Aluno do Embarcatech_37 no IFMA

Nome: Manoel Felipe Costa Furtado

Matrícula: 20251RSE.MTC0086

Atividade 01 – Referente ao capítulo 01 da unidade 02

Tema do Capítulo - IEEE 802.15.4, Bluetooth, Zigbee

Prazo dia 01/06/2025 as 23:59

Enunciado: Analise o conjunto de arquivos fornecidos (server_common.c, server.c, server_with_wifi.c, client.c, btstack_config.h, lwipopts.h, btstack_config.h) e reflita sobre os seguintes pontos. Não altere o código: apenas estude-o e responda às questões de forma clara e objetiva.

Responda as Perguntas:

1. Modularização e responsabilidades
   a. Identifique em qual arquivo reside a lógica de leitura do sensor de temperatura e onde ocorre a conversão para “°C × 100”.

Reposta: A lógica de leitura do sensor de temperatura está implementada na função poll_temp() localizada no arquivo server_common.c. Dentro desta função, após a leitura do ADC e a conversão para graus Celsius na variável deg_c, a conversão final para "°C×100" e o armazenamento na variável global current_temp ocorrem na linha current_temp = (uint16_t)(deg_c * 100);. Essa abordagem centraliza a manipulação do sensor em um local específico, facilitando modificações ou calibrações futuras.

   b. Explique por que a separação entre server_common.c e server.c (ou server_with_wifi.c) é importante para a manutenibilidade do projeto.

Resposta: A separação entre server_common.c e os arquivos server.c ou server_with_wifi.c é crucial para a manutenibilidade. O server_common.c encapsula funcionalidades genéricas do servidor BLE, como o tratamento de pacotes HCI/ATT, callbacks de leitura/escrita GATT e a lógica do sensor de temperatura. Os arquivos server.c e server_with_wifi.c contêm a lógica da aplicação principal, como a configuração de timers específicos ou a integração com funcionalidades adicionais (Wi-Fi e iperf no caso do server_with_wifi.c). Essa divisão permite que a base do servidor BLE seja mantida e testada independentemente das particularidades de cada aplicação, facilitando atualizações e a reutilização do código comum em diferentes projetos.

2. Fluxo de eventos e call-backs

a. Descreva o que acontece, passo a passo, desde que o Stack BTstack emite BTSTACK_EVENT_STATE até o primeiro anúncio (advertising) de BLE.

Resposta: Quando o BTstack está pronto e emite o evento BTSTACK_EVENT_STATE com o estado HCI_STATE_WORKING, a função packet_handler() em server_common.c é acionada. Primeiramente, o endereço local do dispositivo é obtido e impresso. Em seguida, são configurados os parâmetros de anúncio (advertising) através de gap_advertisements_set_params(), definindo o intervalo e tipo de anúncio. Logo após, os dados do anúncio (como nome do dispositivo e UUIDs de serviço) são definidos com gap_advertisements_set_data(). Finalmente, o primeiro anúncio BLE é iniciado com a chamada gap_advertisements_enable(1).

b. Explique como o código garante que as notificações só serão enviadas quando o cliente estiver realmente subscrito (CCCD habilitado) e pronto para recebê-las.

Resposta: O código garante que as notificações só sejam enviadas a um cliente devidamente subscrito através de alguns mecanismos. Primeiramente, no callback att_write_callback() em server_common.c, quando um cliente escreve no Client Characteristic Configuration Descriptor (CCCD) da característica de temperatura, o valor escrito é verificado. Se o valor for GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION, a flag global le_notification_enabled é definida como verdadeira (ou 1). Posteriormente, nas rotinas que disparam as notificações (como no heartbeat_handler ou no packet_handler após ATT_EVENT_CAN_SEND_NOW), esta flag le_notification_enabled é verificada antes de qualquer tentativa de envio com att_server_notify(). O evento ATT_EVENT_CAN_SEND_NOW também assegura que a pilha está pronta para o envio.

3. Temporização sem polling

a. Compare o uso do timer interno do BTstack em server.c com o uso do worker assíncrono em server_with_wifi.c para agendar leituras periódicas de temperatura.

