Ethernet and UART Loopback Transmission

Example Code Introduction for 64-bit NuMicro® Family

Document Information

Application	This example program demonstrates the bidirectional transfer between Ethernet and UART. If the user connects the UART TX/RX, it will enable continuous loopback transmission between Ethernet and UART.
BSP Version	MA35D1_Linux-5.10.y
Hardware	NuMaker-SOM-MA35D16A8 V2.0

The information described in this document is the exclusive intellectual property of Nuvoton Technology Corporation and shall not be reproduced without permission from Nuvoton.

Nuvoton is providing this document only for reference purposes of NuMicro microcontroller and microprocessor based system design. Nuvoton assumes no responsibility for errors or omissions.

All data and specifications are subject to change without notice.

For additional information or questions, please contact: Nuvoton Technology Corporation.

www.nuvoton.com

1. Overview

This example program demonstrates the remote control of multiple serial devices over a network, commonly used for industrial automation data transmission and remote environmental monitoring. In this demonstration, when the UART port of the MA35D1 series microprocessor receives data, it sends the received data through Ethernet to the corresponding port on the network. Conversely, when the MA35D1 receives data from Ethernet, it forwards the data to the corresponding UART port based on the Ethernet port from which the data has been received. This method enables data forwarding between different devices effectively.

2. Code Description

The code for the transfer between UART and Ethernet is located at *eth2uart.c*.

In the main() function, UARTs are initially configured, and successfully configured UARTs are added to a device list. Ethernet sockets are also created to initialize the network. Additionally, a thread (tcp_process) is created to handle data forwarding between UART and Ethernet.

```
int main(int argc, char **argv)
{
    unsigned int i;

    char file_name[50];
    int tcp_fd;

    int32_t ret;
    uint32_t uart_port_num;

    struct Objects_Opt obj;
    FILE *fp;

    _UART_PARS_s UART_Conf;

    /* config uart */
    for(i = 1; i <= TEST_PORT_NUM; i++)
    {
        uart_port_num = i;

        UART_Conf.baudrate = 115200;
        UART_Conf.datalen = 8;
        UART_Conf.stopbit = 1;
        UART_Conf.odd = 0;

        serial_info=open_tty(uart_port_num);
        if(serial_info == NULL)
        {
            continue;
        }

        if(set_tty_speed(serial_info, UART_Conf.baudrate))
        {
            continue;
        }

        if((ret=set_tty_parity(serial_info,UART_Conf.datalen,UART_Conf.odd,
        UART_Conf.stopbit))==0)
        {
            obj.fd = serial_info->fd;
            obj.port_type = UART;
            obj.port = (PORT_BASE+uart_port_num);

            /* added device to the device list table */
            if(!add_link(&obj,&tcp_max_fd,&tcp_link_head,&tcp_link_tail))
            {
```

```
                printf("\nuart add_link occure an error\n");
                while(1);
            }
        }
        else
        {
            printf("set uart%d fail! \n", i);
        }
    }

    for(i = 1; i <= TEST_PORT_NUM; i++)
    {
        /* Create Ethernet socket */
        fd[i] = create_socket(PORT_BASE+i);
    }

    printf("\n\n");

    /* Create thread to run demo */
    if(pthread_create(&tcp_thread_id,NULL,tcp_process,0) !=0 )
    {
        printf("UserTimer thread (create) error\n");
        while(1);
    }

    pthread_join(tcp_thread_id, NULL);

    return 0;
}
```

In the main() function, the open_tty() function is designed to open a serial device with a specified id, initialize its related information, and return a pointer to a _TTY_INFO_s structure. Here's the description of the code:

1. Allocate Memory:
    - The malloc() function allocates memory to store a _TTY_INFO_s structure. The variable ptty is used to store the pointer of the _TTY_INFO_s structure.
2. Initialize Memory:
    - memset() sets the memory pointed to by ptty to zero, ensuring that all fields of the _TTY_INFO_s structure are initialized to zero.
3. Open the Device:
    - Formats the name of the serial device as /dev/ttyS{id} and store it in ptty->name, where {id} is the passed id parameter.
    - The open function opens the device in read/write mode (O_RDWR) and without assigning it as a controlling terminal (O_NOCTTY). If opening fails, it frees the previously allocated memory, prints an error message, and returns NULL.

