Good news, everyone!

mTCP

Developer Documentation

Version: 2020-03-07

M. Brutman (mbbrutman@gmail.com)
http://www.brutman.com/mTCP/

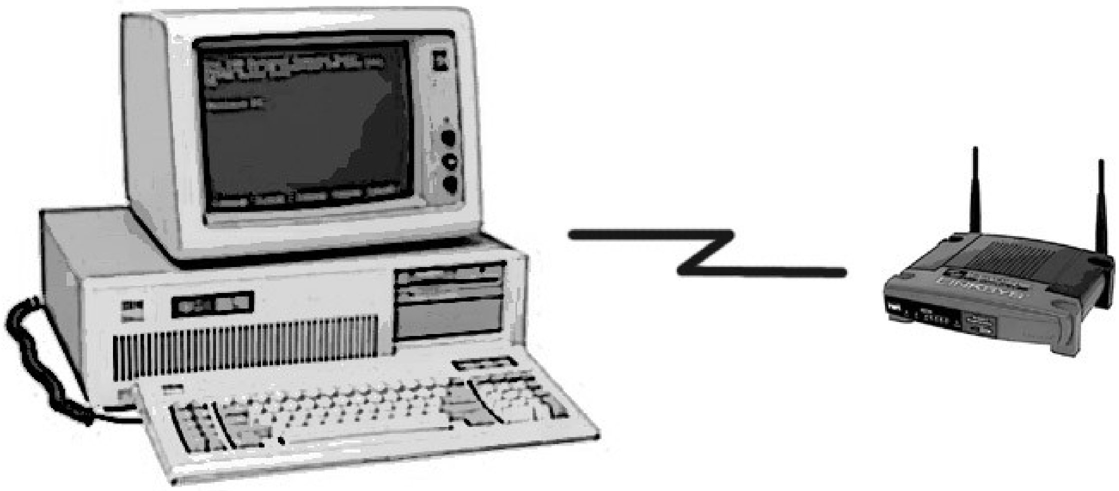

Table of Contents

Overview

Welcome to mTCP!

The primary goal of the project is to bring the joy of TCP/IP networking to older IBM PC compatible machines. These machines include the original IBM PC from 1981 and everything derived from it, including modern virtual machines.

Design decisions have been carefully made to ensure that the right mix of features is included. The target machines are small and constrained in many ways so features like large TCP windows and exotic options are generally out of scope. In general the 80/20 rule applies – most people can live happily with 80% of the features while the other 20% of the features are probably not needed.

Performance is important; an application that is slow enough to make a user gripe will not be used more than once. The 80/20 rule applies here as well - 20% of the code probably is responsible for 80% of the performance, so make that 20% run well and keep the rest clean and easy to maintain.

And finally, the code has to be stable. Nobody wants to deal with data corruption or a system that crashes or malfunctions. DOS is a challenging environment to work in because there are so many ways to trash the machine. Code defensively and test extensively.

What is mTCP?

mTCP is a TCP/IP stack and a set of applications that use the stack. Some utility functions to do common things like parse configuration files and write debug output to a trace file are also provided. Together they form a simple framework for writing TCP/IP applications for DOS.

The TCP/IP code and utility functions are designed to be statically linked into your application. There is no TSR (Terminate and Stay Resident) version of mTCP. This allows each application to customize the stack and include only the features that it needs while improving performance and stability compared to a TSR implementation. However, it also means that TCP/IP functions are not available unless a program is running. (i.e.: The machine will not respond to ping requests while waiting at the DOS command prompt.)

Here are some of the features of mTCP:

- Basic ARP, IPv4, UDP and TCP implementation suitable for small PCs
- IP fragments support
- DNS resolving (via UDP only)
- TCP:
  - support for multiple concurrent open sockets
  - listening sockets for server applications
  - automatic detection and retransmission of packets using measured round trip times
  - TCP zero window support
  - improved flow control handling for dealing with lost packets
- Configuration flexibility: features, buffer sizes, limits, etc. can be customized for each application using #defines
- Comprehensive debug tracing support

- High performance: even the slowest PCs can send and receive TCP/IP sockets data at 70KB/sec with a reasonable Ethernet card.

And here are some of the limitations of mTCP:

- mTCP targets 16 bit DOS only. There is no 32 bit version (yet).
- mTCP programs are always stand-alone DOS executables. There is no TSR version of the TCP/IP stack.
- IP: There is no support for IP options. IP options are needed to implement MTU probing, Traceroute, and a few other handy features so support for IP options may be added in the future.
- TCP: There is almost no support for TCP options. You can pretty far without them so this is not much of a problem.

History

Sometime in 2003 or 2004 somebody figured out that a Xircom PE3 Ethernet adapter would work on a standard IBM PCjr parallel port. For those of you who don't know the IBM PCjr, this is fairly important - the PCjr has no ISA slots so it can't use a traditional Ethernet card. This discovery led me to experiment with the Trumpet TCP/IP stack for DOS. Most of the software that I found was buggy or out of date so while DOS networking was intriguing, it was also disappointing. NCSA Telnet was an exception; it allowed me to use DOS to connect to current Unix machines.

In 2005 I decided to write my own TCP/IP stack for DOS. It started with some attempts to send and receive packets using a packet driver. Sending was generally working but every packet that I received caused my development machine to crash. Out of desperation I decided to test using the Ethernet card in the machine instead of the Xircom PE3 connected to the parallel port and I was able to receive packets without crashing the machine! After three days of struggling to figure out the problem I sought help from Russ Nelson, the originator of the packet driver specification. Russ kindly pointed out that the Xircom packet driver was deficient with regards to the state of the processor flags. With the work-around he gave me I was able to reliably send and receive packets on both adapters and I relearned an important lesson - old code would violate the specs in many possible ways, and without source code it was very difficult to debug.

By necessity the first protocol that I implemented was ARP. IP and UDP came next. By 2007 I was working on TCP using what would become the netcat program to exercise the code. By summer 2008 the TCP code was complete and stable enough to create my first application, IRCjr. The name mTCP came later when I needed a name to describe the collection of applications that I had built.

Since that time I've been adding applications and making continual enhancements to the base TCP/IP stack. Today mTCP refers to several different things: the set of applications, the TCP/IP code, and the software framework that ties them together.

