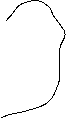
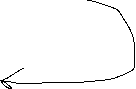
Final Project – T1 Telnet client

A screenshot of a computer

Description automatically generated



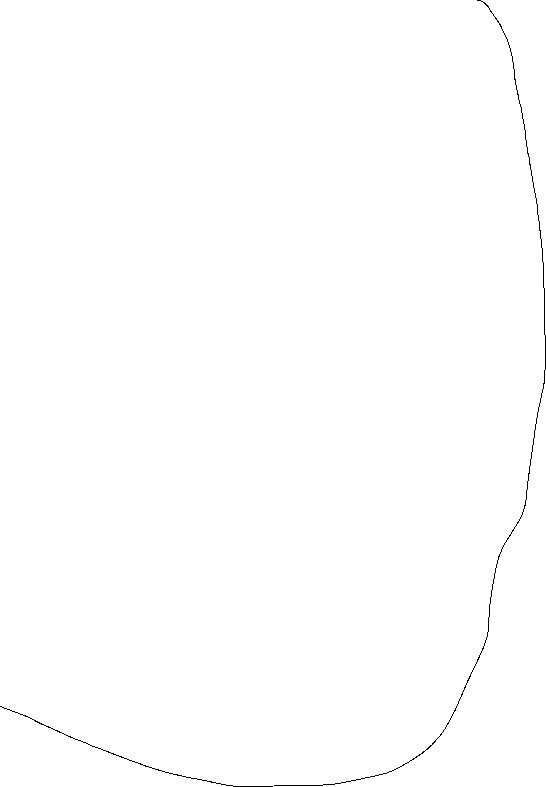
Error



Error

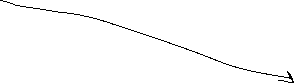
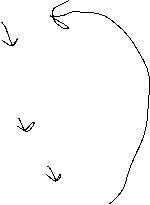


network\_close\_socket:



malformed\_ip\_address\_exit:

network\_premature\_exit:



cWriteStdout:

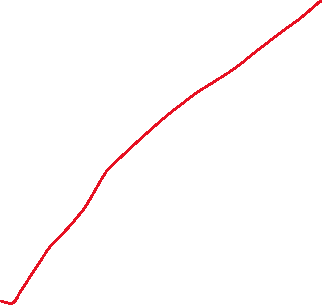
cReadSocket:

cReadStdin:

cWriteSocket:

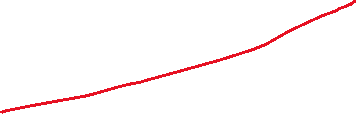
Error

Error

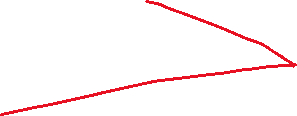


Error

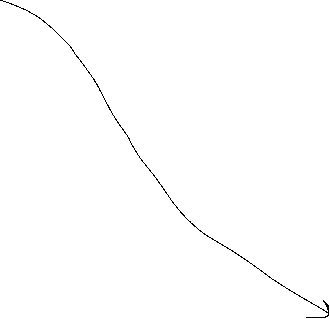
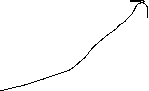
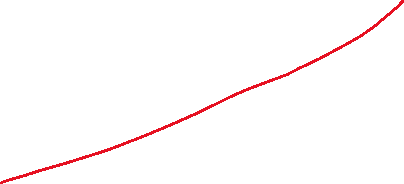
Error



Error



Error



premature\_exit:

digits\_count\_loop:

string\_length\_loop:

cStrtoul

string\_ip\_parse\_loop:

octet\_complete:

null\_byte\_encountered:

cStrIP\_to\_Octets



cWriteString:



cExit

invalid\_program\_arguments:

parse\_program\_arguments:

network\_open\_socket:

network\_connect:

check\_socket\_file\_descriptor\_done:

check\_socket\_file\_descriptor:

check\_read\_file\_descriptors:

check\_stdin\_file\_descriptor:

network\_read\_write\_loop:

network\_setup\_file\_descriptors:

\_start

## \_start:

* This is where the program will start.
* This function will pop eax from stack and will check if user entered 3 arguments.
* If the user didn’t enter 3 arguments it will print an error and exit by calling cWriteString and cExit
* If user entered 3 arguments, it directs the program to parse\_program\_arguments:

parse\_program\_arguments:

* This function will convert the ip address and port numbers from strings to numbers.
* This function pops eax from stack and calls cStrIP\_to\_Octets to convert ip address to number.
* It then checks how many arguments are left in eax, if less than 0 (no port number) it will direct to invalid\_program\_arguments:
* Otherwise, it moves on and calls cStrtou, which will convert the port number from string to number.
* Next, it stores the converted port number (which is in eax) into a variable called ipPort
* This function compares eax with 0 again, if greater than 0 we move to network\_open\_socket function.

