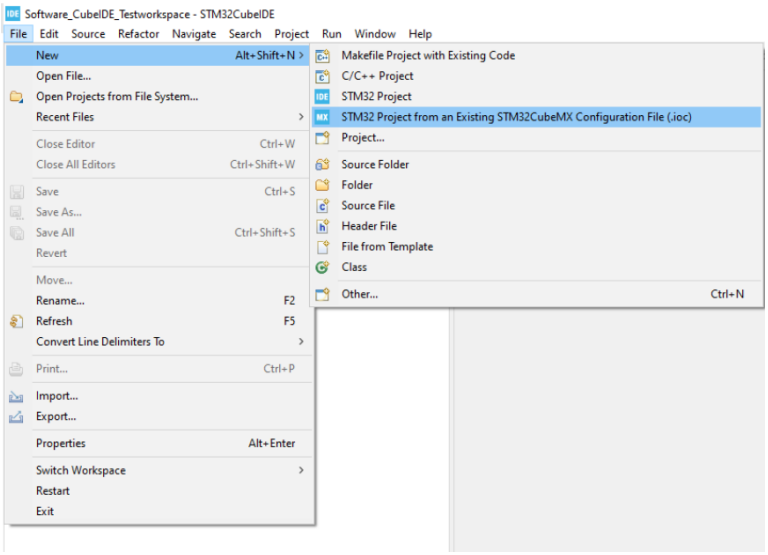
Topic:	TCP Echo-Server
Author:	PARA
Date:	17.09.2021

0. Before you start

This document will give you an overview of the source code for the example package TCP Echo-Server. For a more detailed explanation of how the microcontroller works, please refer to the STM32F429 Reference Manual (RM0090) and the description of STM32F4 HAL drivers (UM1725) provided by STMicroelectronics.

In this example you will work with the STM32CubeIDE by STMicroelectronics and use the HAL Library. Therefore, you need to **install STM32CubeIDE**.

After installation, open the IDE and create a new project from existing configuration file (.ioc).



In the next window, browse for the ioc-file “ExamplePackage_TCP_Echoserver.ioc”, which is provided with the example packages file, as the *STM32CubeMX .ioc file* and choose a project name. After pressing *Finish*, STM32CubeIDE will generate the project. Next, open the project folder in your file explorer and replace the directories *Core*, *Drivers*, *LWIP* and *Middlewares* with the directories of the same name provided in the example package. After refreshing your project in CubeIDE, you should be able to build and flash the source code.

The building and flashing process is similar to the System Workbench IDE by OpenSTM32. So make sure that you have read the documentation to the “ExamplePackage_GettingStarted”. Note, that the display size is this time specified in the file *Core/Inc/global_Display_Touch_HAL.h.*

1. Introduction

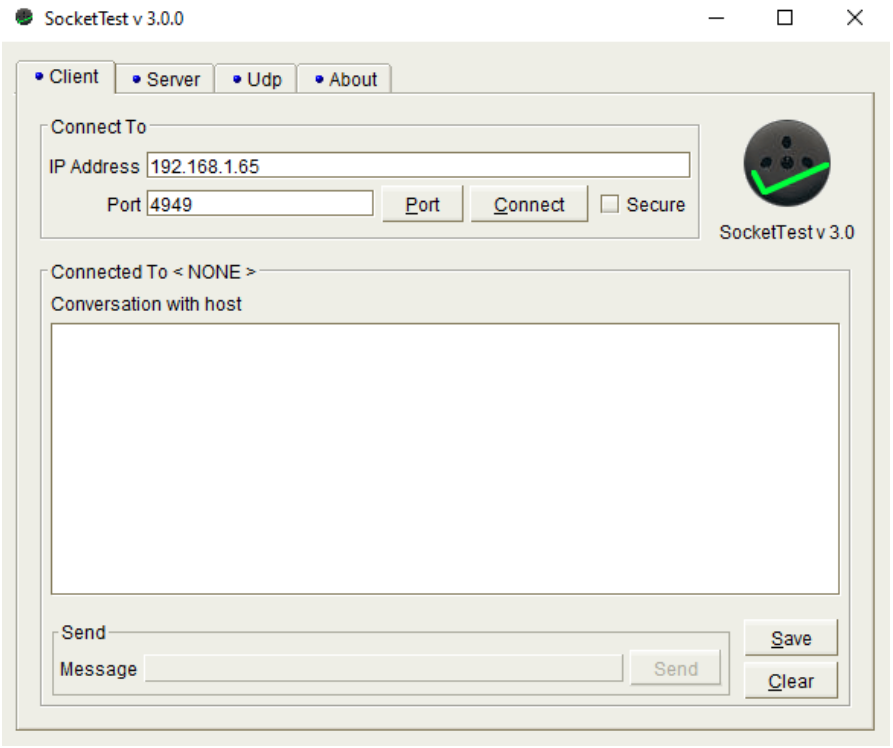
In this example package we will implement a simple echo-server. A TCP-connection to the display board will be established and the messages that you send to the board will be given back. To connect your display board via ethernet to a network **you need the Ethernet-Breakout-Board** provided by EBS-SYSTART.

