // Insert packet type, like REQ or NAK
      add(num);                                // Insert packet ID number
      add(0);                                  // Insert most significant byte of length
      length = *str - 3;                       // point to the first data byte
   }
   while (length) {                            // copy the whole string into packet
      length--;                                // decrement packet length
      add(*str);                               // add current character to packet
      str++;                                   // point to next character
   }
   length = ~checksum2;                        // invert the checksum
   add(length&255);                            // Insert checksum msb
   add(length/256);                            // Insert checksum lsb
   tx_end=tx_ptr;                              // Set end of buffer marker to end of packet
   tx_ptr = 0;                                 // Point to the beginning of the packet
}

// Test the option list in packet for valid passwords
//   option is the 16 bit field, where a high accepts the option one greater than the
//   bit #
//   returns 2 for LCP NAK, 1 is only correct fields found, and zero means bad options
//   return also modifies RX_STR to list unacceptable options if NAK or REJ required
unsigned char TestOptions(unsigned int option){
   unsigned int size;                          // size is length of option string
   unsigned ptr1 = 8,                          // ptr1 points data insert location
            ptr2 = 8;                          // ptr2 points to data origin
   char pass = 3;                              // pass is the return value
   size = rx_str[7]+4;                         // size if length of packet
   if (size>MaxRx) size=MaxRx;                 // truncate packet if larger than buffer
   while (ptr1<size) {                         // scan options in receiver buffer
      if (rx_str[ptr1]==3 && rx_str[ptr1+2]!=0x80  && rx_str[2]==0xc2)
         pass&=0xfd;                           // found a CHAP request, mark for NAK
      if (!((1<<(rx_str[ptr1]-1))&option))
         pass=0;                               // found illegal options, mark for REJ
      ptr1 += rx_str[ptr1+1];                  // point to start of next option
   }
   if (!(pass&2)) {                            // If marked for NAK or REJ
      if (pass&1) {                            // save state for NAK
         option=0xfffb;
      }
      for (ptr1=8; ptr1<size;) {
         if (!((1<<(rx_str[ptr1]-1))&option)) {// if illegal option
            for (pass=rx_str[ptr1+1]; ptr1<size && pass; ptr1++) {  // move option
```

```
               rx_str[ptr2]=rx_str[ptr1];      // move current byte to new storage
               ptr2++;                         // increment storage pointer
               pass--;                         // decrement number of characters
            }
         } else {
            ptr1+=rx_str[ptr1+1];              // point to next option
         }
      }
      rx_str[7] = ptr2-4;                      // save new option string length
      pass=0;                                  // restore state for REJ
      if (option==0xfffb) pass=1;              // restore state for NAK
   }
   return pass;
}

// Send a string and loop until wait string arrives or it times out
//  send is the string to transmit
//  wait is the string to wait for
//  timeout is in multiples of 10 milliseconds
//  addr1 is used to control the status LED, 0=off, 1=flash, 2=on
//  returns 0 if timeout, returns 1 if wait string is matched
char sendwait(const char *send, const char *wait, unsigned int timeout) {
   addr2=addr3=0;
   for (TIME_SET(0); TIME<timeout; ) {            // loop until time runs out
      if (!addr1) PORTB&=0xFB;                    // if addr1=0 turn off status LED
      else if (addr1==1) {                        // if addr1=1 flash status LED
         if (TIME&4) PORTB&=0xFB;                 // flash period is 8 x 10ms
         else PORTB|=4;
      } else PORTB|=4;                            // if addr1>1 turn on status LED
      if (serial_rx_ready()) {                    // is there an incoming character
         PORTB|=1;                                // turn on the Rx LED
         addr4 = serial_get();                    // get character
         if (serial_error()) serial_fix();        // clear serial errors
         if (wait[addr2]==addr4) addr2++;         // does char match wait string
         else addr2=0;                            // otherwise reset match pointer
         PORTB&=0xFE;                             // turn off the Rx LED
         if (!wait[addr2]) return 1;              // finished if string matches
      } else if (send[addr3] && (serial_tx_ready())) { // if char to send and Tx ready
         if (send[addr3]=='|') {                  // if pause character
            if (TIME>100) {                       // has 1 second expired yet?
               TIME_SET(0);                       // if yes clear timer
               addr3++;                           // and point to next character
            }
         } else {
            PORTB|=2;                             // turn on Tx LED
            TIME_SET(0);                          // clear timer, timeout starts after last char
            serial_send(send[addr3]);             // send the character
            addr3++;                              // point to next char in tx string
         }
         PORTB&=0xFD;                             // turn off Tx LED
         if (!send[addr3] && !(*wait))
            return 1;                             // done if end of string and no wait string
      }
   }
   return 0;                                      // return with 0 to indicate timeout
}