Development environment

Compiler

The short story: use Open Watcom 1.9 on a Windows or Linux machine and cross-compile for 16 bit DOS.

My original development environment was an 80386-40Mhz clone running DOS 5 and Borland Turbo C++ 3.0 for DOS. Turbo C++ provided an IDE, generated 16 bit code, allowed me to write inline assembler, had good documentation, and it previously had a wide base of support. The C++ features were reasonable in 1994 but by 2005 it was quite a bit out of date and incomplete. As I had more and more code to compile the machine took longer to build the project. Eventually I got irritated by the code generation, which was easy to debug but often inefficient.

Sometime around 2010 I decided to move to Open Watcom 1.8 to get a more up-to-date compiler with better code generation. Open Watcom was a good upgrade, but it was painful to use on the old development machine. At this point I moved to a Windows XP machine and used Open Watcom to cross-compile the mTCP applications for 16 bit DOS. Combining Open Watcom with Cygwin on Windows gave me a much better development environment; I had plenty of screen real estate for multiple source code windows and documentation and the machine was infinitely faster at building the project. I upgraded to Open Watcom 1.9 when it became available and I am still using this development environment today.

It would be possible to fix the source code to work with Borland Turbo C++ again but I have found that Open Watcom is generally a superior compiler. Moving to any other compiler would involve changing makefiles, rewriting the assembler portions, and possible working around compiler bugs and run-time differences.

Supported Memory Models

mTCP targets 16 bit DOS which means that you have to deal with the awful memory segments of the x86 architecture.

In general, all of the code supports whatever memory model you choose. The following memory models are used and tested:

- Small: single segment for code and data
- Compact: single code segment with multiple data segments
- Large: multiple code and data segments

The Medium model (multiple code segments with a single data segment) probably works but no applications use it yet.

When using a memory model choose the smallest that your program can work with to take advantage of near pointers, which can improve performance.

Keep in mind that when you do pointer arithmetic that you can wrap around the segment. You will see several sections of code where mTCP normalizes far pointers (adjusts the segment to make the offset as close to 0 as possible) to avoid any segment wrapping problems when doing arithmetic later.

For more information see the section on “Memory Models” in the “Miscellaneous Notes for Programming Under DOS” section.

Directory structure

These are the directories where code is located.

- TCPINC   Includes for the TCP/IP library
- TCPLIB   Code for the TCP/IP library
- INCLUDE   Includes that might be useful across multiple applications
- UTILS   Utilities used during the build process. (Patch.cpp)
- APPS   Top level directory for applications

I generally use upper case letters and adhere to classic DOS 8.3 filenames for the code in case we need to compile under a more limited environment, like the original compilers that were used for the project.

Each application lives in a separate directory under APPS. Each application has a makefile, a configuration file, at least one C file and possibly some header files.

The configuration file is technically a header file that is used to set the configuration options for the mTCP library using a system of #defines. A default set of configuration options is included from TCPINC/GLOBAL.H. Those options include things like buffer sizes and counts, ARP cache configuration, TCP options to include, etc. The local configuration file can override settings in GLOBAL.H as needed.

Notes on the included makefiles

There are generally three targets in a MAKEFILE:

- "clean" erases the executables and object files
- "all" does a clean and builds the application
- "patch" calls the patch utility to fix the Watcom runtime for two irritants. (See the next section for a detailed explanation.) This step is optional.

Within each MAKEFILE there is a set of compiler options that are specific to each program. There is also a set of comments that describe the options kind of like a "quick" reference; the actual Open Watcom docs should be consulted for detailed explanations.

Options generally fall into three classes:

- Error checking
- Code generation for performance
- Calling conventions and memory model configuration

Programs that are performance sensitive will tend to use compiler options that favor fast code generation at the expense of program size. Where performance is not terribly important the compiler be directed to produce more compact code.

Stack bounds checking is normally turned off. This is for performance and code size reasons. You can turn it on during development, but be sensitive to the number of objects you are putting on the stack, your call depth, and the stack size.

(See “Stack space usage” for a discussion on stack space usage.)

The "Patch" utility

At the moment the UTILS directory has one program in it called PATCH. PATCH modifies the program binaries after they have been built, fixing two small problems I perceive in the Open Watcom runtime:

- The runtime looks for a copyright date in the BIOS of your machine and if it does not find one in the correct format it assumes you are on a NEC PC98 compatible machine. This is disastrous when the runtime makes BIOS calls for keyboard handling and generating sound. Affected machines include the old Epson Equity series and early Compaq machines.
- The runtime expands the near heap to maximum size in large data model programs when it first starts up. This makes the programs look like they are consuming 64KB more memory than they actually are.

I "fixed" the problems by patching the code in the runtime. The correct way to fix these problems would be to fix and rebuild Open Watcom but I have not tackled that yet. My fix involves modifying the routines so that they return immediately instead of running.

The MAKEFILE for PATCH will build it so that it runs under your current host operating system; it could be compiled to run under 16 bit DOS but you really don't need to be in 16 bit DOS to patch the programs. It is going to read the map file generated by the linker, look for the routines to patch, and patch them if they are present.

You don't need to patch the executables - that is an optional step. But if you don't you may have problems with older machines that do not have a BIOS copyright date in them at the right spot and your memory consumption will look artificially high.

The original name for the patch utility was called "patch.exe", which worked fine under Windows XP. Windows Vista and later think that any program with the word "setup" or "patch" is some sort of installer, and they will try to pop up a warning asking you if you want to grant the program permission to continue. This behavior is stupid, and the workaround is to rename it to "ptach.exe" instead.

Technical notes and design choices

Application architecture

mTCP applications are stand-alone DOS programs. The TCP/IP stack is compiled into the program directly.

The DOS programming model generally gives you control over the entire machine while your program is active. There are no software threads or advanced operating system functions such as wake-up timers, signals, etc. It is assumed that your program is the only program running, save for interrupt handlers and TSR utilities.