## invalid\_program\_arguments:

* + This sub-function will print an error and exit by calling cWriteString and cExit if the user did not enter a port number.

## network\_open\_socket:

* This function starts by creating a socket by using a system call.
* Next, it stores the file descriptor in a variable called sockfd.
* it will then compare eax (socket file descriptor) with 0 to check for errors.
* if fd is greater than 0 we jump to network\_connect
* if less than 0, we print an error and exit by calling cWriteString and cExit.

## network\_connect:

* This function will fill in the network family, port, and IP address information, in order to connect.
* First it moves the serverSockaddr into edi
* Next it stores the network family in ES DI
* It precedes to do the same with the ipPort variable, but it needs to store it in big endian.
* It accomplishes this by moving ipPort into ax, then shifting it 8 bits to the right and doing it again.
* Next, it will store the ipOctets in esi and then zero out the remaining bytes of the structure.
* Now the fuction will perform some instructions that will set up an array that will hold the arguments for connect()
* The instructions move these variables into the array: connectArgs, sockfd, serverSockaddr, serverSockaddrlen
* Next, the function performs a system call for connect()
* It checks for a successful connection and directs to network\_setup\_file\_descriptors if successful.
* If not successful: it prints an error and exit by calling cWriteString and cExit.

## network\_setup\_file\_descriptors:

* This function will add stdin and the socket file descriptor to the array.
* It starts by clearing out the masterReadFdArray
* Then it moves masterReadFdArray into edi and proceeds to move 1 into the array to add stdin
* Next, it finds the offset to the beginning of the file descriptor array in order to set the correct bit for the socket file descriptor in the array.
* The last thing this function does is perform bitwise operations on the sockfd variable in order to set the correct bit.

## network\_read\_write\_loop:

* This function is a loop that will keep looping for as long as the user stays connected to the socket; it contains a select() system call as well as sub- functions that facilitate reading and writing from the socket and standard input.
* The first part of this function is the preparation for a making the select system call; it moves the masterRdArray into the checkReadFdArray, which gets passed into the system call.
* Next, the system call is performed, and if there are any errors, it will print an error and exit using cWriteString and network\_premature\_exit.
* If no errors, it moves on to check\_read\_file\_descriptors.

## check\_read\_file\_descriptors:

## check\_stdin\_file\_descriptor:

* + These two sub-functions are the same, they just get called differently in different places.
  + This function checks to see if the read/write file descriptor is set; to see if there is data to be read from stdin and written to the socket.
  + This sub-function will load the first byte of the checkReadFrArray into al and mask it, then check if it is set.
  + If it is set it reads the data into a buffer and writes to the socket using cReadStdin and cWriteSocket
  + If the first bit is not set, then it jumps to check\_socket\_file\_descriptor.

## check\_socket\_file\_descriptor:

* + This sub-function checks if the socket file descriptor is set; to see if there is data to be read from socket and written to standard out.
  + This function loads checkReadFdArray into esi
  + Then it performs bitwise and shifting operations on sockfd, very much like in network\_setup\_file\_descriptors, in order to check the correct bit.
  + At the end of the operations, it checks the correct bit from sockfd (bit stored in bl) against the corresponding bit in checkReadFdArray (bit stored in al)
  + If the bits are the same (set), it reads from socket and writes to stdout using cReadSocket and cWriteStdout
  + If the bits are not the same (not set), it jumps to check\_socket\_file\_descriptor\_done.

## check\_socket\_file\_descriptor\_done:

* + This sub-function only has one instruction; it contains a jump back to the beginning of the loop (network\_read\_write\_loop)

## network\_premature\_exit:

## network\_close\_socket:

* These two functions are the same; they just get called differently in different locations.
* This function contains a system call to close the socket and then proceeds to call cExit.

## cExit:

* This function simply contains a system call to exit the program.

## cReadStdin:

* This function performs a read system call from stdin.
* It stores the data in a variable called readBuffer and stores the length in readBufferLen.

## cReadSocket:

* This function performs a read system call from the sockfd file descriptor.
* It stores the data in a variable called readBuffer and stores the length in readBufferLen.

## cWriteStdout:

* This function performs a write system call to stdout.
* It writes the data from readBuffer and uses the length parameter from readBufferLen.

## cWriteSocket:

* This function performs a write system call to the sockfd file descriptor.
* It writes the data from readBuffer and uses the length parameter from readBufferLen.