4. Set File Descriptor Flags:
   - fcntl() is used to set file descriptor flags. Here, it sets the flags to 0, which means disabling non-blocking mode and other flags.
5. Get Current Terminal Attributes:
   - tcgetattr() retrieves the current terminal attributes and stores them in ptty->otm.
6. Return Result:
   - Returns a pointer to the _TTY_INFO_s structure, which contains the information about the opened serial device.

```
_TTY_INFO_s *open_tty(uint32_t id)
{
    static _TTY_INFO_s *ptty;

    ptty = (_TTY_INFO_s *)malloc(sizeof(_TTY_INFO_s));
    if(ptty == NULL)
    {
        return NULL;
    }

    memset(ptty,0,sizeof(_TTY_INFO_s));

    sprintf(ptty->name,"/dev/ttyS%d",id);

    ptty->fd = open(ptty->name, O_RDWR | O_NOCTTY,0);
    if (ptty->fd <0)
    {
        free(ptty);
        printf("open tty%d fail\n",id);
        return NULL;
    }

    fcntl(ptty->fd, F_SETFL, 0);

    tcgetattr(ptty->fd,&ptty->otm);

    return ptty;
}
```

In the main() function, the set_tty_speed() function is used to set the baud rate for a specified serial device. It reads the current terminal attributes, configures them with the new baud rate, and applies the settings. Here's the description of the code:

1. Get Current Terminal Attributes:
   - tcgetattr() retrieves the current terminal attributes and stores them in ptty->ntm. If the retrieval fails, it prints an error message and returns 1.

2. Initialize “ntm” Structure:
   - bzero() is used to clear the ptty->ntm structure’s memory, ensuring all fields are initialized to zero.
3. Set Input and Output Baud Rate:
   - cfsetispeed() and cfsetospeed() are used to set the terminal’s input and output baud rates to the specified “speed”. The “speed” is set according to the input “baudrate” variable
4. Flush Input Buffer:
   - tcflush() is used to flush the terminal’s input buffer, discarding any unread data.
5. Apply New Terminal Attributes:
   - tcsetattr() applies the new terminal attributes immediately (TCSANOW).

```
int set_tty_speed(_TTY_INFO_s *ptty, uint32_t baudrate)
{
    uint32_t speed;

    if( tcgetattr(ptty->fd,&ptty->ntm) != 0)
    {
        printf("Setup Serial [%s] speed fail\n",ptty->name);
        return 1;
    }
    bzero(&ptty->ntm, sizeof(ptty->ntm));

    switch(baudrate)
    {
        case 1200:
            speed= B1200;
        break;
        case 9600:
            speed = B9600;
        break;
        case 19200:
            speed = B19200;
        break;
        case 115200:
            speed = B115200;
        break;
        case 921600:
            speed = B921600;
        break;
    }

    cfsetispeed(&ptty->ntm, speed);
    cfsetospeed(&ptty->ntm, speed);

    tcflush(ptty->fd, TCIFLUSH);
    tcsetattr(ptty->fd,TCSANOW,&ptty->ntm);
    return 0;
}
```

In main(), the set_tty_parity() function is used to configure serial port settings, including data bits, parity, and stop bits. The function details are as follows:

