

Tarefa Prática IOT 3

Nome: Carlos Delfino

Matricula: 202510110980368

Embarcotech Residência Profissional

Ano: 2025

Sumário

Sumário

Sumário.....	2
Introdução.....	2
Publisher (Servidor).....	2
Carga das bibliotecas e frameworks e constantes.....	3
Definição das assinaturas das funções.....	3
Inicialização do WiFi, GPIO e cliente MQTT.....	3
Resolução do nome do domínio do MQTT para IP.....	4
O loop principal (Super Loop).....	4
Função responsável pela publicação do estado do botão.....	5
Tratando a resolução de nome de domínio para IP.....	5
Tratando a conexão com o broker MQTT.....	6
Subscriber (Cliente).....	6
Estrutura que representa o estado do cliente.....	6
Definição de constantes relativas ao cliente (Subscriber).....	7
Funções auxiliares.....	7
Os Callbacks.....	8
Funções Auxiliares do MQTT.....	9
Callback que trata os dados entregues das subscrições.....	9
Callback de conexão.....	10
Função auxiliar para inicialização da conexão com o broker MQTT.....	10
Função de callback do DNS.....	11
Primeira parte da função main(), inicialização do firmware.....	11
Função main() inicialização do método de conexão.....	12
Super Loop.....	14

Introdução

Foram desenvolvidos dois firmwares, um para enviar a situação do botão para o MQTT, e outro para receber o estado do botão, mudando o estado de um LED como indicativo.

Optei por usar meu próprio Broker, que foi implementado em mqtt://mqtt.rapport.tec.br utilizando o Mosquitto, criei uma conta especial para uso do Monitor Wellingson:

Usuário: wellingson

Senha: embarcawellingson492

Abaixo apresento o código explicado.

Publisher (Servidor)

O Publisher é o código responsável por gerar os dados de telemetria do equipamento, através dele é obtido os dados no hardware e enviados separadamente para cada tópico no MQTT. Um publisher também pode ser um subscribe reagindo as atualizações no broker enviadas por outro publisher.

Carga das bibliotecas e frameworks e constantes

Na figura abaixo vemos as bibliotecas que foram importadas, e a declaração de cada variável necessária para o funcionamento da lógica. E duas macros (constantes) que identificam a porta do servidor MQTT e o pino do botão a ser usado.

```
c main.c > ...
1  #include <stdio.h>
2  #include <string.h>
3  #include "pico/stdlib.h"
4  #include "hardware/gpio.h"
5  #include "hardware/adc.h"
6  #include "pico/cyw43_arch.h"
7  #include "lwip/apps/mqtt.h"
8  #include "lwip/ip_addr.h"
9  #include "lwip/dns.h"
10
11 #define MQTT_BROKER_PORT 1883
12
13 // Configurações do Botão
14 #define BUTTON_STATE 5
15
16 // Variáveis Globais
17 static mqtt_client_t *mqtt_client;
18 static ip_addr_t broker_ip;
19 static char mqtt_button_topic[50];
20 static bool mqtt_connected = false;
21 static bool last_button_state = false;
22
```

Definição das assinaturas das funções

Logo a seguir temos as assinaturas das funções necessárias para organizar o código, trazendo assim clareza, declarar os callbacks.

```
23 // Protótipos de Funções
24 static void mqtt_connection_callback(mqtt_client_t *client, void *arg, mqtt_connection_status_t st
25
26 void publish_button_state(bool pressed);
27
28 void dns_check_callback(const char *name, const ip_addr_t *ipaddr, void *callback_arg);
29
```