For establishing the websocket on the STM32-microcontroller we use the open-source TCP/IP stack lwIP, which is implemented as third party middleware inside the CubeIDE.

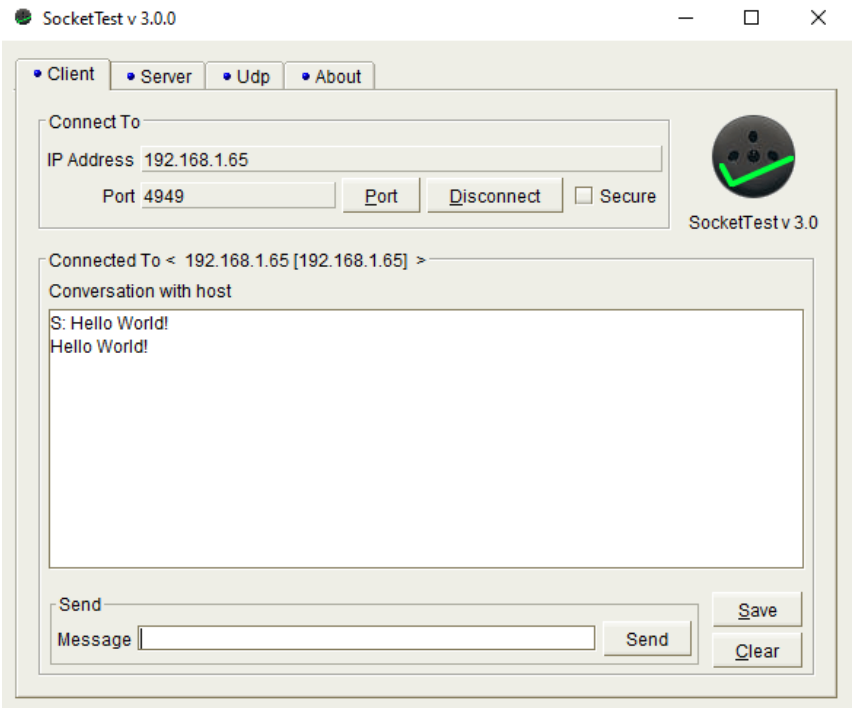
1.1 Connecting to the Webserver

After successfully flashing the code and connecting the display board to your local network with the Ethernet-Breakout-Board, you can establish a TCP connection with it. We recommend the free software SocketTest by akshath to test your newly created server.

Open the application and go to the tab *Client*. Here you need to enter the IP Address of the webserver and the port number. Those can be chosen by you and changed in the source code of the example package.



Press *Connect* and you should be able to send messages to the display board, which will be echoed back.