void flash(void) {                                // flash all LEDs if catastrophic failure
   for (TIME_SET(0);;) {
      if (TIME&8) PORTB|=0x07;                    // flash period is 16 x 10ms
      else PORTB&=0xF8;
      if (TIME>3000) PORTB&=0xF7;                 // after 30 seconds turn off the power
   }
}

void pulse(unsigned char data) {                  // pulse Status LED with IP address
```

```
   TIME_SET(0);
   for(number=0;number<9;) {                    // pulse out 8 address bits and a blank
      if (TIME<100) PORTB&=0xFB;                // turn off Status LED between bits
      else if (number<8) PORTB|=4;              // start each address bit here
      if (TIME>200 || (!(data&0x80) && TIME>120)) { // end of bit?
         TIME_SET(0);                           // yes, then restart timer for next bit
         number++;                              // increment bit counter
         data<<=1;                              // position address to send next bit
      }
   }
}

// The main loop, login script, PPP state machine, and ping transponder
void main(void) {
   signed int c;                          // serial character received
   unsigned int packet = 0;               // Type of the last received packet, reused as temp
   unsigned char state = 0;               // PPP negotiation state, from dialing=0 to done=6
   unsigned char extended = 0;            // flag if last character was an escape sequence

   PORTA=0;
   PORTB=0;                               // Turn off power supply, turn off LEDs
   PORTC=0;
   TRISA=0;                               // Turn all I/O into outputs
   TRISB=0x00;
   TRISC=0xC0;
   OPTION=0x85;                           // Set up TIMER 0 for millisecond counting
   INTCON=0xA0;

   serial_init();                         // Initialize serial port to 2400 baud format N81
   TIME_SET(0);
   while (TIME<25);                       // 250 millisecond delay to prevent false power up
   PORTB=8;                               // Turn on the power so user can release power button

   for(number=1;;number++) {              // Redial indefinitely every 30 seconds
      if (number==10) PORTB&=0xF7;        // Turn off power if dialing fails
      addr1=0;                            // Set flag to keep the Status LED off
      if(!sendwait("|+++|\rath\r|atz\r|at&fs11=55\r|atdt","atdt",3000))// Init modem
         flash();
      addr1=1;                            // Set flag to flash Status LED
      // Modify this line with your ISP phone number
      if (sendwait("5551234\r","NNECT",3000)) {
         addr1=2;                         // Set flag to keep the Status LED on
         if (sendwait("","": ",1000))     // Wait for user id prompt
            if (sendwait("\x7e\xff\x7d\x23\x08\x08\x08\x08","~~",1000))
               break;                     // Start PPP
            else {                        // Fallback to script if required
               if (sendwait("userid\r","word:",200))// Modify these lines as described
                  if (sendwait("password\r","tion:",1000))
                     if (!sendwait("ppp\r","IP address",200))
                       // Modify is start PPP command is not ppp or 2
                       sendwait("2\r","IP address",200);
            }
         break;
      }
   }

   // State machine loop until successful ping or PPP negotiation timeout
   for (TIME_SET(0);;) {
      if (TIME>7000 || number>20) PORTB&=0xF7;
      if (serial_rx_ready()) {            // Incoming character?
         PORTB ^=1;                       // Turn on Rx LED
         c = serial_get();                // get the character
         if (serial_error()) serial_fix();// clear Rx errors
         if (c == 0x7E) {                 // start or end of a packet
            if (rx_ptr && (checksum1==0xF0B8))
               packet = rx_str[2]*256 + rx_str[3]; // if CRC passes accept packet
```