1. Get Current Terminal Attributes:
   - tcgetattr() is used to retrieve the current settings of the terminal associated with ptty->fd and store them in ptty->ntm. If retrieving the attributes fails, an error message is printed and 1 is returned.
2. Set Data Bits:
   - Uses ptty->ntm.c_cflag &= ~CSIZE to clear the existing data bit settings. c_cflag is used to control character size.
   - Sets the data bits to 7 (CS7) or 8 (CS8) based on “databits”.
   - If the data bits value is unsupported, print an error message and return 5.
3. Set Parity:
   - PARENB is used to enable the parity bit, and INPCK is used to enable input parity checking.
   - The function configures parity settings based on the parity value. For no parity (0), it disables parity checking. For odd parity (1), it enables odd parity. For even parity (2), it enables parity checking and sets it to even parity. If the parity value is unsupported, it prints an error message and returns 2.
4. Set Stop Bits:
   - Configures the stop bits to 1 (~CSTOPB) or 2 (CSTOPB) based on stopbits. If the chosen stop bits are unsupported, print an error message and return 3.
5. Set Other Terminal Attributes:
   - Sets the terminal to local mode and enable receiving (CLOCAL | CREAD).
   - Configures raw mode with c_lflag, c_oflag, and c_iflag (disables canonical mode, echo, and processing).
   - Configures VTIME and VMIN for non-blocking reads.
   - Flushes the input buffer with tcflush() and applies the new settings with tcsetattr().

```
int set_tty_parity(_TTY_INFO_s *ptty,uint32_t databits,uint32_t parity,uint32_t stopbits)
{
    if( tcgetattr(ptty->fd,&ptty->ntm) != 0)
    {
        printf("Setup Serial [%s] parity fail\n",ptty->name);
        return 1;
    }

    ptty->ntm.c_cflag &= ~CSIZE;
    switch (databits)
    {
```

```
        case 7:
            ptty->ntm.c_cflag |= CS7;
        break;
        case 8:
            ptty->ntm.c_cflag |= CS8;
        break;
        default:
            printf("Unsupported data size\n");
        return 5;
    }

    /* set parity */
    switch (parity)
    {
        case 0: // non-parity
            ptty->ntm.c_cflag &= ~PARENB;
            ptty->ntm.c_iflag &= ~INPCK;
        break;

        case 1: // odd-parity
            ptty->ntm.c_cflag |= (PARODD|PARENB);
            ptty->ntm.c_iflag |= INPCK;
        break;

        case 2: // even-parity
            ptty->ntm.c_cflag |= PARENB;
            ptty->ntm.c_cflag &= ~PARODD;
            ptty->ntm.c_iflag |= INPCK;
        break;

        default:
            printf("Unsupported parity\n");
        return 2;
    }

    /* stop-bits */
    switch (stopbits)
    {
        case 1:
            ptty->ntm.c_cflag &= ~CSTOPB;
        break;
        case 2:
            ptty->ntm.c_cflag |= CSTOPB;
        break;
        default:
            printf("Unsupported stop bits\n");
        return 3;
    }

    ptty->ntm.c_cflag |= (CLOCAL | CREAD);
    ptty->ntm.c_lflag &=~(ICANON | ECHO | ECHOE | ISIG);
    ptty->ntm.c_oflag &=~OPOST;
    ptty->ntm.c_iflag &=~(IXON | ICRNL);

    ptty->ntm.c_cc[VTIME] =0;
    ptty->ntm.c_cc[VMIN] = 1;
```

```
    tcflush(ptty->fd, TCIFLUSH);
    if (tcsetattr(ptty->fd,TCSANOW,&ptty->ntm) != 0)
    {
        printf("Setup Serial [%s] parity fail\n");
        return 4;
    }

    return 0;
}
```

In the main() function, the create_socket() function is designed to create and configure a TCP socket, bind it to a specified port, and start listening for incoming connection requests. Here's the description of the function:

1. Initialize struct “m_tcpsocket_addr” (This structure is used to set the address structure of the socket):
   - “memset” clears the m_tcpsocket_addr structure, ensuring all its bytes are set to zero.
   - “sin_family” is set to AF_INET, indicating that this is an IPv4 address.
   - “sin_addr.s_addr” is set to INADDR_ANY, which means the socket will accept connections from any IP.
2. Create the Socket:
   - The socket function creates a TCP socket. AF_INET specifies IPv4, SOCK_STREAM indicates a TCP socket, and 0 typically means the default protocol is used.
3. Set the Port and Bind the Socket:
   - “htons” converts the port number from host byte order to network byte order.
   - The bind function binds the socket to the specified port. If bind fails, it prints an error message and returns -1.
4. Start Listening for Connections:
   - The listen() function puts the socket in passive mode, ready to accept incoming connections. 10 specifies the maximum number of queued connections.
5. Add the device to the linked list and return the socket descriptor (fd):
   - In the function add_link(), this device will be added to the linked list to facilitate subsequent query and modification of the device status.
   - If all operations are successful, the function returns the created socket descriptor(fd).

```
int create_socket(unsigned short port)
{
    int fd;
    int Client_Sockfd;
    struct sockaddr_in m_tcpsocket_addr;
    struct sockaddr_in Client_addr;
    struct Objects_Opt obj;
```

```
    /* initial memory */
    memset(&m_tcpsocket_addr,0,sizeof(m_tcpsocket_addr));

    m_tcpsocket_addr.sin_family = AF_INET;

    m_tcpsocket_addr.sin_addr.s_addr = INADDR_ANY;

    fd = socket(AF_INET,SOCK_STREAM,0);

    if(-1 == fd)
    {
        printf("create tcp socket failed,errno = %d\n",errno);
        return -1;
    }

    m_tcpsocket_addr.sin_port = htons(port);
    if(bind(fd,(struct sockaddr *)&m_tcpsocket_addr,sizeof(m_tcpsocket_addr))==-1)
    {
        printf("Bind1 failed!\n");
        return -1;
    }

    if(listen(fd,10)==-1)
    {
        printf("Listen is failed!\n");
        return -1;
    }

    obj.port = port;
    obj.fd   = fd;
    obj.port_type = ETHERNET;

    if(!add_link(&obj,&tcp_max_fd,&tcp_link_head,&tcp_link_tail))
    {
        printf("\nadd_link occure an error\n");
        while(1);
    }

    return fd;
}
```