2. Explanation of Example Code

2.1 Main function

```
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_LWIP_Init();
  /* USER CODE BEGIN 2 */
  if(backlight_on){
      HAL_Delay(100);
      HAL_GPIO_WritePin(GPIOH, GPIO_PIN_6, GPIO_PIN_SET);
      HAL_GPIO_WritePin(GPIOG, GPIO_PIN_9, GPIO_PIN_SET);
  }
  SDRAM_FLASH_init();
  DMA2D_Display_init();
  DMA2D_Fill_Color(0xFFFFFFFF, Layer_1, Buffer_1);
  DMA2D_Draw_Image((HDP-320)/2, (VDP-240)/2, 320, 240, 255, DMA2D_REPLACE_ALPHA, (uint32_t)&image_data_ebssystart_logo, DMA2D_INPUT_ARGB8888, Layer_1, Buffer_1);
  HAL_Delay(4000);
  DMA2D_Fill_Color(0xFFFFFFFF, Layer_1, Buffer_1);
  DMA2D_Draw_Image((HDP-300)/2, (VDP-123)/2, 300, 123, 255, DMA2D_NO_MODIF_ALPHA, (uint32_t)&image_data_2in1_display_logo, DMA2D_INPUT_ARGB8888, Layer_1, Buffer_1);
  HAL_Delay(4000);
  DMA2D_Fill_Color(0xFFFFFFFF, Layer_1, Buffer_1);
  DMA2D_write_string("I'm echoing under", (HDP-17*16)/2, (VDP-11*79)/2, 0xFF000000, &courier_new, Layer_1, Buffer_1, Layer_1, Buffer_1);
  DMA2D_write_string("IP: 192.168.1.65", (HDP-17*16)/2, (VDP-11*79)/2+33, 0xFF000000, &courier_new, Layer_1, Buffer_1, Layer_1, Buffer_1);
  DMA2D_write_string("Port: 4949", (HDP-17*16)/2, (VDP-11*79)/2+56, 0xFF000000, &courier_new, Layer_1, Buffer_1, Layer_1, Buffer_1);
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* Read a received packet from the Ethernet buffers and send it to the lwIP for handling */
    ethernetif_input(&gnetif);
//    HAL_GPIO_TogglePin(GPIOH, GPIO_PIN_6);
//    HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_9);
//    HAL_Delay(2000);

    /* Handle timeouts */
    sys_check_timeouts();
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

We will start our walkthrough in the code at the main function since this gives a perfect overview of the steps that we will take. At the beginning, the used peripherals are initialized. *HAL_Init()* resets all peripherals and initializes the Flash interface and the Systick. Then, the various clocks are configured.

Afterwards the GPIO clocks are enabled and the lwIP stack is initialized. We will later take a closer look at this function.

Until now, we stepped through functions that are automatically generated by STM32CubeIDE according to changes you make in the ioc-file of your project. The next two initialization functions are part of our driver packages and do the setup for the SDRAM and the display. We need the SDRAM, because this is the location, where the data to be displayed on the screen will be stored.

The commands from line 109 to 118 implement a short start sequence with our logos. Afterwards the needed information to connect to the echo-server is displayed. This contains the IP-address and the port number.

Finally, the main loop is entered. In this loop, the ethernet buffer is periodically checked for received packages. Those will be handled as specified in the initialization of lwIP. The function

sys_check_timeouts must also be called periodically in the main loop to handle all timers for all protocols in the stack.

2.2 MX_LWIP_Init