The life cycle of an mTCP application is:

- Startup and read command line arguments, configuration files, environment variables, etc.
- Initialize the TCP/IP stack
- Enter an event processing loop and execute it until the program is ready to terminate
- Shut down the TCP/IP stack
- Exit

The event processing loop will vary from program to program but it generally does the following:

- Check to see if new data from the TCP/IP stack is available.
- Drive pending ARP requests and outgoing TCP/IP packets.
- Process keyboard, update the screen, and perform program logic.

The approach is best described as “busy polling”; there is no operating support for things like events that would wake the program up to do these tasks so we use polling as a substitute. Whenever your program has to wait for something, it should be entering into a loop that does these three things.

Here is an example from Netcat:

```
// Listen is non-blocking.  Need to wait
while ( 1 ) {

  if ( CtrlBreakDetected ) {
    rc = -1;
    break;
  }

  PACKET_PROCESS_SINGLE;
  Arp::driveArp( );
  Tcp::drivePackets( );

  mySocket = TcpSocketMgr::accept( );
  if ( mySocket != NULL ) {
    listeningSocket->close( );
    TcpSocketMgr::freeSocket( listeningSocket );
    rc = 0;
    break;
  }
```

```
    if ( bioskey(1) != 0 ) {
      char c = bioskey(0);
      if ( (c == 27) || (c == 3) ) {
        rc = -1;
        break;
      }
    }
  }
```

Here the code is waiting for a new socket connect to be made. The busy loop checks the keyboard for Ctrl-Break, Ctrl-C or ESC being pressed and exits the loop if that happens. Otherwise, it processes new incoming packets, continues resolving any pending ARP requests, and continues to push out any TCP/IP packets that need to be sent. Processing new incoming data might result in a socket connection so if we see that a new socket connection is available we also exit the loop.

Packet Driver basics

There are hundreds (and probably thousands) of Ethernet cards available for personal computers. It would be impossible for TCP/IP programs to know how to interface with all of them. To make the task feasible Ethernet cards provide device drivers that hide the specifics of the Ethernet card and provide a generic programming interface for higher level programs to use. Several different programming interfaces are available, including NDIS, ODI and Packet Driver. mTCP uses the Packet Driver interface, so any Ethernet card with a packet driver should be usable with mTCP.

A packet driver is a DOS terminate-and-stay-resident (TSR) utility that provides the generic packet driver interface to higher level software but knows the details of how to send and receive data using a specific Ethernet card. Each Ethernet card or family of Ethernet cards requires a specific packet driver TSR to make it usable with mTCP or other programs written to use the packet driver interface. The packet driver TSR provides two primary services - it sends packets on your behalf and it tries to give you packets that it receives from the Ethernet card. The interface to the packet driver is a combination of software interrupts and callbacks. The code that interfaces to the packet driver has to be fast and rock solid.

Packet driver interfacing

PACKET.CPP is the code responsible for interfacing to the packet driver and for managing the buffers used by the packet driver. The buffer management code is the important part; when a new packet arrives a buffer needs to be provided to receive the contents of the packet. If a buffer is not available the packet is lost.

When an mTCP application starts it will try to allocate memory for the buffers that will be used to receive incoming data from the packet driver. Assuming that each incoming packet buffer is 1514 bytes, you can allocate 43 buffers with a single call to malloc and have all of them fit in the same segment. (1514 is the size of a standard Ethernet frame which includes the Ethernet header but not the trailing CRC.) If you need more space you will have to change the code to call malloc several times or use a different malloc variant that can handle larger requests. Larger numbers of buffers can help you during bursts of traffic, especially on slow machines, but if the machine is too slow it just simply may not be able to keep up with all of the incoming packets.

After allocating memory the application needs to find the packet driver and register with it. Registration is a handshaking process that tells the packet driver what packets the program is interested in and how to deliver the packets to the program. mTCP applications assume that they are not sharing the machine or the Ethernet adapter with anything else so they register to receive all Ethernet packet types from the packet driver. mTCP handles at least two different Ethernet packet types - ARP and IPv4. It would be possible to restructure this to make registration calls for each specific Ethernet packet type to be processed, at the cost of added complexity.

When a packet arrives on the wire the Ethernet card will buffer it and eventually signal an interrupt to the host machine to let it know that there is work to do. The packet driver will receive this interrupt and handle the data. If the packet driver decides to give the packet to the application it will make two calls to the application. The first call is to get the address of a buffer to use for the incoming data. The second call is to indicate that the buffer is filled and that it may be processed by the application.

Both calls happen “under the interrupt” so the application code needs to be very quick. Some programs might choose to do work during these calls, such as examine the incoming data and perform some action. mTCP programs only do buffer management; on the first call it provides a buffer from the free list and on the second call it moves the buffer to a ring buffer where the application will see it. This approach reduces the risk of dropped packets by minimizing the time spent in the interrupt handler. More complex code might not be safe either; you don’t know where your program was interrupted, so calling DOS or language run-time routines might not be safe.

Buffers that have been filled are placed in a ring buffer so that they are processed in the order in which they arrive. The tail of the ring buffer is updated when the packet driver makes the second call to let the code know the buffer has been filled. Only the receive code called by the packet driver should update the tail of the ring buffer.

Buffers that are in use are managed by mTCP and application code. It is the responsibility of that code to return all buffers after they are used. Buffers should be quickly processed and returned to avoid the case where there are no free buffers to give the packet driver. Copy what you need from the buffer and return it; once you give a buffer back you have no idea how quickly it will be reused, and a dangling pointer into a buffer is a coding error.

The buffer management code is careful to disable interrupts when manipulating the free list. This is necessary because you don't want to have the packet driver try to take a buffer while you are maintaining the free list. That would cause a race condition that would corrupt the free list. This locking is effectively the only locking we need to do in the entire protocol stack.

Free buffers are put on the free list which is implemented as a stack. The most recently freed buffer will be the first to be given to the packet driver. This behavior improves performance on machines with caches.

Incoming packet processing

When an incoming packet has been received it is placed on a ring buffer where it waits to be processed. When mTCP polls to see if new packets are available it is checking to see if the ring buffer is not empty.

If a packet is on the ring buffer it needs to be processed. The first step is to determine the packet type and route it to the appropriate handler. mTCP generally only handles ARP and IP packets; other packet types are ignored and the incoming buffers are returned to the free pool. ARP and IP each have handlers that will do further processing, such as respond to ARP requests or route the IP packet to the TCP or UDP parts of the code.