## cWriteString:

* This function performs a write system call to stdout but creates its own stack first.
* It writes the data from its stack (ebp + 12) and uses the length parameter from its stack (ebp + 8).
* Then it smashes its stack

## cStrIP\_to\_Octets:

* This function takes in a pointer to an ip address string and converts the ip address to a numerical representation of the 4 octets and stores it in ipOctets
* This function starts by creating its own stack, then allocating space for a temporary substring variable used to parse the ip address.
* The function then prepares several registers for the operations that will be performed, such as zeroing out ecx, and edx which will used as counters.
* Then it loads the ip address string into edi, which was previously in ebp.
* Now that everything is ready the program moves into string\_ip\_parse\_loop:

## string\_ip\_parse\_loop:

* + This sub-function loop is the heart of the ip address parsing.
  + It loads the first byte of the ip address into al and increments ecx, the counter for each digit of each octet.
  + Then the function checks if the digit is a dot. If it is then this octet is complete, and it jumps to octet\_complete.
  + If not at the end, then it compares the digit to a 0. if it is a 0 then it jumps to null\_byte\_encountered.
  + Then the function checks the ecx counter. If it is greater than 4 then this octet is done, and it jumps to octet\_complete.
  + If it doesn’t meet any of the other conditions, we need to process the digit.
  + It copies the digit to the temporary string space allocated earlier, then starts the loop again.

## null\_byte\_encountered:

* + This sub-function checks where the null byte was encountered.
  + If it was encountered at the end, then that’s normal and it jumps to octet\_complete:
  + If it’s not at the end, then that’s a problem. Return -1 and jump to malformed\_ip\_address\_exit .

## octet\_complete:

* + This sub-function adds a null terminator to the temporary octal string and then sends the string to be converted to a number by calling cStrtoul.
  + It starts by moving 0 into al and then storing al in edi.
  + Next it saves the position in the ip string and the octal counter by pushing them to the stack .
  + The function then pushes the temporary string to the stack and calls cStrtoul.
  + It then checks if there were any error in the return value (eax), if there are it jumps to malformed\_ip\_address\_exit.
  + If there are no errors, it restores the octal counter and continues.
  + Then it copies the ip octet into the ip octet array through edi.
  + It increments the octet counter (edx).
  + The function then restores the position in the ip string (esi) by popping it from the stack.
  + Then the function resets the position in the temporary octal string and resets the octal digit counter, in order to process the next octal.
  + It compares the octal counter against 4 and if it is less than 4 it jumps to string\_ip\_parse\_loop.
  + Otherwise, the ip is parsed and returns 0 for a success.

## malformed\_ip\_address\_exit:

* + This sub-function destroys the stack created in cStrIP\_to\_Octets.

## cStrtoul:

* This function is responsible for converting an ascii string of numbers to integers.
* It starts by creating a stack for itself and allocates space for the multiply operand.
* It then points esi to the beginning of the string in ebp. Then it copies the string into edi.
* It starts the string\_length\_loop, which gets esi to the end of string.
* The function then subtracts the beginning (edi) of the string from the end of the string (ebx), which puts the length of the string in ebx.
* If the string length is not greater than 0 it jumps to premature\_exit.
* Then the function prepares eax, ecx, and edx to hold the current character, digit position, and final results respectively.
* Then it sets esi to the beginning of the string again and starts the digits\_count\_loop.

## string\_length\_loop:

* + This loop compares each digit in edi with 0 and keeps looping until it is found.

## digits\_count\_loop:

* + This sub-function takes a digit and multiplies by power of 10 then checks if it needs to do it again.
  + It loads the digit into al and decreases the string length counter.
  + It multiplies the digit by 10 and checks if it needs to multiply by any more powers of 10.
  + If it doesn’t it jumps to exponent\_loop\_skip.
  + If it does, then the program moves onto exponent\_loop.
  + It the end of

## exponent\_loop:

* + - This loop multiplies ecx by 10 and increments edi
    - Then it compares edi and ebx
    - If ebx is greater than edi it needs to keep looping, so it jumps to exponent\_loop.
    - If edi is greater then moves to exponent\_loop\_skip.

## exponent\_loop\_skip:

* + - This sub-function compares the digit to 48.
    - If it is greater than 48, it jumps to lower\_bound\_met.
    - If it’s not, then it sets eax to -1 and jumps to premature\_exit.

## lower\_bound\_met:

* + - This sub-function checks if the digit is lower than 57.
    - If it is, then it jumps to upper\_bound\_met.
    - If not, then it sets eax to -1 and jumps to premature\_exit.

## upper\_bound\_met:

* + - This sub-function subtracts 48 (ascii 0) from the digit to get the true digit value.
    - Then the function multiplies the digit by the powers of ten from earlier
    - The digit is then added to the final result in ebx
    - The function then compares the string length (ebx) with 0.
    - If it is greater than 0, it still has digits left to process, so it loops back to digits\_count\_loop, to start again with the next digit.
    - If the string length is 0, the final result is moved into eax (return value).
    - Then the program moves to premature\_exit.

## premature\_exit:

* + This sub-function smashes the stack created at the beginning of cStrtoul.