Inicialização do WiFi, GPIO e cliente MQTT

Já na função **main()**, temos a inicialização do framework para uso do WiFi e do GPIO conforme o necessário para uso do botão. Também temos a inicialização do MQTT.

```

30 int main()
31 {
32     stdio_init_all();
33     sleep_ms(ms: 2000);
34     printf("\n=== Iniciando MQTT Button Monitor ===\n");
35
36     // Inicializa Wi-Fi
37     if (cyw43_arch_init()) {
38         printf("Erro na inicialização do Wi-Fi\n");
39         return -1;
40     }
41     cyw43_arch_enable_sta_mode();
42
43     printf("[Wi-Fi] Conectando...\n");
44     if (cyw43_arch_wifi_connect_timeout_ms(ssid: WIFI_SSID, pw: WIFI_PASSWORD, auth: CYW43_AUTH_WPA2
45         printf("[Wi-Fi] Falha na conexão Wi-Fi\n");
46         return -1;
47     } else {
48         printf("[Wi-Fi] Conectado com sucesso!\n");
49     }
50
51     // Configura GPIO do botão
52     gpio_init(gpio: BUTTON_STATE);
53     gpio_set_dir(gpio: BUTTON_STATE, out: GPIO_IN);
54     gpio_pull_up(gpio: BUTTON_STATE); // <<< ATENÇÃO: pull-up ativado
55
56     // Inicializa cliente MQTT
57     mqtt_client = mqtt_client_new();

```

Resolução do nome do dominio do MQTT para IP

No código a seguir temos a resolução do nome do broker mqtt, nela é cadastrado a função de callback **dns_check_callback** que trata a resolução do nome, caso tudo ocorra bem continua a execução.

```

58
59     // Resolve DNS do broker MQTT
60     err_t err = dns_gethostbyname(hostname: MQTT_BROKER, addr: &broker_ip, found: dns_check_callback
61     if (err == ERR_OK) {
62         dns_check_callback(name: MQTT_BROKER, ipaddr: &broker_ip, callback_arg: NULL);
63     } else if (err == ERR_INPROGRESS) {
64         printf("[DNS] Resolvendo...\n");
65     } else {
66         printf("[DNS] Erro ao resolver DNS: %d\n", err);
67         return -1;
68     }

```

O loop principal (Super Loop)

Então entramos no loop principal (ou super loop) nele processamos o as tarefas que são periódicas, primeiro mantemos a conexão wifi sempre ativa, em seguida, obtemos o estado do botão e verificamos se foi alterado em relação a ultima interação, caso tenha sido alterado é chamada a função que interagem com o broker MQTT. Então aguardamos 1 segundo para a próxima interação.

```

72 // Loop principal
73 while (true) {
74     // Atualiza tarefas de rede
75     cyw43_arch_poll();
76
77     // Lê o estado do botão
78     bool button_state = !gpio_get(gpio: BUTTON_STATE); // <<< Inverte porque é pull-up
79
80     // Se mudou de estado, publica
81     if (button_state != last_button_state) {
82         printf("[BOTÃO] Estado mudou para: %s\n", button_state ? "ON" : "OFF");
83         publish_button_state(pressed: button_state);
84         last_button_state = button_state;
85     }
86
87     sleep_ms(ms: 1000); // Ajuste conforme desejado
88 }
89
90 return 0;
91 }
92
93 // Callback da conexão MQTT

```

Função responsável pela publicação do estado do botão

Na função **publish_button_state(bool)** é recebido um parâmetro que representa o estado do botão, ela verifica se o firmware está conectado ao broker MQTT, constrói o tópico MQTT que irá armazenar o estado do botão, e faz a publicação do tópico reservando o retorno da função para analisar se a publicação foi correta.

```

103
104 void publish_button_state(bool pressed) {
105     if (!mqtt_connected) {
106         printf("[MQTT] Não conectado, não publicando estado da porta\n");
107         return;
108     }
109     char topic_button_state[50];
110     snprintf(topic_button_state, sizeof(topic_button_state), "%s/button", mqtt_button_topic);
111
112     const char *message = pressed ? "ON" : "OFF";
113
114     printf("[MQTT] Publicando: tópico='%s', mensagem='%s'\n", topic_button_state, message);
115
116     err_t err = mqtt_publish(client: mqtt_client, topic: topic_button_state, payload: message, payload_len: strlen(message));
117
118     if (err == ERR_OK) {
119         printf("[MQTT] Publicação enviada com sucesso\n");
120     } else {
121         printf("[MQTT] Erro ao publicar: %d\n", err);
122     }
123 }