The PACKET_PROCESS_SINGLE macro performs the check to see if a new packet is available on the ring buffer and calls the processing function if needed. Processing an incoming packet may result in new data being available for your code; an ARP request could have been satisfied, a UDP packet might have been received, or a TCP socket might have new data available. Of course nothing might happen; we might just respond to an ARP request or answer an ICMP Ping packet.

An alternative approach to the ring buffer could be to drive protocol processing from the interrupt where the packet is received. This would require more locking, be harder to debug, and significantly restrict what the mTCP protocol code could do while under the interrupt. The polling method used here might cause unnecessary spinning while waiting for packets, but it is simpler and more robust.

Incoming UDP packet handling

UDP is a simple protocol that puts a lot of the burden on the end application.

A user application is required to register to receive UDP traffic on specific UDP ports. The registration includes providing a callback function that will be used when a UDP packet on the specified port is received.

When a UDP packet is received it is put on the ring buffer. When the code calls PACKET_PROCESS_SINGLE the protocol code will first do basic IP protocol checking and will route it to the UDP protocol code. That code will then figure out if a user application is interested in that UDP packet and call the user supplied callback function if necessary. The callback function should quickly copy what it needs from the packet and return the packet to the free list. The user application can then later check to see if it has new data to work with and act accordingly.

Incoming TCP packet handling

TCP is a much more complex protocol. To receive data on a TCP socket the socket must first be created and established by the user application.

When a TCP packet is received it is put on the ring buffer. When the code calls PACKET_PROCESS_SINGLE the protocol code will first do basic IP protocol checking and will route it to the TCP protocol code. The TCP protocol code will then look for a socket connection that matches the packet, or for a listening socket connection that can match the packet. If no matches are found the packet is rejected and freed.

If a match is found the packet is processed in the context of the socket that owns it. The state of the socket determines how the packet is handled. Eventually the data (if any) is copied from the packet, the state of the socket is updated, and the packet is returned.

User code needs to poll its open sockets to look for state changes and new data. The recv method on a socket can be used to check for new data. Other socket methods allow a user to detect when a socket has been closed. Code that listens for new sockets needs to periodically check for new sockets with the "accept" call.

Outgoing UDP packet handling

UDP is a "best effort" delivery protocol. Each packet is treated as an individual entity without any state being shared between packets. There is no loss detection or retry capability unless the user application codes it. This makes UDP light and fast.

Sending a UDP packet is easy. The packet gets correctly formatted at both the UDP and IP layer, and the packet driver is called to send the packet. After that, it is gone. If it does not make it the user application needs to timeout and try again.

Outgoing TCP packet handling

Most end user applications will call "send" on an active socket to send data. This is all they need to know about.

Under the covers things are much more complex. TCP pre-allocates a pool of buffers that are used exclusively for sending TCP data. The pool of buffers contains meta data for managing the sending of the data, room for the protocol headers, and room for the end user data. The TCP protocol handler owns the buffers; end user applications generally should not be aware of or manipulating these buffers.

New data to send is first enqueued on a ring buffer called "outgoing." These are packets that need to be sent down the wire at the next opportunity. The actual sending of these buffers is triggered by a function called "Tcp::drivePackets." Requiring an explicit call to send the packets allows for multiple packets to be batched up for sending, which is more efficient.

After packets are sent down the wire they need to be held in a ring buffer called "sent" until their arrival is acknowledged by the other side of the socket connection. In the event that a packet is lost a timeout timer will cause packets to be resent. When a packet is finally acknowledged from the other side the buffer can be recycled.

IP Checksumming

IP and TCP checksums are required at all times. UDP checksums are optional but recommended. mTCP always computes and verifies all checksums including UDP checksums. Although the Ethernet CRC protects the packet from corruption, IP is routable across different subnets so corruption of the packets is possible when going through a gateway, hence the need to check checksums. Bad hardware such as flaky memory or a bad Ethernet device can also be detected with the IP, UDP and TCP checksums.

The IP, TCP and UDP checksums are handled as part of the protocol handling so there is nothing that a user needs to be aware of. A packet with a checksum error will not be processed because it is invalid. It will generate a warning log message if tracing is turned on.

This code is notable because it is the only code written entirely in assembler. The checksum routines are extremely important for performance and the nature of the code makes it hard for any C compiler to effectively optimize it. The main loop of the checksum routine does map nicely to the hardware even though the hardware is 16 bit - the 'carry' bit makes all of the difference.

Loop unrolling is used to reduce the branching overhead within the checksum routines. In general you can count on the IP headers to be at least 20 bytes long. UDP headers are eight bytes, and TCP headers are a minimum of twenty bytes. All headers grow in increments of four bytes. The code takes these into account so that the number of bytes in a header is a multiple of a number of bytes in the loop iteration.

IP Fragments

A single IP packet can be as large as 64KB in length. While it is technically possible to use such large packet sizes, in practice is is quite rare because it causes additional overhead and complexity. In the early days of the Internet fragments were more common because not everything was Ethernet, and IP gateways had to support a minimum MTU of 576 bytes. But now that everything basically looks like Ethernet the need to create fragments has mostly gone away. (Dial-up users and people not using Ethernet will still encounter fragments.)

mTCP has some minimal fragment support:

- The maximum IP packet size is 2KB. This setting is configurable and has been tested up to 8KB.
- A maximum of four fragmented IP packets can be reassembled at a time.
- An IP packet can have a maximum of ten fragments.
- The TCP library will never send more data than fits in the MTU size of the machine, so fragments are not generated by mTCP.
- The UDP library will create and send fragments as needed.

In short, the mTCP support for fragments is enough to get by if something accidentally gets fragmented. If you are going to be using a strange network topology that results in more fragmentation you will want to recompile mTCP to support more in-flight fragment reassemblies. If you are using UDP and need IP frames greater than 2KB in size you will also need to recompile.

Timer management

My early code used the C runtime to get the time of day at either 1 second of granularity using time() or at millisecond granularity using gettime( ). The granularity using gettime was effectively 55 ms, due to the way that BIOS keeps track of time.