In the “tcp_process” thread, when a device connected via Ethernet is detected for the first time by MA35D1, it is added to the device list table. Conversely, if this device disconnects, it is removed from the device list.

Subsequently, when data is sent from Ethernet, the thread identifies the corresponding UART port based on the Ethernet port from which the data has been received. It then sends the data out through this UART port. Similarly, data received from UART ports is forwarded out through the corresponding Ethernet port using the same mapping from the device list table.

Here is an explanation of the thread's functionality:

1. Initialize net_fds:

    - Use FD_ZERO() to initialize the net_fds (fd_set) file descriptor set, making it empty.

- “fd_set” is used with the select() function to monitor multiple file descriptors. It allows you to keep track of which file descriptors are ready for reading, writing, or have errors.

2. Set File Descriptors to Monitor:

   - Traverse the list of “Links_Objects” nodes, and use FD_SET() to add each valid file descriptor to the net_fds set so that select will monitor these descriptors.

3. Wait for File Descriptors to be Ready:

   - Use select() to wait until at least one file descriptor is ready(when select() returns a value greater than 0). tcp_max_fd is the maximum file descriptor value plus one.

4. Handle select() Return Values:

   - Handle different scenarios based on the return value of select(): error(return value less than 0), timeout(return value equal to 0), or one or more file descriptors are ready(return value greater than 0).
   - When file descriptors are ready (return value greater than 0), process them according to their “port_type”.