```
      extended &= 0x7E;            // clear escape character flag
      rx_ptr = 0;                  // get ready for next packet
      checksum1 = 0xFFFF;          // start new checksum
   } else if (c == 0x7D) {         // if tilde character set escape flag
      extended |= 1;
   } else {
      if (extended&1) {            // if escape flag
         c ^= 0x20;                // recover next character
         extended &= 0xFE;         // clear Rx escape flag
      }
      if (rx_ptr==0 && c!=0xff) rx_str[rx_ptr++] = 0xff; // uncompress PPP header
      if (rx_ptr==1 && c!=3) rx_str[rx_ptr++] = 3;
      if (rx_ptr==2 && (c&1)) rx_str[rx_ptr++] = 0;
      rx_str[rx_ptr++] = c;        // insert character in buffer
      if (rx_ptr>MaxRx) rx_ptr = MaxRx;// Inc pointer up to end of buffer
      checksum1 = calc(c^checksum1) ^ (checksum1/256); // calculate CRC checksum
   }
   PORTB&=0xFE;                    // turn off Status LED
} else if (tx_end && (serial_tx_ready())) { // Data to send and Tx empty?
   PORTB|=2;                       // turn on Tx LED
   c = tx_str[tx_ptr];             // get character from buffer
   if (tx_ptr==tx_end) {           // was it the last character
      tx_end=0;                    // mark buffer empty
      c='~';                       // send tilde character last
      PORTB&=0xFD;                 // turn off Tx LED
   } else if (extended&2) {        // sending escape sequence?
      c^=0x20;                     // yes then convert character
      extended &= 0xFD;            // clear Tx escape flag
      tx_ptr++;                    // point to next char
   } else if (c<0x20 || c==0x7D || c==0x7E) { // if escape sequence required?
      extended |= 2;               // set Tx escape flag
      c = 0x7D;                    // send escape character
   } else {
     if (!tx_ptr) c='~';           // send ~ if first character of packet
     tx_ptr++;
   }
   serial_send(c);                 // Put character in transmitter
}

if (packet == LCP) {
   switch (rx_str[4]) {            // Switch on packet type
      case REQ:
         state &= 0xfd;            // clear remote ready state bit
         if (c=TestOptions(0x00c6)) {// is option request list OK?
            if (c>1) {
               c = ACK;            // ACK packet
               if (state<3) state |= 2;// set remote ready state bit
            } else {
               rx_str[10]=0xc0;    // else NAK password authentication
               c = NAK;
            }
         } else {                  // else REJ bad options
            c = REJ;
         }
         TIME_SET(0);
         MakePacket(LCP,c,rx_str[5],rx_str+7); // create LCP packet from Rx buffer
         break;
      case ACK:
         if (rx_str[5]!=number) break;// does reply id match the request
         if (state<3) state |= 1;  // Set the local ready flag
         break;
      case NAK:
         state &= 0xfe;            // Clear the local ready flag
         break;
      case REJ:
         state &= 0xfe;            // Clear the local ready flag
```

```
            break;
         case TERM:
            break;
      }
      if (state==3) state = 4;        // When both ends ready, go to state 4
   } else if (packet == PAP) {
      switch (rx_str[4]) {            // Switch on packet type
         case REQ:
            break;                    // Ignore incoming PAP REQ
         case ACK:
            state = 5;                // PAP ack means this state is done
            break;
         case NAK:
            break;                    // Ignore incoming PAP NAK
      }
   } else if (packet == IPCP) {
      switch (rx_str[4]) {            // Switch on packet type
         case REQ:
            if (TestOptions(0x0004)) {// move to next state on ACK
               c = ACK;
               state = 6;
            } else {                  // otherwise reject bad options
               c = REJ;
            }
            MakePacket(IPCP,c,rx_str[5],rx_str+7);
            break;                    // Create IPCP packet from Rx buffer
         case ACK:
            if (rx_str[5]==number) {  // If IPCP response id matches request id
               state = 7;             // Move into final state
               pulse(addr1);          // Pulse Status LED to show the
               pulse(addr2);          // IP address
               pulse(addr3);
               pulse(addr4);
               PORTB|=4;              // Turn on Status LED after pulsing
               TIME_SET(5800);        // Move timer ahead for quicker PING
            }
            break;
         case NAK:                    // This is where we get our address
            addr1 = rx_str[10];
            addr2 = rx_str[11];       // Store address for use in IP packets
            addr3 = rx_str[12];
            addr4 = rx_str[13];
            MakePacket(IPCP,REQ,rx_str[5],rx_str+7);
            break;                    // Make IPCP packet from Rx buffer
         case REJ:
            break;                    // Ignore incoming IPCP REJ
         case TERM:
            break;                    // Ignore incoming IPCP TERM
      }
   } else if (packet == IP) {
      if (state<7 || (rx_str[19]==addr4 && rx_str[18]==addr3 &&
                      rx_str[17]==addr2 && rx_str[16]==addr1)) {
               // ignore echoed packets from our address or before we reach state 7
               // may power down here because echoes are good indications that modem
               // connection hung-up
               // This would be a good place to insert a traceroute test and
               // response
      } else if (rx_str[13]==1) {     // IP packet with ICMP payload
         if (rx_str[24]==8) {         // Received PING request
            rx_str[20]=rx_str[16];    // Copy 4 origin address bytes to
                                      // destination address
            rx_str[21]=rx_str[17];
            rx_str[22]=rx_str[18];
            rx_str[23]=rx_str[19];
            rx_str[19]=16;            // Length of IP address(4) + ping protocol(8) + 4
            rx_str[24]=0;             // Change received ping request(8) to ping reply(0)
```