Repeated calls to gettime were too expensive, and comparing the timestamps was also very expensive. This lead to a lot of wasted time trying to determine if enough elapsed time had past when trying to detect a timeout.

A normal PC BIOS will setup a timer that ticks every 55 milliseconds. The timer updates a counter in the BIOS area of RAM so one can easily count timer ticks by looking at that counter. The problem with the counter is that it can rollover on you. To prevent having to worry about the rollover case mTCP hooks the timer interrupt and keeps its own tick count. That allows us to easily determine elapsed time (by looking at the tick counter) without worrying about the rollover case. (In theory you have to worry about it if your program runs for more than 2700 days …)

Stack initialization and teardown

After your code has loaded and checked its parameters it will need to start the mTCP stack. The Utils::initStack( ) call is used to do this. Here is the sequence of steps that are performed:

- Initialize the data structures used to interface to the packet driver.
- Check for the existence of the packet driver.
- Register with the packet driver to accept all types of packets.
- Hook the BIOS timer interrupt.
- Initialize ARP, IP, and ICMP, and TCP
- Initialize DNS
- Enable packet receiving by turning on the flow of buffers

If any of these steps fail initStack will clean up correctly.

If initStack completes you have two interesting problems to handle:

- The packet driver is calling mTCP for each new packet it receives
- The timer tick interrupt is causing some mTCP code to run

While not really problems, you need to be aware that these things are happening. If your program terminates abnormally without unhooking from the packet driver and the timer interrupt your machine will crash. You need to ensure that you always use Utils::endStack( ) while your program is ending to ensure that these things get unhooked properly.

Most applications will hook Ctrl-Break and Ctrl-C to make sure that the user doesn't break out of the application without having the proper stack shutdown performed. The replacement handler sets a global variable that indicates the user wants to leave the program. This variable should be checked during busy loops to ensure the program remains responsive in the event that the user wants to quit.

Utility functions

Trace files

Communications code is hard to debug with a traditional IDE and breakpoints because things happen asynchronously and there is a lot of state involved. Hitting a breakpoint and then letting a developer examine the state of the machine usually takes too long, causing connections to die.

As a result mTCP uses a log file as its primary method of debugging. Normally the trace support and trace points are compiled into the code but are not active, allowing the program to run at close to full speed. If debugging is needed the trace support can be turned on by setting two environment variables - one to specify the output file and one to specify which tracepoints are active. The ARP, IP, UDP, TCP, and DNS protocols can all be selected for tracing. Application specific messages can also be selected. In addition, a trace point can be marked as a warning message, allowing one to capture warnings even if tracing is not enabled for a specific protocol or the application. Lastly, full packet dumps can also be turned on dynamically.

Tracing is useful but it can slow a program down dramatically, especially when running on a slower system. It is possible to compile mTCP with no tracing support to get a minor performance boost and to generate smaller executables, but doing so greatly reduces the ability to debug problems.

Token parsing

mTCP does a lot of token parsing when reading configuration files and in some of the programs. A utility function called getNextToken is used for this. It has the following properties:

- Leading whitespace is ignored.
- Spaces are delimiters.
- Quoting (“) can be used to group words with spaces together as one token.
- The next token is returned in a user provided buffer
- A pointer to the next location to start parsing at is also returned, unless you run out of input in which case NULL is returned.

Configuration file handling

When an mTCP program starts it will process command line flags and then read configuration data from the mTCP configuration file. The configuration file is located by examining the MTCPCFG environment variable. Things like the TCP/IP address, netmask, gateway, and DNS servers are read from the file.

mTCP applications also store application-specific configuration in the configuration file. A simple interface for opening the file and looking for specific values is provided:

- Utils::openCfgFile: open the MTCPCFG file
- Utils::getAppValue: search the MTCPCFG file for a key and return the value, if available.
- Utils::closeCfgFile: close the MTCPCFG file

More complex applications are free to use other mechanisms for managing their configuration.

The mTCP Sample program

A sample program that works like a bare-bones netcat program can be found in APPS/SAMPLE. The sample is a simplified version of the Netcat program. This program can listen for an incoming socket or make a socket connection to a server. Anything you receive from the socket gets put on the screen and anything you type will be sent to the other end. Alt-H shows onscreen help and Alt-X exits the program.

Program startup and reading the mTCP configuration file

All mTCP programs need to read the mTCP configuration file to find out which interrupt the packet driver is using, what the IP address and netmask are, and some other optional settings. A function called Utils::parseEnv does all of this for you:

```
int main( int argc, char *argv[] ) {

  fprintf( stderr,
           "mTCP Sample program by M Brutman (mbbrutman@gmail.com) (C)opyright 2012-2020\n\n" );

  // Read command line arguments
  parseArgs( argc, argv );

  // Setup mTCP environment
  if ( Utils::parseEnv( ) != 0 ) {
    exit(-1);
  }
```

The program first checks its own command line parameters by calling its own parseArgs routine. After that it calls Utils::parseEnv so that mTCP can find and read its configuration file. There are no parameters to use but there is a return code to check. Any return code other than 0 is a failure.

Starting the TCP/IP Stack

If you made it this far then mTCP has read its configuration parameters and you are ready to tell it to start sending and receiving packets. The next call makes the stack go live:

```
  // Initialize TCP/IP stack
  if ( Utils::initStack( 2, TCP_SOCKET_RING_SIZE, ctrlBreakHandler, ctrlCHandler ) ) {
    fprintf( stderr, "\nFailed to initialize TCP/IP - exiting\n" );
    exit(-1);
  }

  // From this point forward you have to call the shutdown( ) routine to
  // exit because we have the timer interrupt hooked.
```

The first parameter is a number of TCP sockets to create. The second parameter is a number of outgoing TCP buffers to create. Sockets and buffers take up memory space so allocate only what you need. Parameters three and four are your Ctrl-Break and Ctrl-C handlers. If the function returns anything but zero you have an error and the program should terminate. The most common error is not being able to find the packet driver because it was not loaded or it was loaded at a different interrupt number.