5. Handle different “port_type”:

   - When “port_type == ETHERNET”, it indicates that a new Ethernet connection is established. Accept the new connection, add it to the linked list, and change this “port_type” to “CONNECT_TCP”.
   - When “port_type == UART”, it means that new data is received from UART. Read the data from UART and send it to the corresponding Ethernet endpoint.
   - When “port_type == CONNECT_TCP”, it signifies that new data is received on the Ethernet endpoint. Read the data from Ethernet and write it to the corresponding UART port based on the port number. If the size of the received data is 0, it means the Ethernet connection is closed. In this case, close the file descriptor, remove it from the set, and delete the connection from the linked list.

```
void *tcp_process (void *arg)
{
    int ret, uart_rev, tcp_rev, Client_Sockfd, eth_connect_fd;
    uint32_t i, r;
    struct Objects_Opt obj;
    struct sockaddr_in Client_addr;
    struct Links_Objects *p;
    socklen_t sin_size;
    fd_set net_fds;
    int wd;

    while(1)
    {
        /* initialize net_fds to zero */
        FD_ZERO(&net_fds);

        p =  tcp_link_head;
```

```
        for(;p != NULL;p = p->next)
        {
            if((p->opt).fd > 0)
            {
                /* add a file descriptor to net_fds */
                FD_SET((p->opt).fd,&net_fds);
            }
        }

        /* Wait file descriptors be set */
        ret = select(tcp_max_fd +1,&net_fds,NULL,NULL,NULL);

        if (ret < 0)
        {
            /* get number of ready file descriptors fail */
        }
        else if (ret == 0)
        {
            /* no file descriptors be set */
        }
        else if(ret > 0)
        {
            p =  tcp_link_head;

            for(;p != NULL;p = p->next)
            {
                if((p->opt).fd > 0)
                {
                    /* check if a fd is set in net_fds */
                    if (FD_ISSET((p->opt).fd, &net_fds))
                    {
                        if((p->opt).port_type == ETHERNET)
                        {   /* If a device is connected to the MA35D1,
                               add the new device to device list */

                            eth_connect_fd = accept((p->opt).fd, (struct sockaddr
*)&Client_addr, &sin_size);

                            if (eth_connect_fd <= 0)
                            {
                                printf("fd=%d accept error! \n",(p->opt).fd);
                                continue;
                            }

                            obj.fd = eth_connect_fd;
                            obj.port_type = CONNECT_TCP;
                            obj.port = (p->opt).port;

                            if(!add_link(&obj,&tcp_max_fd,&tcp_link_head,&tcp_link_tail))
                            {
                                printf("\nconnect tcp add_link occure an error\n");
                                while(1);
                            }

                        }
                        else if((p->opt).port_type == UART)
```

```
                    {   /* If the UART port receives data, the data is sent out
                           through Ethernet */

                        uart_rev = read((p->opt).fd,RxBuffer,MAX_PACKAGE);

                        if(uart_rev > 0)
                        {
                            struct Links_Objects *pp;
                            pp =  tcp_link_head;
                            for(;pp != NULL;pp = pp->next)
                            {
                                if((pp->opt).port_type == CONNECT_TCP)
                                {
                                    if((pp->opt).port == (p->opt).port)
                                    {
                                        send((pp->opt).fd, RxBuffer, uart_rev, 0);
                                    }
                                }
                            }
                        }

                    }
                    else if((p->opt).port_type == CONNECT_TCP)
                    {   /* If the Ethernet receives data, the data is sent out through
                           UART */

                        tcp_rev = recv((p->opt).fd,tcp_buf,sizeof(tcp_buf),0);

                        if (tcp_rev == -1)
                        {
                            printf("tcp receive error!\n");
                            while(1);
                        }
                        else if (tcp_rev == 0)
                        {   /* Ethernet connect lose */

                            close((p->opt).fd);

                            /* clear a file descriptor from a set */
                            FD_CLR((p->opt).fd, &net_fds);

                            /* delete the connection from the linked list */
                            del_link((p->opt).fd,&tcp_link_head,&tcp_link_tail);
                        }
                        else
                        {
                            struct Links_Objects *pp;

                            pp =  tcp_link_head;
                            for(;pp != NULL;pp = pp->next)
                            {
                                if((pp->opt).port_type == UART)
                                {
                                    if((pp->opt).port == (p->opt).port)
                                    {
                                        write((pp->opt).fd,tcp_buf,tcp_rev);
```

```
                                        }
                                    }
                                }
                            }
                        }
                    }

                }
            }

        }

    }

    pthread_exit(NULL);
}
```

3. Software and Hardware Requirements

3.1 Software Requirements

- BSP version
  - MA35D1_Linux-5.10.y

3.2 Hardware Requirements

- Circuit components
  - NuMaker-SOM-MA35D16A8 V2.0
- Pin Connect
  - Connect the RS232_11 to the PC COM Port

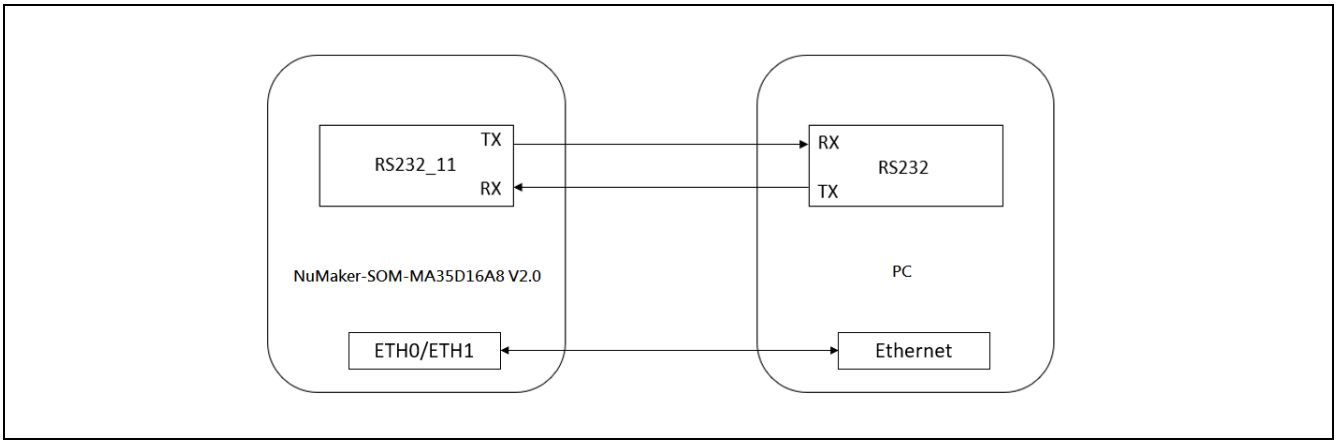


Figure 3-1 Pin Connect

4. Directory Information

The directory structure is shown below.