```
            packet = rx_str[28]+rx_str[30];// Calculate 1's comp checksum
            rx_str[26] = packet&255;
            rx_str[27] = packet/256;
            packet = rx_str[27]+rx_str[29]+rx_str[31];
            rx_str[27] = packet&255;
            packet = packet/256 + rx_str[26];
            rx_str[26] = packet&255;
            rx_str[27] += packet/256;
            rx_str[26] = ~rx_str[26]; // Invert the checksum bits
            rx_str[27] = ~rx_str[27];
            MakePacket(IP,0,1,rx_str+19);// Make IP packet from modified Rx buffer
         } else if (rx_str[24]==0) {  // Received PING reply
            if ((rx_str[28]|rx_str[30]|rx_str[31])+rx_str[29]==1)
               PORTB&=0xF7;           // Turn off the power after successful ping
         }
      }
   } else if (packet == CCP) {
      switch (rx_str[4]) {            // If CCP response id matches request id
         case REQ:
            c = REJ;
            if (TestOptions(0x0004)) c = ACK; // ACK option 3 only, REJ anything else
            MakePacket(CCP,c,rx_str[5],rx_str+7);// Create CCP ACK or REJ packet
                                       // from Rx buffer
      }
   } else if (packet) {               // Ignore any other received packet types
   } else if (!tx_end && (state==0 || state==2) && TIME>100) {
                                       // Once a second try negotiating LCP
      number++;                        // Increment Id to make packets unique
      TIME_SET(0);                     // Reset timer
      MakePacket(LCP,REQ,number,"\x0E\x02\x06\x00\x0A\x00\x00\x07\x02\x08\x02");
                                       // Request LCP options 2,7,8
   } else if (!tx_end && state == 4 && TIME>100) {
                                       // Once a second try negotiating password
      TIME_SET(0);                     // Reset timer
      number++;
      // format like printf("%c%c%s%c%s",strlen(name)+strlen(password)+6,
      //strlen(name),name,strlen(password),password);
      // Modify this line as described above
      MakePacket(PAP,REQ,number,"\x14\x06\xuserid\x08password");
   } else if (!tx_end && state == 6 && TIME>100) {
                                       // Once a second try negotiating IPCP
      number++;                        // Increment Id to make packets unique
      TIME_SET(0);                     // Reset timer
      MakePacket(IPCP,REQ,number,"\xA\x3\x6\x0\x0\x0\x0");
                                       // Request IPCP option 3 with addr 0.0.0.0
   } else if (!tx_end && state == 7 && TIME>3000) { // Every 30 seconds do a ping
      TIME_SET(0);                     // Reset timer
      number++;                        // Increment ping count
      MakePacket(IP,0,1,"\x10\xCF\xA1\x75\x43\x8\x0\xF7\xFE\x0\x1\x0\x0");
                                       // Ping 207.161.117.67
   }
   packet = 0;                         // Indicate that packet is processed
  }
}
```

NOTES:

NOTES:

NOTES:

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
Microchip Technology Inc.
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-786-7200 Fax: 480-786-7277
Technical Support: 480-786-7627
Web Address: http://www.microchip.com

Atlanta
Microchip Technology Inc.
500 Sugar Mill Road, Suite 200B
Atlanta, GA 30350
Tel: 770-640-0034 Fax: 770-640-0307

Boston
Microchip Technology Inc.
5 Mount Royal Avenue
Marlborough, MA 01752
Tel: 508-480-9990 Fax: 508-480-8575

Chicago
Microchip Technology Inc.
333 Pierce Road, Suite 180