If you succeed then mTCP is alive and possibly receiving and processing packets from the packet driver. On a busy network you will be getting traffic almost immediately. Next it will be time to get into your main program loop to start processing those packets.

mTCP hooks the timer interrupt to be able to keep track of time more efficiently than the C runtime can. If your program runs far enough to initialize the TCP/IP stack (past Utils::initStack) then you need to ensure that your program will call the routine to end mTCP gracefully (Utils::endStack).

Unless your program crashes in a horrible manner this should not be a problem. But one major hole that people forget about is Ctrl-Break; somebody pressing this can force your program to end early without properly shutting down mTCP. This is why a Ctrl-Break and Ctrl-C handler are required for initStack. The new Ctrl-Break handler sets a flag that we can check in our main program loop.

Getting a socket connection

The next part of the program involves either waiting for an incoming socket connection or making a socket connection to another system. The choice is made on the command line. When waiting for a socket from another system your program behaves like a server. When contacting another system your program behaves like a client.

Resolving a server name and connecting to a server

If you are making a connection to another system you will need to go through DNS name resolution and do a socket connect:

```
TcpSocket *mySocket;

int8_t rc;
if ( Listening == 0 ) {

  fprintf( stderr, "Resolving server address - press Ctrl-Break to abort\n\n" );

  IpAddr_t serverAddr;

  // Resolve the name and definitely send the request
  int8_t rc2 = Dns::resolve( ServerAddrName, serverAddr, 1 );
  if ( rc2 < 0 ) {
    fprintf( stderr, "Error resolving server\n" );
    shutdown( -1 );
  }
```

Dns::resolve is used to convert server names to IP addresses. If you used a numerical IP address then the first call to Dns::resolve will resolve it and you will not have to wait. Otherwise, you need to enter a loop to wait for DNS to complete.

```
  uint8_t done = 0;

  while ( !done ) {

    if ( CtrlBreakDetected ) break;
    if ( !Dns::isQueryPending( ) ) break;

    PACKET_PROCESS_SINGLE;
    Arp::driveArp( );
    Tcp::drivePackets( );
    Dns::drivePendingQuery( );

  }
```

The loop has to process incoming packets, retry Arp requests if necessary, and retry DNS requests if necessary. Those functions handled by PACKET_PROCESS_SINGLE, Arp::driveArp, and Dns::drivePendingQuery.

(This DNS implementation is UDP based. Dns::drivePendingQuery is only needed because UDP packets can get lost, and we need a way to detect if we need to resend our DNS request.)

```
     // Query is no longer pending or we bailed out of the loop.

    rc2 = Dns::resolve( ServerAddrName, serverAddr, 0 );

    if ( rc2 != 0 ) {
      fprintf( stderr, "Error resolving server\n" );
      shutdown( -1 );
    }
```

When DNS completes or times out the loop will end. At the end of the loop another call to Dns::resolve tells you the final result.

```
    mySocket = TcpSocketMgr::getSocket( );

    mySocket->setRecvBuffer( RECV_BUFFER_SIZE );

    fprintf( stderr, "Server resolved to %d.%d.%d.%d - connecting\n\n",
             serverAddr[0], serverAddr[1], serverAddr[2], serverAddr[3] );

    // Non-blocking connect.  Wait 10 seconds before giving up.

    rc = mySocket->connect( LclPort, serverAddr, ServerPort, 10000 );

  }
```

If DNS resolved the name and gave you an IP address you will need to allocate a socket, set the TCP receive buffer size for the socket, and make a TCP socket connect call. mTCP owns all of the socket data structures - you just get a pointer to them. The call to TcpSocketMgr?::getSocket gets you a socket to use. When you are done with a socket you should return it using the TcpSocketMgr?::freeSocket call.

Listening for an incoming socket

If you decided to listen for an incoming socket instead things are a little different:

```
  else {

    fprintf( stderr, "Waiting for a connection on port %u. Press [ESC] to abort.\n\n",
LclPort );

    TcpSocket *listeningSocket = TcpSocketMgr::getSocket( );
    listeningSocket->listen( LclPort, RECV_BUFFER_SIZE );
```

Here we allocate a socket using TcpSocketMgr?::getSocket but instead of using this socket to make an outgoing connection to a server we are going to use it to listen for incoming sockets. The listen call tells mTCP what port to listen on and what receive buffer size to use for newly created sockets.

```
    // Listen is non-blocking.  Need to wait
    while ( 1 ) {

      if ( CtrlBreakDetected ) {
        rc = -1;
        break;
      }

      PACKET_PROCESS_SINGLE;
      Arp::driveArp( );
      Tcp::drivePackets( );

      mySocket = TcpSocketMgr::accept( );
      if ( mySocket != NULL ) {
        listeningSocket->close( );
```

```
          TcpSocketMgr::freeSocket( listeningSocket );
          rc = 0;
          break;
        }

        if ( _bios_keybrd(1) != 0 ) {
          char c = _bios_keybrd(0);
          if ( (c == 27) || (c == 3) ) {
            rc = -1;
            break;
          }
        }
      }
    }
```

After that, we wait! The loop checks for keyboard input to see if the user wants to end the program early. It also processes incoming packets. (If a user hits Ctrl-Break the CtrlBreakDetected? variable will be set by the new interrupt handler we installed after TCP/IP went live. Instead of ending the program prematurely, we can do an orderly shutdown.)

When an incoming socket connection is received the TcpSocketMgr?::accept call will return a pointer to the socket. At that point we stop listening for sockets by closing the listening socket and we fall into the common code for sending and receiving data on the socket.

Main processing loop

The code is too large to reproduce here verbatim, but here is the general idea:

```
while ( programIsNotEnding ) {
  Process Incoming Packets
  Perform some processing on those packets
  Check for User input
}
```

Lets get into more detail ...

Process Incoming Packets

mTCP needs to be told when it can run to process packets. Unlike a modern operating system, it does not happen in the background automatically.

The way to tell mTCP to process packets is to make a series of calls:

```
// Service the connection
PACKET_PROCESS_SINGLE;
Arp::driveArp( );
Tcp::drivePackets( );
```

PACKET_PROCESS_SINGLE is a macro that checks for new packets from the packet driver and if any are found it processes those packets.

Arp::driveArp is used to check up on pending Arp requests and retry them if necessary.

Tcp::drivePackets is used to send any packets that you might have queued up to send.