```

Tratando a resolução de nome de domínio para IP

A função **dns_check_callback(const char, const ip_addr_t, void)** que é responsável pelo tratamento da obtenção do número IP do servidor.

```

125 // Callback de DNS
126 void dns_check_callback(const char *name, const ip_addr_t *ipaddr, void *callback_arg) {
127     if (ipaddr != NULL) {
128         broker_ip = *ipaddr;
129         printf("[DNS] Resolvido: %s -> %s\n", name, ipaddr_ntoa(addr: ipaddr));
130
131         struct mqtt_connect_client_info_t ci = {
132             .client_id = "pico-client",
133             .keep_alive = 60,
134             .client_user = NULL,
135             .client_pass = NULL,
136             .will_topic = NULL,
137             .will_msg = NULL,
138             .will_qos = 0,
139             .will_retain = 0
140         };
141
142         printf("[MQTT] Conectando ao broker...\n");
143         mqtt_client_connect(client: mqtt_client, ipaddr: &broker_ip, port: MQTT_BROKER_PORT, cb: mqtt
144     } else {
145         printf("[DNS] Falha ao resolver DNS para %s\n", name);
146     }
147 }

```

Tratando a conexão com o broker MQTT

Já a função `mqtt_connection_callback(mqtt_client_t, void, mqtt_connection_status_t)` trata a conexão com o broker setando o flag `mqtt_connected` conforme o resultado da tentativa de conexão.

```

93 // Callback de conexão MQTT
94 static void mqtt_connection_callback(mqtt_client_t *client, void *arg, mqtt_connection_status_t st
95     if (status == MQTT_CONNECT_ACCEPTED) {
96         printf("[MQTT] Conectado ao broker!\n");
97         mqtt_connected = true;
98     } else {
99         printf("[MQTT] Falha na conexão MQTT. Código: %d\n", status);
100         mqtt_connected = false;
101     }
102 }

```

Subscriber (Cliente)

O Subscriber irá monitorar o broker para reagir as atualizações do tópico de interesse. E como dito no Publisher também pode se comportar como um Publisher.

Como estou aprendendo, cada código adoto uma abordagem nova no intuito de analisar sugestões e exemplos, tendo assim uma diversificação na lógica usada.

Estrutura que representa o estado do cliente

Já no subscriber adotei um struct para conter o estado do cliente e informações da conexão, como informações do cliente, os dados recebidos do tópico, o tópico que está sendo tratado naquele momento, seu comprimento, o endereço do broker se está conectado, quantos subscribers estão conectados, aqui neste parâmetro mostra que um subscriber também pode ser um publisher.

```

32 typedef struct {
33     mqtt_client_t* mqtt_client_inst;
34     struct mqtt_connect_client_info_t mqtt_client_info;
35     char data[MQTT_OUTPUT_RINGBUF_SIZE];
36     char topic[MQTT_TOPIC_LEN];
37     uint32_t len;
38     ip_addr_t mqtt_broker_address;
39     bool connect_done;
40     int subscribe_count;
41     bool stop_client;
42 } MQTT_CLIENT_DATA_T;
43

```

Definição de constantes relativas ao cliente (Subscriber)

Como pode ser visto no código intuitivamente, é definido o tempo de manutenção da conexão, o QOS quanto ao método de subscrição e publicação dos dados, o tópico que indicará se o cliente está online, e se será um tópico único para informar se está online ou multiplos.

```

45 #define MQTT_KEEP_ALIVE_S 60
46
47 // qos passed to mqtt_subscribe
48 // At most once (QoS 0)
49 // At least once (QoS 1)
50 // Exactly once (QoS 2)
51 #define MQTT_SUBSCRIBE_QOS 1
52 #define MQTT_PUBLISH_QOS 1
53 #define MQTT_PUBLISH_RETAIN 0
54
55 // topic used for last will and testament
56 #define MQTT_WILL_TOPIC "/online"
57 #define MQTT_WILL_MSG "0"
58 #define MQTT_WILL_QOS 1
59
60 #ifndef MQTT_BASE_TOPIC
61 #define MQTT_BASE_TOPIC "rack_inteligente"
62 #endif
63
64 // Set to 1 to add the client name to topics, to support multiple devices using the same server
65 #ifndef MQTT_UNIQUE_TOPIC
66 #define MQTT_UNIQUE_TOPIC 1
67 #endif
68