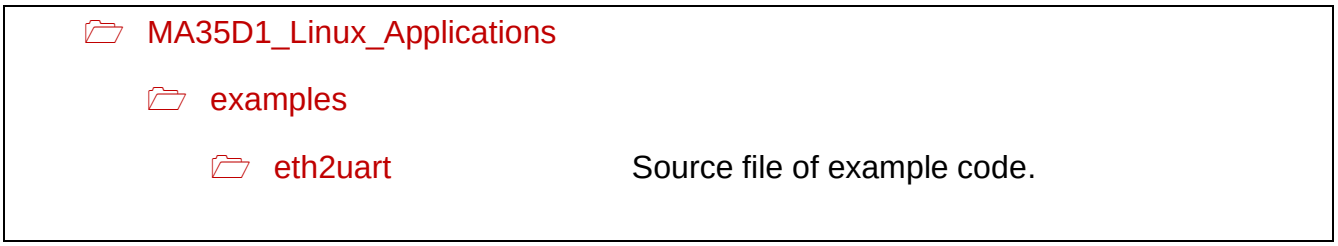
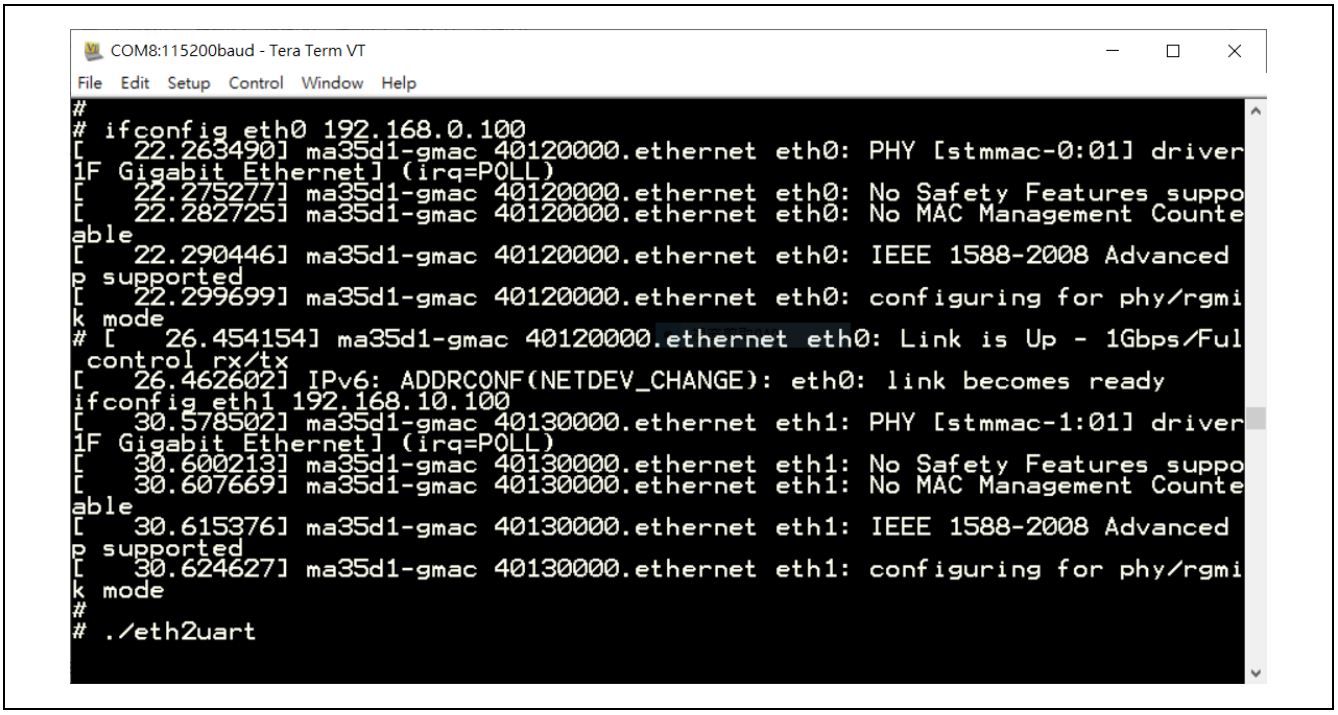