(Earlier you saw another call: Dns::drivePendingQuery. That functions like these calls, but it is only needed when performing a DNS lookup.)

You could have one function or macro that hides these three functions under the covers. I have them broken up into three different calls so that you can more carefully control when you check for incoming packets vs. when you decide to push packets out. But in general you will find the code making all three calls at the same time.

Perform some processing on those packets

This program is a simplified version of the Netcat program. So its major job is to receive and display things from the socket and to send user keystrokes out on the socket.

This is the code that checks the socket for new data:

```
if ( mySocket->isRemoteClosed( ) ) {
  done = 1;
}

// Process incoming packets first.

int16_t recvRc = mySocket->recv( recvBuffer, RECV_BUFFER_SIZE );

if ( recvRc > 0 ) {
  write( 1, recvBuffer, recvRc );
}
else if ( recvRc < 0 ) {
  fprintf( stderr, "\nError reading from socket\n" );
  done = 1;
}
```

If the socket is not closed then the code will try to receive data on it. A return code of 0 indicates no data, which is not an error. A negative return code indicates a socket error and a positive return code is the number of bytes that were received. Any bytes that are received are written to to STDOUT right away.

Here is the code that checks for user input:

```
if ( CtrlBreakDetected ) {
  fprintf( stderr, "\nCtrl-Break detected\n" );
  done = 1;
}

if ( _bios_keybrd(1) ) {

  uint16_t key = _bios_keybrd(0);
  char ch = key & 0xff;

  if ( ch == 0 ) {

    uint8_t ekey = key >> 8;

    if ( ekey == 45 ) { // Alt-X
      done = 1;
    }
    else if ( ekey == 35 ) { // Alt-H
      fprintf( stderr, "\nSample: Press Alt-X to exit\n\n" );
    }

  }
  else {
    int8_t sendRc = mySocket->send( (uint8_t *)&ch, 1 );
    // Should check the return code, but we'll leave that
    // as an exercise to the interested student.
  }
}
```

Note the first check is to check that the user has not hit Ctrl-Break. If they did our interrupt handler would have set the flag, which tells us to end the program.

If the user hits a key we read the keyboard and process the key as appropriate. We recognize special keys Alt-H (Help) and Alt-X (Exit). Any other keystroke gets sent on the socket using the send call.

The actual packet does not get sent on the wire until you make the Tcp::drivePackets( ) call. That code also handles resending the packet if it gets lost along the way.

Ending your program gracefully

Once the TCP/IP stack is initialized and receiving packets you need to ensure that it gets shut down properly before your program ends. If you don't your system will eventually crash because of the dangling timer interrupt.

Calling Utils::endStack will do the trick. In my code I usually have a shutdown routine that I can call from anywhere that will make the Utils::endStack call and return to DOS:

```
static void shutdown( int rc ) {
  Utils::endStack( );
  Utils::dumpStats( stderr );
  fclose( TrcStream );
  exit( rc );
}
```

Utils::endStack will take care of the timer interrupt that it had hooked. Since my code installed the interrupt handler for Ctrl-Break it is responsible for removing it. Both the Ctrl-Break and Ctrl-C handlers were installed but only the Ctrl-Break handler needs to be restored. DOS will take care of restoring the Ctrl-C handler automatically.

Building the sample

To build the sample program you should have Open Watcom installed. You should be able to compile a test program with it.

The easiest way to build the sample is to:

- unpack the mTCP source code
- create a new subdirectory under the "APPS" directory with the sample code in it
- change directory to the new directory
- run "wmake"

The included Makefile is already setup to find all of the mTCP include files. Wmake will use the Makefile and create an executable for you. The executable can then be run on your favorite real or emulated DOS machine.

Summary

The sample program showed you the following techniques:

- Reading the mTCP configuration file
- TCP/IP initialization
- Listening for incoming sockets
- Resolving a host using DNS

- Opening a socket connection to another machine
- Reading and writing on the socket
- Proper shutdown

The real Netcat program isn't that much more complicated - it just handles all of the special processing needed for Telnet newline processing, screen echoing, and it does some tricks for better performance when sending and receiving data from files. You will find this makes a useful skeleton program for your own network applications.

Miscellaneous notes for programming under DOS

Many of us are familiar with 32 or 64 bit flat address space environments. Well, things are a little bit different in this environment. While C code is generally portable there are a few things you need to be aware of.

The DOS 16 bit programming environment

Welcome to old school programming ...

Here are the downsides:

- The operating system is more of a program loader with some low level library functions.
- Memory is segmented. Pointers can be two bytes or four bytes, and you can have both types in a program.
- There is no virtual memory or memory protection.
- There are no threads.
- Space is very limited.
- You basically have complete control of the machine
- The older machines can be quite slow (4.77Mhz)

It basically looks like embedded programming does today, but on a strange architecture without the benefit of modern compilers and tools. But as a reward you get total control of the machine.

Memory models

This is a 16 bit environment! Registers are 8 or 16 bits in size. It is obvious that we can't fit everything we need (ROM, program code, program data, stack, and heap) in a 16 bit address space. Intel x86 architecture lets us get around the 16 bit nature of our pointers by using segments and combining pointers together.

The compiler generally does a good job of hiding the fact that code lives in segments; it will generate the right series of instructions and pointer fixups to do whatever you need. The compiler is also fairly good at managing your data pointers, giving you the illusion of a larger address space. But if you use large data structures or do a lot of pointer manipulation and arithmetic you need to be aware of the segmented memory architecture. (Pointers wrap at 64K boundaries unless you take precautions!)

My general method of getting around pointer problems is to limit each instance of a data structure to 64KB or less. That allows me to do whatever pointer math I need to without fear of running over a segment. If I need to do larger manipulations I will normalize the pointers instead of turning on "huge" pointers in the compiler.

Here are the standard memory models we can choose from:

- small: a single code segment and a single data segment; near pointers for code and data.
- compact: a single code segment and multiple data segments; near pointers for code, far pointers for data
- medium: multiple code segments and a single data segment; far pointers for code, near pointers for data

- large: multiple code and multiple data segments; far pointers for code and data
- huge: the same as large but the pointers get normalized during runtime operations. (Very slow!)