```

Funções auxiliares

Já no cliente (Subscriber) optei por definir as funções antes da função **main()** que as chamam, assim não preciso definir as assinaturas das funções.

Primeiro defini a função **full_topic(MQTT_CLIENT_DATA_T)** que constrói a hierarquia do tópico, assim fica mais flexível quando se lida com mais de um tópico.

Já a função **control_led(bool)** é a função responsável pelo estado físico do LED controlado pelo respectivo tópico quando ocorre chama a função.


```

69
70 static const char *full_topic(MQTT_CLIENT_DATA_T *state, const char *name) {
71 #if MQTT_UNIQUE_TOPIC
72     static char full_topic[MQTT_TOPIC_LEN];
73     snprintf(full_topic, sizeof(full_topic), "%s%s", state->mqtt_client_info.client_id, name);
74     return full_topic;
75 #else
76     return name;
77 #endif
78 }
79
80 static void control_led(bool on) {
81     // Publish state on /state topic and on/off led board
82     const char* message = on ? "ON" : "OFF";
83     printf("Control LED: %s\n", message);
84     if (on)
85     {
86         gpio_put(gpio: LED_RED_PIN, value: 1);
87         gpio_put(gpio: LED_GREEN_PIN, value: 0);
88     }
89     else
90     {
91         gpio_put(gpio: LED_RED_PIN, value: 0);
92         gpio_put(gpio: LED_GREEN_PIN, value: 1);
93     }
94 }

```

Os Callbacks

As funções de callback são responsáveis por tratar eventos gerados pelo framework MQTT, temos três funções em questão, **pub_request_cb(void, err_t)** que trata as requisições de publicação de tópicos, e no nosso caso apenas alertamos quanto a algum erro que tenha ocorrido, sem nenhum outro tratamento.

Temos a função **sub_request_cb(void, err_t)** que trata as requisições de subscrição, primeiro ela alerta se houve algum erro, caso verdadeiro emite um sinal de panico que interrompe o processamento. Caso esteja tudo bem ela processa a subscrição no tópico no qual ela foi associada somando 1 ao contador de subscrição.

Já a função **sub_request_cb(void, err_t)** que trata as requisições de subscrição, primeiro ela alerta se houve algum erro, caso verdadeiro emite um sinal de panico que interrompe o processamento. Caso esteja tudo bem, ela subtrai 1 do contador de subscrição, e analisa se a contagem está correta, através do **assert** e do if que também verifica se o estado do cliente MQTT é como desligado.


```

96 static void pub_request_cb(__unused void *arg, err_t err) {
97     if (err != 0) {
98         printf("pub_request_cb failed %d", err);
99     }
100 }
101
102 static void sub_request_cb(void *arg, err_t err) {
103     MQTT_CLIENT_DATA_T* state = (MQTT_CLIENT_DATA_T*)arg;
104     if (err != 0) {
105         panic(fmt: "subscribe request failed %d", err);
106     }
107     state->subscribe_count++;
108 }
109
110 static void unsub_request_cb(void *arg, err_t err) {
111     MQTT_CLIENT_DATA_T* state = (MQTT_CLIENT_DATA_T*)arg;
112     if (err != 0) {
113         panic(fmt: "unsubscribe request failed %d", err);
114     }
115     state->subscribe_count--;
116     assert(state->subscribe_count >= 0);
117
118     // Stop if requested
119     if (state->subscribe_count <= 0 && state->stop_client) {
120         mqtt_disconnect(client: state->mqtt_client_inst);
121     }
122 }
123