Figure 4-1 Directory Structure

5. Example Code Execution

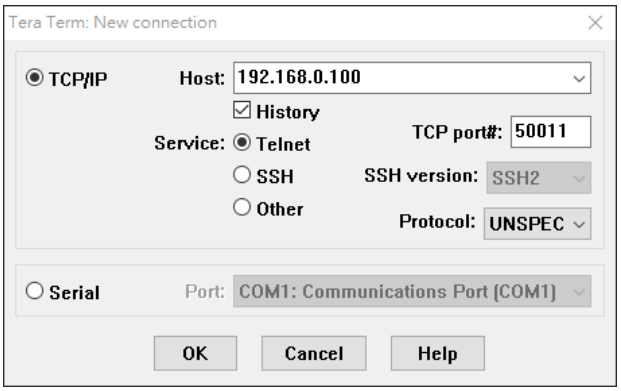
1. Browse the sample code folder as described in the Directory Information section and enter the “Make” command to compile example program. Then copy the compile file “eth2uart” to a USB drive.
2. Refer to the document (https://www.nuvoton.com/export/resource-files/en-us--MA35D1_Buildroot_Quick_Start_Rev1.01.pdf) to compile a Linux kernel program that can run on SOM board and execute the boot.
3. Connect RS232_11 on the SOM board to the PC, then open a terminal application (e.g. Putty or Teraterm) on the PC. Select the serial COM port and configure the baud rate to 115200.
4. Open the Serial COM Port using the Terminal tool (on the SOM board, use UART11 and UART16).
5. On the MA35D1, enter “ifconfig eth0 192.168.0.100” and “ifconfig eth1 192.168.10.100” to activate the network.
6. Copy “eth2uart” from the USB drive to the Linux directory on the SOM board, and then enter the command “./eth2uart” to execute the example program.



7. Use the Terminal tool to open the TCP/IP connection：

   Ethernet 0 IP address is 192.168.0.100, port numbers are from 50001 to 50016.
   Ethernet 1 IP address is 192.168.10.100, port numbers are from 50001 to 50016.

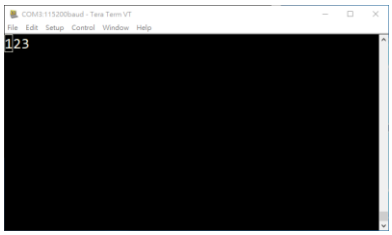
8. (Port numbers 50001 ~ 50016 correspond to UART1 ~ 16 for transmission or reception) (The SOM board uses UART11, so the port number used is 50011.)



9. Ethernet data transfer to UART demonstration:

   Enter “123” in the TCP/IP connection window, and the port number is 50011.

10. The UART11 serial COM port window will display “123”.



11. UART transmitting data to Ethernet demonstration:

    Enter "123" in the UART11 serial COM port.

12. The TCP/IP connection window with port number 50011 will display “123”.

6. Revision History

Date	Revision	Description
2024.06.18	1.00	Initial version.

Important Notice

Nuvoton Products are neither intended nor warranted for usage in systems or equipment, any malfunction or failure of which may cause loss of human life, bodily injury or severe property damage. Such applications are deemed, "Insecure Usage".

Insecure usage includes, but is not limited to: equipment for surgical implementation, atomic energy control instruments, airplane or spaceship instruments, the control or operation of dynamic, brake or safety systems designed for vehicular use, traffic signal instruments, all types of safety devices, and other applications intended to support or sustain life.

All Insecure Usage shall be made at customer's risk, and in the event that third parties lay claims to Nuvoton as a result of customer's Insecure Usage, customer shall indemnify the damages and liabilities thus incurred by Nuvoton.

Please note that all data and specifications are subject to change without notice.
All the trademarks of products and companies mentioned in this datasheet belong to their respective owners.