```
53 /**
54   * LwIP initialization function
55   */
56 void MX_LWIP_Init(void)
57 {
58   /* IP addresses initialization */
59   IP_ADDRESS[0] = 192;
60   IP_ADDRESS[1] = 168;
61   IP_ADDRESS[2] = 1;
62   IP_ADDRESS[3] = 65;
63   NETMASK_ADDRESS[0] = 255;
64   NETMASK_ADDRESS[1] = 255;
65   NETMASK_ADDRESS[2] = 255;
66   NETMASK_ADDRESS[3] = 0;
67   GATEWAY_ADDRESS[0] = 192;
68   GATEWAY_ADDRESS[1] = 168;
69   GATEWAY_ADDRESS[2] = 1;
70   GATEWAY_ADDRESS[3] = 1;
71
72 /* USER CODE BEGIN IP_ADDRESSES */
73 /* USER CODE END IP_ADDRESSES */
74
75   /* Initilialize the LwIP stack without RTOS */
76   lwip_init();
77
78   /* IP addresses initialization without DHCP (IPv4) */
79   IP4_ADDR(&ipaddr, IP_ADDRESS[0], IP_ADDRESS[1], IP_ADDRESS[2], IP_ADDRESS[3]);
80   IP4_ADDR(&netmask, NETMASK_ADDRESS[0], NETMASK_ADDRESS[1] , NETMASK_ADDRESS[2], NETMASK_ADDRESS[3]);
81   IP4_ADDR(&gw, GATEWAY_ADDRESS[0], GATEWAY_ADDRESS[1], GATEWAY_ADDRESS[2], GATEWAY_ADDRESS[3]);
82
83   /* add the network interface (IPv4/IPv6) without RTOS */
84   netif_add(&gnetif, &ipaddr, &netmask, &gw, NULL, &ethernetif_init, &ethernet_input);
85
86   /* Registers the default network interface */
87   netif_set_default(&gnetif);
88
89   if (netif_is_link_up(&gnetif))
90   {
91     /* When the netif is fully configured this function must be called */
92     netif_set_up(&gnetif);
93   }
94   else
95   {
96     /* When the netif link is down this function must be called */
97     netif_set_down(&gnetif);
98   }
99
100  /* Set the link callback function, this function is called on change of link status*/
101  netif_set_link_callback(&gnetif, ethernetif_update_config);
102
103  /* Create the Ethernet link handler thread */
104
105 /* USER CODE BEGIN 3 */
106   tcp_echoserver_init();
107 /* USER CODE END 3 */
108 }
```

At the beginning of this initialization function we specify the IP addresses for the network connection of the display board. We used 192.168.1.65, but if this IP address is occupied in your network or you just want to use another one, you can change it. Make sure to alter the message, that will be shown on the display accordingly.

The subsequent functions are generated automatically by the CubeIDE and set up the data structures and handlers for the network interface.

Afterwards, we need to initialize the TCP server. This is done by calling *tcp_echoserver_init*.

2.3 tcp_echoserver_init

```
35 void tcp_echoserver_init(void){
36       pcb = tcp_new();
37       if(pcb != NULL){
38           err_t err = tcp_bind(pcb, IP_ADDR_ANY, 4949);
39           if(err == ERR_OK){
40               pcb = tcp_listen(pcb);
41               tcp_accept(pcb, accept_callback);
42           }
43           else{
44               memp_free(MEMP_TCP_PCB, pcb);
45           }
46       }
47 }
```

At first, a TCP protocol control block (TCP pcb) is created. This pcb will be bound to port 4949. Again, if you prefer another port, you can change this value. If the binding produced an error, the memory of the *pcb*-object is freed. Otherwise, we set the state of the connection to LISTEN, which means that it is able to accept incoming connections, and specify the function, which shall be called if an incoming connection is accepted. This function will be *accept_callback*.

2.4 accept_callback