```

Funções Auxiliares do MQTT

A função **sub_unsub_topics(MQTT_CLIENT_DATA_T)** é uma função que auxilia a subscrição ou descadastramento dos tópicos, associado o respetivo callback como citado acima.

```

124 static void sub_unsub_topics(MQTT_CLIENT_DATA_T* state, bool sub) {
125     mqtt_request_cb_t cb = sub ? sub_request_cb : unsub_request_cb;
126     mqtt_sub_unsub(client: state->mqtt_client_inst, topic: full_topic(state, name: "/exit"), qos: MQTT_QOS_1);
127     mqtt_sub_unsub(client: state->mqtt_client_inst, topic: full_topic(state, name: "/door"), qos: MQTT_QOS_1);
128 }
129

```

Callback que trata os dados entregues das subscrições

Temos um callback que é extremamente importante, ele é responsável por monitorar as inscrições em tópicos e tratar quando chegam dados que foram atualizados nos respectivos tópicos. A função em questão se chama **mqtt_incoming_data_cb(void, const u8_t, u16_t, u8_t)** e processa as atualizações de cada tópico, extraíndo o ramo final do tópico e tratando adequadamente os dados recebidos.

```

130 static void mqtt_incoming_data_cb(void *arg, const u8_t *data, u16_t len, u8_t flags) {
131     MQTT_CLIENT_DATA_T* state = (MQTT_CLIENT_DATA_T*)arg;
132     #if MQTT_UNIQUE_TOPIC
133         const char *basic_topic = state->topic + strlen(state->mqtt_client_info.client_id);
134     #else
135         const char *basic_topic = state->topic;
136     #endif
137     strncpy(state->data, (const char *)data, len);
138     state->len = len;
139     state->data[len] = '\0';
140
141     printf("Topic: %s, Basic Topic: %s, Message: %s\n", state->topic, basic_topic, state->data);
142     if (strcmp(basic_topic, "/door") == 0)
143     {
144         printf("Door topic: %s\n", state->data);
145         if (lwip_stricmp(str1: (const char *)state->data, str2: "ON") == 0 || strcmp((const char *)s
146             control_led(on: true);
147         else if (lwip_stricmp(str1: (const char *)state->data, str2: "OFF") == 0 || strcmp((const ch
148             control_led(on: false);
149     } else if (strcmp(basic_topic, "/exit") == 0) {
150         state->stop_client = true; // stop the client when ALL subscriptions are stopped
151         sub_unsub_topics(state, sub: false); // unsubscribe
152     }
153 }

```

Callback de conexão

```

160 static void mqtt_connection_cb(mqtt_client_t *client, void *arg, mqtt_connection_status_t status) {
161     MQTT_CLIENT_DATA_T* state = (MQTT_CLIENT_DATA_T*)arg;
162     if (status == MQTT_CONNECT_ACCEPTED) {
163         state->connect_done = true;
164         sub_unsub_topics(state, sub: true); // subscribe;
165
166         // indicate online
167         if (state->mqtt_client_info.will_topic) {
168             mqtt_publish(client: state->mqtt_client_inst, topic: state->mqtt_client_info.will_topic,
169         }
170     } else if (status == MQTT_CONNECT_DISCONNECTED) {
171         if (!state->connect_done) {
172             panic(fmt: "Failed to connect to mqtt server");
173         }
174     }
175     else {
176         panic(fmt: "Unexpected status");
177     }
178 }