Itasca, IL 60143
Tel: 630-285-0071 Fax: 630-285-0075

Dallas
Microchip Technology Inc.
4570 Westgrove Drive, Suite 160
Addison, TX 75248
Tel: 972-818-7423 Fax: 972-818-2924

Dayton
Microchip Technology Inc.
Two Prestige Place, Suite 150
Miamisburg, OH 45342
Tel: 937-291-1654 Fax: 937-291-9175

Detroit
Microchip Technology Inc.
Tri-Atria Office Building
32255 Northwestern Highway, Suite 190
Farmington Hills, MI 48334
Tel: 248-538-2250 Fax: 248-538-2260

Los Angeles
Microchip Technology Inc.
18201 Von Karman, Suite 1090
Irvine, CA 92612
Tel: 949-263-1888 Fax: 949-263-1338

New York
Microchip Technology Inc.
150 Motor Parkway, Suite 202
Hauppauge, NY 11788
Tel: 516-273-5305 Fax: 516-273-5335

San Jose
Microchip Technology Inc.
2107 North First Street, Suite 590
San Jose, CA 95131
Tel: 408-436-7950 Fax: 408-436-7955

AMERICAS (continued)

Toronto
Microchip Technology Inc.
5925 Airport Road, Suite 200
Mississauga, Ontario L4V 1W1, Canada
Tel: 905-405-6279 Fax: 905-405-6253

ASIA/PACIFIC

Hong Kong
Microchip Asia Pacific
Unit 2101, Tower 2
Metroplaza
223 Hing Fong Road
Kwai Fong, N.T., Hong Kong
Tel: 852-2-401-1200 Fax: 852-2-401-3431

Beijing
Microchip Technology, Beijing
Unit 915, 6 Chaoyangmen Bei Dajie
Dong Erhuan Road, Dongcheng District
New China Hong Kong Manhattan Building
Beijing 100027 PRC
Tel: 86-10-85282100 Fax: 86-10-85282104

India
Microchip Technology Inc.
India Liaison Office
No. 6, Legacy, Convent Road
Bangalore 560 025, India
Tel: 91-80-229-0061 Fax: 91-80-229-0062

Japan
Microchip Technology Intl. Inc.
Benex S-1 6F
3-18-20, Shinyokohama
Kohoku-Ku, Yokohama-shi
Kanagawa 222-0033 Japan
Tel: 81-45-471- 6166 Fax: 81-45-471-6122

Korea
Microchip Technology Korea
168-1, Youngbo Bldg. 3 Floor
Samsung-Dong, Kangnam-Ku
Seoul, Korea
Tel: 82-2-554-7200 Fax: 82-2-558-5934

Shanghai
Microchip Technology
RM 406 Shanghai Golden Bridge Bldg.
2077 Yan'an Road West, Hong Qiao District
Shanghai, PRC 200335
Tel: 86-21-6275-5700 Fax: 86 21-6275-5060

ASIA/PACIFIC (continued)

Singapore
Microchip Technology Singapore Pte Ltd.
200 Middle Road
#07-02 Prime Centre
Singapore 188980
Tel: 65-334-8870 Fax: 65-334-8850

Taiwan, R.O.C
Microchip Technology Taiwan
10F-1C 207
Tung Hua North Road
Taipei, Taiwan, ROC
Tel: 886-2-2717-7175 Fax: 886-2-2545-0139

EUROPE

United Kingdom
Arizona Microchip Technology Ltd.
505 Eskdale Road
Winnersh Triangle
Wokingham
Berkshire, England RG41 5TU
Tel: 44 118 921 5858 Fax: 44-118 921-5835

Denmark
Microchip Technology Denmark ApS
Regus Business Centre
Lautrup hoj 1-3
Ballerup DK-2750 Denmark
Tel: 45 4420 9895 Fax: 45 4420 9910

France
Arizona Microchip Technology SARL
Parc d'Activite du Moulin de Massy
43 Rue du Saule Trapu
Batiment A - ler Etage
91300 Massy, France
Tel: 33-1-69-53-63-20 Fax: 33-1-69-30-90-79

Germany
Arizona Microchip Technology GmbH
Gustav-Heinemann-Ring 125
D-81739 München, Germany
Tel: 49-89-627-144 0 Fax: 49-89-627-144-44

Italy
Arizona Microchip Technology SRL
Centro Direzionale Colleoni
Palazzo Taurus 1 V. Le Colleoni 1
20041 Agrate Brianza
Milan, Italy
Tel: 39-39-65791-1 Fax: 39-39-6899883

09/21/99

Microchip received ISO 9001 Quality System certification for its worldwide headquarters, design, and wafer fabrication facilities in January 1997. Our field-programmable PICmicro® 8-bit MCUs, KEELOQ® code hopping devices, Serial EEPROMs, related specialty memory products and development systems conform to the stringent quality standards of the International Standard Organization (ISO).