```
49 static err_t accept_callback(void *arg, struct tcp_pcb *newpcb, err_t err){
50     err_t ret_err;
51     struct tcp_server_struct *es;
52
53     LWIP_UNUSED_ARG(arg);
54     LWIP_UNUSED_ARG(err);
55
56     /* set priority for the newly accepted tcp connection newpcb */
57     tcp_setprio(newpcb, TCP_PRIO_MIN);
58
59     /* allocate structure es to maintain tcp connection informations */
60     es = (struct tcp_server_struct *)mem_malloc(sizeof(struct tcp_server_struct));
61     if(es != NULL){
62         es->state = ES_ACCEPTED;
63         es->pcb = newpcb;
64         es->retries = 0;
65         es->p = NULL;
66
67         /* pass newly allocated es structure as argument to newpcb */
68         tcp_arg(newpcb, es);
69
70         tcp_recv(newpcb, receive_callback);
71         tcp_err(newpcb, error_callback);
72         tcp_poll(newpcb, poll_callback, 0);
73         ret_err = ERR_OK;
74     }
75     else{
76         /*  close tcp connection */
77         tcp_server_connection_close(newpcb, es);;
78         /* return memory error */
79         ret_err = ERR_MEM;
80     }
81
82     return ret_err;
83 }
```

One part of the *accept_callback* is to generate an instance of the *tcp_server_struct*, which will contain the information of the TCP connection. This instance will be passed as argument to the pcb, so that it can be accessed by the other callback functions.

The main task of the accept_callback is to specify these callback functions. In line 70 – 72 we specify the callbacks that shall be called if new data arrives (*receive_callback*), if a fatal error has occurred on the connection (*error_callback*) and the function that shall be called to poll the application (*poll_callback*).

2.5 receive_callback

In the receive function, different actions are taking place, depending on the server state. In error cases or if the connection is closed by a remote host, the memory of the data structures, which hold the information about the connection, will be freed.

The more interesting part is in line 125 – 160, where it is specified what shall happen with successfully received data.

```
125       else if(es->state == ES_ACCEPTED)
126       {
127         /* first data chunk in p->payload */
128         es->state = ES_RECEIVED;
129
130         /* store reference to incoming pbuf (chain) */
131         es->p = p;
132
133         /* initialize LwIP tcp_sent callback function */
134         tcp_sent(tpcb, sent_callback);
135
136         /* send back the received data (echo) */
137         tcp_server_send(tpcb, es);
138
139         ret_err = ERR_OK;
140       }
141       else if (es->state == ES_RECEIVED)
142       {
143         /* more data received from client and previous data has been already sent*/
144         if(es->p == NULL)
145         {
146           es->p = p;
147
148           /* send back received data */
149           tcp_server_send(tpcb, es);
150         }
151         else
152         {
153           struct pbuf *ptr;
154
155           /* chain pbufs to the end of what we recv'ed previously  */
156           ptr = es->p;
157           pbuf_chain(ptr,p);
158         }
159         ret_err = ERR_OK;
160       }
```

As you can see, the received data (variable *p*) won't be processed but only sent directly back.

2.6 error_callback

```
180 static void error_callback (void *arg, err_t err){
181       struct tcp_server_struct *es;
182
183       LWIP_UNUSED_ARG(err);
184
185       es = (struct tcp_server_struct *)arg;
186       if (es != NULL)
187       {
188         /*  free es structure */
189         mem_free(es);
190       }
191 }
```

The error callback only frees the memory of the *tcp_server_struct*.

2.7 poll_callback

```
199 static err_t poll_callback(void *arg, struct tcp_pcb *tpcb)
200 {
201   err_t ret_err;
202   struct tcp_server_struct *es;
203
204   es = (struct tcp_server_struct *)arg;
205   if (es != NULL)
206   {
207     if (es->p != NULL)
208     {
209       tcp_sent(tpcb, sent_callback);
210       /* there is a remaining pbuf (chain) , try to send data */
211       tcp_server_send(tpcb, es);
212     }
213     else
214     {
215       /* no remaining pbuf (chain)  */
216       if(es->state == ES_CLOSING)
217       {
218         /*  close tcp connection */
219         tcp_server_connection_close(tpcb, es);
220       }
221     }
222     ret_err = ERR_OK;
223   }
224   else
225   {
226     /* nothing to be done */
227     tcp_abort(tpcb);
228     ret_err = ERR_ABRT;
229   }
230   return ret_err;
231 }
```

The poll function will be called, when the connection is idle (i.e. not data is either transmitted or received). If there is still data in the *tcp_server_struct*, that should be sent (line 207), this will be done. If there is nothing to do, the connection will be closed (line 219 and 227).

3. Ideas for Exercise Project

Here, we want to give you some suggestions how you could modify our example code and make your own little project.

In our example package *Processing Touch Input and UART Communication* we implemented a basic UART communication which sends the touch input data to a connected PC. You could now implement a similar application which sends this information via Ethernet to a computer in your network.