```

Função auxiliar para inicialização da conexão com o broker MQTT

```

179
180 static void start_client(MQTT_CLIENT_DATA_T *state) {
181 #if LWIP_ALTCP && LWIP_ALTCP_TLS
182     const int port = MQTT_TLS_PORT;
183     printf("Using TLS\n");
184 #else
185     const int port = MQTT_PORT;
186     printf("Warning: Not using TLS\n");
187 #endif
188
189     state->mqtt_client_inst = mqtt_client_new();
190     if (!state->mqtt_client_inst) {
191         panic(fmt: "MQTT client instance creation error");
192     }
193     printf("IP address of this device %s\n", ipaddr_ntoa(addr: &(netif_list->ip_addr)));
194     printf("Connecting to mqtt server at %s\n", ipaddr_ntoa(addr: &state->mqtt_broker_address));
195
196     cyw43_arch_lwip_begin();
197     if (mqtt_client_connect(client: state->mqtt_client_inst, ipaddr: &state->mqtt_broker_address, po
198         | panic(fmt: "MQTT broker connection error");
199     }
200     printf("Connected to mqtt server\n");
201 #if LWIP_ALTCP && LWIP_ALTCP_TLS
202     // This is important for MBEDTLS_SSL_SERVER_NAME_INDICATION
203     mbedtls_ssl_set_hostname(altcp_tls_context(state->mqtt_client_inst->conn), MQTT_BROKER);
204 #endif
205     mqtt_set_inpub_callback(client: state->mqtt_client_inst, pub_cb: mqtt_incoming_publish_cb, data_
206     printf("Subscribed to topics\n");
207     cyw43_arch_lwip_end();
208 }
209

```

Função de callback do DNS

```

209
210 // Call back with a DNS result
211 static void dns_found(const char *hostname, const ip_addr_t *ipaddr, void *arg) {
212     MQTT_CLIENT_DATA_T *state = (MQTT_CLIENT_DATA_T*)arg;
213     if (ipaddr) {
214         state->mqtt_broker_address = *ipaddr;
215         start_client(state);
216     } else {
217         panic(fmt: "dns request failed");
218     }
219 }
220

```

Primeira parte da função main(), inicialização do firmware

```

221 int main(void) {
222     stdio_init_all();
223
224     // Aguarda a conexão da serial
225     while (!stdio_usb_connected())
226     {
227         sleep_ms(ms: 1000);
228     }
229
230     gpio_init(gpio: LED_RED_PIN);
231     gpio_init(gpio: LED_GREEN_PIN);
232     gpio_set_dir(gpio: LED_RED_PIN, out: GPIO_OUT);
233     gpio_set_dir(gpio: LED_GREEN_PIN, out: GPIO_OUT);
234
235     printf("mqtt client starting\n");
236
237     static MQTT_CLIENT_DATA_T state;
238
239     if (cyw43_arch_init()) {
240         panic(fmt: "Failed to initialize CYW43");
241     }
242

```

Função **main()** inicialização do método de conexão

Neste estágio é inicializado a metodologia de conexão ao MQTT, o uso do TLS ou SSL


```

242
243     // Generate a unique name, e.g. pico1234
244     char client_id_buf[sizeof(MQTT_BASE_TOPIC)];
245     memcpy(&client_id_buf[0], MQTT_BASE_TOPIC, sizeof(MQTT_BASE_TOPIC) - 1);
246     client_id_buf[sizeof(client_id_buf) - 1] = 0;
247     printf("Device name %s\n", client_id_buf);
248
249     state.mqtt_client_info.client_id = client_id_buf;
250     state.mqtt_client_info.keep_alive = MQTT_KEEP_ALIVE_S; // Keep alive in sec
251     #if defined(MQTT_USERNAME) && defined(MQTT_PASSWORD)
252         state.mqtt_client_info.client_user = MQTT_USERNAME;
253         state.mqtt_client_info.client_pass = MQTT_PASSWORD;
254     #else
255         state.mqtt_client_info.client_user = NULL;
256         state.mqtt_client_info.client_pass = NULL;
257     #endif
258     static char will_topic[MQTT_TOPIC_LEN];
259     strncpy(will_topic, full_topic(state: &state, name: MQTT_WILL_TOPIC), sizeof(will_topic));
260     state.mqtt_client_info.will_topic = will_topic;
261     state.mqtt_client_info.will_msg = MQTT_WILL_MSG;
262     state.mqtt_client_info.will_qos = MQTT_WILL_QOS;
263     state.mqtt_client_info.will_retain = true;
264     #if LWIP_ALTCP && LWIP_ALTCP_TLS
265         // TLS enabled
266         #ifdef MQTT_CERT_INC
267             static const uint8_t ca_cert[] = TLS_ROOT_CERT;
268             static const uint8_t client_key[] = TLS_CLIENT_KEY;
269             static const uint8_t client_cert[] = TLS_CLIENT_CERT;
270             // This confirms the identity of the server and the client
271             state.mqtt_client_info.tls_config = altcp_tls_create_config_client_2wayauth(ca_cert, sizeof(ca_
272             client_key, sizeof(client_key), NULL, 0, client_cert, sizeof(client_cert));
273         #if ALTCP_MBEDTLS_AUTHMODE != MBEDTLS_SSL_VERIFY_REQUIRED
274             WARN_printf("Warning: tls without verification is insecure\n");
275         #endif
276     #else
277         state->client_info.tls_config = altcp_tls_create_config_client(NULL, 0);
278         WARN_printf("Warning: tls without a certificate is insecure\n");
279     #endif
280 #endif