(Note: Normalizing a pointer means adjusting the segment and offset such that the offset is as close to zero as possible. This means the offset is between 0 and 0xF.)

The applications are generally compiled using the "compact" or "large" memory models. I try to use the smaller memory models where possible; near pointers give you better performance. The buffer sizes required by the applications generally require far pointers for data, although some of the smaller applications get away with the small memory model where both data and code use near pointers.

See the Open Watcom documentation and any decent Intel assembly language book for a discussion of memory models, near pointers and far pointers.

Data types

Although registers are 16 bits in size the compiler can handle data types up to 32 bits in size with no problems.

I generally use typedefs so that it is very easy to see how wide a field is just from looking at the declaration. For example, I use "uint16_t" instead of "unsigned int". This helps when you are constantly transitioning between different architectures.

x86 is a Little-endian architecture. "Network byte order" is used by the TCP/IP protocols, and it is Big-endian. You will see usages of ntohs, ntohl, htons, htonl in the code where we need to do conversions. Use these macros correctly and religiously, and keep in mind that if something looks correct in memory if it gets loaded into a register as it might be byte swapped during the load.

If I need to look at a section of memory in two or more different ways I'll generally use a C union to map the memory out instead of doing pointer casting. The compiler likes this approach better and you are less likely to make a coding mistake. (Casting has its uses but you are generally overriding any type checking the compiler is doing.)

Stack space usage

Stack space is a finite resource. It gets allocated at link time and can not be grown at run time. Overflowing your stack means that you are corrupting something.

Be sensitive to the number of objects you are putting on the stack, your call depth, and the overall stack size. My coding style tends to put large objects in static (global) variables to avoid bloating the stack.

Using multiple functions to perform work adds to call overhead but allows you to grow and shrink the call stack as needed. In contrast large functions with lots of variables generally consume a lot of stack space all of the time, and they only release their stack storage when the function exits. (A good example - never put a large stack variable in main( ) because unless you are constantly using it you will never be able to return to recycle the stack space.)

Even though you may have allocated enough stack space in your environment, other versions of DOS running on different machines will behave differently. I ran into this problem on some machines where

DOS did not have its own interrupt stacks; the combination of DOS and packet drivers in use caused stack overflows. All of the stack sizes have been adjusted since then to be overly generous.

Stack bounds checking is normally turned off in the makefiles. This is for performance and code size reasons. You can turn it on during development but after you have correctly sized your stack and are done with your testing it is probably not providing value.

Memory fragmentation and object pooling

We don't have a sophisticated memory allocator to work with so you need to pay attention to your usage of the heap.

The biggest threat to long running programs that use heap is heap fragmentation. Unless you are working with fixed length data structures where all of the allocations are the same size or you impose strict rules on how you allocate memory, you run the risk of heap fragmentation. If the heap gets fragmented your request for a memory allocation will fail, and your program will probably die.

I minimize my run-time use of the heap by allocating most of the memory that I need up front and then managing that memory myself. For example, there is a fixed pool of buffers used for incoming packets. One large memory allocation for incoming buffers gets done at program startup. The list of used and free buffers is maintained by the packet handling code, which uses a simple stack to maintain the free list. Once the initial allocation is made the buffer management code never goes back to the heap.

This is in contrast to going to the heap when you need a buffer and returning the buffer to the heap when you are done with it. Using the heap like that increases the overhead, risks fragmentation, and is not safe if you do it from within an interrupt handler.

Screen handling

Screen handling on older machines can be terribly slow, especially when using the BIOS routines. Programs like IRCjr and Telnet would be unusable on 4.77Mhz machines if only the BIOS routines were used.

To make these programs perform well a virtual screen buffer is used. The virtual screen buffer provides backscroll capabilities and can be updated very quickly because it is just memory. The real screen is updated from the virtual buffer when the user interface decides to resync them. In Telnet this can lead to “jump scrolling” when a lot of new lines are sent. Smoother scrolling would be nicer but would also be much slower.

On original IBM CGA cards this technique can lead to “CGA snow”, a phenomenon that occurs when the CPU alters video memory while the video card is drawing the screen. This is due to the design of the original CGA card; they did not use dual ported memory. Later machines and clone cards generally do not have this problem. If you do have this problem you can set the MTCP_NO_SNOW environment variable which will fix the problem but make things a little slower. (Direct screen writes will still be used, but they will be done only during the vertical retrace interval.)

Tested machines and environments

mTCP is not particularly fussy about what it runs on. As long as you have some variant of DOS and a machine (physical or emulated) that supports Ethernet via a packet driver, mTCP should work.

What follows is a partial list of machines and environments I have personally tested on:

Machines:

- 8086/8088 class: IBM PC, IBM PC XT, IBM PCjr, IBM PS/2 Model 25, IBM PC Convertible, IBM Portable PC
- 80286/80386/80486 class: IBM PC AT, IBM PS/2 L40SX, generic 80386-40, generic 80486s
- Pentium and better class: generic Pentium 133, IBM Aptiva (Pentium II)

CPUs: 8088, 8086, V20, 80286, 80386, 80486, Pentium, Pentium II, etc.

Video cards: MDA (monochrome), CGA, EGA, and VGA. (All using MDA and CGA modes.)

Ethernet cards:

- Xircom PE3 series (parallel port attached Ethernet adapter)
- 3Com Etherlink II (3C503, etc.) (ISA)
- NE1000 and clones (ISA)
- NE2000 and clones (ISA)
- Western Digital/SMC 8003 series (WD8003E, WD8003EP, WD8003WT, etc.) (ISA)
- LinkSys LNE100 (PCI)
- Intel EtherExpress 8/16 (ISA)
- Davicom DM9008F (ISA)
- Danpex EN-2200T (NE2000 clone)

Operating systems:

- DOS 2.1, DOS 3.3, DOS 5.0, DOS 6.22, Win 98 DOS
- FreeDOS

Emulation and virtual machines:

- VMWare and VirtualBox
- DOSBox: DOSBox doesn't officially support Ethernet. Previously I've used the “H-A-L 9000 megabuild” of DOSBox to get NE2000 emulation, but this is very out of date now. I'd love to see NE2000 emulation fully supported in DOSBox but I don't know if that is in the roadmap anytime soon.