```

```

281
282     cyw43_arch_enable_sta_mode();
283     if (cyw43_arch_wifi_connect_timeout_ms(ssid: WIFI_SSID, pw: WIFI_PASSWORD, auth: CYW43_AUTH_WPA2
284         panic(fmt: "Failed to connect");
285     }
286     printf("\nConnected to Wifi\n");
287
288     // We are not in a callback so locking is needed when calling lwip
289     // Make a DNS request for the MQTT server IP address
290     cyw43_arch_lwip_begin();
291     int err = dns_gethostbyname(hostname: MQTT_BROKER, addr: &state.mqtt_broker_address, found: dns_
292     cyw43_arch_lwip_end();
293
294     if (err == ERR_OK) {
295         // We have the address, just start the client
296         start_client(state: &state);
297     } else if (err != ERR_INPROGRESS) { // ERR_INPROGRESS means expect a callback
298         panic(fmt: "dns request failed");
299     }
300

```

Função main() - Super Loop

Finalmente temos o super loop que mantém a conexão ativa e aguarda por eventos de dados por até 10000ms (dez mil milissegundos)

```
300
301     while (!state.connect_done || mqtt_client_is_connected(client: state.mqtt_client_inst)) {
302         cyw43_arch_poll();
303         cyw43_arch_wait_for_work_until(until: make_timeout_time_ms(ms: 10000));
304     }
305
```

As configurações do projeto, CMakeFile.txt

O arquivo CmakeFile.txt é responsável por gerar o ambiente e orquestrado para a compilação adequada do firmware.

Vamos analisar apenas um dos CmakeFile.txt o que está mais complexo.

Configurações do ambiente

Nesta etapa inicial o ambiente é preparado, sendo feita a definição da placa que será usada, e qual SDK e Ferramentas de compilação serão usados.

```
1  # == DO NOT EDIT THE FOLLOWING LINES for the Raspberry Pi Pico VS Code Extension to work ==
2  if(WIN32)
3      set(USERHOME $ENV{USERPROFILE})
4  else()
5      set(USERHOME $ENV{HOME})
6  endif()
7  set(sdkVersion 2.1.1)
8  set(toolchainVersion 14_2_Rel1)
9  set(picotoolVersion 2.1.1)
10 set(picoVscode ${USERHOME}/.pico-sdk/cmake/pico-vscode.cmake)
11 if (EXISTS ${picoVscode})
12     include(${picoVscode})
13 endif()
14 # =====
15 set(PICO_BOARD pico_w CACHE STRING "Board type")
16
17 # CMake configuration for firmware_client_mqtt
18 cmake_minimum_required(VERSION 3.13)
19
20 set(CMAKE_C_STANDARD 11)
21 set(CMAKE_CXX_STANDARD 17)
22 set(CMAKE_EXPORT_COMPILE_COMMANDS ON)
```

Parametrizando a conexão

A seguir parametrizamos as macros de conexão com valores padrões e obitidas no arquivo env.cmaker

```

30
31 # Variáveis default (caso não estejam no .env)
32 if(NOT DEFINED ENV{WIFI_SSID})
33     message(WARNING "Variável WIFI_SSID não definida no .env.")
34     set(ENV{WIFI_SSID} "ArvoreDosSaberes")
35 endif()
36 if(NOT DEFINED ENV{WIFI_PASSWORD})
37     message(WARNING "Variável WIFI_PASSWORD não definida no .env.")
38     set(ENV{WIFI_PASSWORD} "Arduino2022")
39 endif()
40 if(NOT DEFINED ENV{MQTT_BROKER})
41     message(WARNING "Variável MQTT_BROKER não definida no .env.")
42     set(ENV{MQTT_BROKER} "mqtt.rapport.tec.br")
43 endif()
44 if(NOT DEFINED ENV{MQTT_BASE_TOPIC})
45     message(WARNING "Variável MQTT_BASE_TOPIC não definida no .env.")
46     set(ENV{MQTT_BASE_TOPIC} "rack_inteligente")
47 endif()
48 if(NOT DEFINED ENV{MQTT_RACK_NUMBER})
49     message(FATAL_ERROR "Variável MQTT_RACK_NUMBER não definida no .env. Favor, defina essa variável")
50 endif()
51
52 set(WIFI_SSID "$ENV{WIFI_SSID}")
53 set(WIFI_PASSWORD "$ENV{WIFI_PASSWORD}")
54 set(MQTT_BROKER "$ENV{MQTT_BROKER}")
55 set(MQTT_BASE_TOPIC "$ENV{MQTT_BASE_TOPIC}")
56 set(MQTT_RACK_NUMBER "$ENV{MQTT_RACK_NUMBER}")

```

Parametrização final

Na parametrização final é carregado o arquivo auxiliar do sdk do RP Pico, define as linguagens que serão usadas para compilar o projeto, no caso C, C++ e Assembly. Define-se o nome do executável gerado e sua versão, desativa a porta serial e ativa porta serial USB. Carrega as bibliotecas necessárias. Define os diretórios de inclusão de headers e finalmente transfere as macros para o estágio de compilação para que sejam visíveis internamente no código C/C++


```

69 set(MQTT_RACK_NUMBER "${MQTT_RACK_NUMBER}" CACHE INTERNAL "MQTT Rack Number for examples ")
70
71 # Pull in Raspberry Pi Pico SDK (must be before project)
72 include(pico_sdk_import.cmake)
73
74 project(firmware_client_mqtt C CXX ASM)
75
76 # Initialise the Raspberry Pi Pico SDK
77 pico_sdk_init()
78
79 add_executable(firmware_client_mqtt
80     firmware_client_mqtt.c
81 )
82
83 pico_set_program_name(firmware_client_mqtt "firmware_client_mqtt")
84 pico_set_program_version(firmware_client_mqtt "0.1")
85
86 pico_enable_stdio_uart(firmware_client_mqtt 0)
87 pico_enable_stdio_usb(firmware_client_mqtt 1)
88
89 target_link_libraries(firmware_client_mqtt
90     pico_stdlib
91     pico_cyw43_arch_lwip_threadsafe_background
92     pico_lwip_mqtt
93 )
94
95 target_include_directories(firmware_client_mqtt PRIVATE ${CMAKE_CURRENT_LIST_DIR})
96
97 # Wi-Fi and MQTT configuration (edit as needed)
98 target_compile_definitions(firmware_client_mqtt PRIVATE
99     WIFI_SSID="\${WIFI_SSID}\\"
100     WIFI_PASSWORD="\${WIFI_PASSWORD}\\"
101     MQTT_BROKER="\${MQTT_BROKER}\\"
102     MQTT_BASE_TOPIC="\${MQTT_BASE_TOPIC}\\"
103     MQTT_RACK_NUMBER="\${MQTT_RACK_NUMBER}\\"
104 )
105
106 pico_add_extra_outputs(firmware_client_mqtt)
107

```