Resposta: Em server.c, um timer da própria BTstack, do tipo btstack_timer_source_t e nomeado heartbeat, é utilizado. Ele é configurado com a função heartbeat_handler e adicionado ao run loop do BTstack usando btstack_run_loop_set_timer() e btstack_run_loop_add_timer(). Já em server_with_wifi.c, que integra funcionalidades de Wi-Fi, a abordagem é diferente: um async_at_time_worker_t chamado heartbeat_worker é definido, também associado à

função heartbeat_handler. Este worker é adicionado ao contexto assíncrono do CYW43 (cyw43_arch_async_context()) através da função async_context_add_at_time_worker_in_ms() para realizar as leituras periódicas de temperatura e outras tarefas do heartbeat.

b. Quais são os benefícios e riscos de cada abordagem em relação a consumo de CPU e responsividade do BLE?

Reposta: A abordagem do timer interno do BTstack (server.c) tende a ter um consumo de CPU relativamente baixo, pois está integrada ao loop de eventos principal da pilha BLE e só é ativado quando necessário, sem polling ativo. No entanto, sua responsividade pode ser afetada se outras operações BLE longas estiverem bloqueando o run loop. A abordagem do worker assíncrono (server_with_wifi.c) é projetada para melhor coexistência com as operações de Wi-Fi gerenciadas pelo async_context do CYW43, podendo oferecer melhor responsividade em ambientes com múltiplas tarefas de rádio. O risco principal desta segunda abordagem é a complexidade adicional no gerenciamento e depuração de tarefas concorrentes no async_context, e um potencial overhead ligeiramente maior devido à camada de abstração do worker.

4. Integração BLE × Wi-Fi

a. Em server_with_wifi.c, quais chamadas iniciam o modo estação Wi-Fi e o servidor iperf?

Resposta: No arquivo server_with_wifi.c, a inicialização do modo estação Wi-Fi (STA mode) é feita pela chamada cyw43_arch_enable_sta_mode(). Após habilitar o modo STA, a conexão a uma rede Wi-Fi específica é tentada com cyw43_arch_wifi_connect_timeout_ms(WIFI_SSID, WIFI_PASSWORD, CYW43_AUTH_WPA2_AES_PSK, 30000). O servidor iperf, por sua vez, é iniciado utilizando a função lwiperf_start_tcp_server_default(&iperf_report, NULL), que configura o servidor para escutar na porta TCP padrão do iperf e utilizar a função iperf_report como callback para os resultados.

b. Discuta potenciais conflitos de recursos (radio, memória, CPU) quando BLE e Wi-Fi rodam simultaneamente no mesmo core do RP2040.

Resposta: A execução simultânea de BLE e Wi-Fi no mesmo core do RP2040, utilizando o mesmo rádio CYW43, pode levar a conflitos de recursos. O rádio em si é um recurso compartilhado que precisa alternar entre os protocolos, o que pode causar contenção e degradação de performance para ambos, especialmente se houver tráfego intenso (interferência na banda de 2.4GHz). A CPU também é compartilhada para processar as duas pilhas de rede (BTstack para BLE e LwIP para Wi-Fi) e as lógicas de

aplicação, podendo levar a gargalos e afetar a latência e responsividade. A memória é outro recurso crítico; ambas as pilhas e as aplicações consomem RAM, e um gerenciamento inadequado pode levar a estouro de memória (btstack_config.h e lwipopts.h definem limites para buffers e pools).

5. Gerenciamento de conexão e power management

a. Como o código trata a desconexão do cliente BLE? Explique quais variáveis de estado são atualizadas e como isso reflete no LED de status.

Resposta: O tratamento da desconexão do cliente BLE ocorre na função packet_handler() em server_common.c. Quando um evento HCI_EVENT_DISCONNECTION_COMPLETE é recebido, a variável global le_notification_enabled é definida como 0 (ou false), indicando que o cliente não está mais subscrito para notificações. Adicionalmente, o handle da conexão con_handle é invalidado, atribuindo-lhe HCI_CON_HANDLE_INVALID. No client.c (que não controla o LED do servidor, mas sim seu próprio LED), a desconexão também leva à invalidação do connection_handle e, se um listener_registered estava ativo, ele é desregistrado e a flag listener_registered é zerada, o que por sua vez afeta o padrão de piscar do LED no heartbeat_handler do cliente, fazendo-o piscar mais lentamente.

b. Sugira, com base na implementação atual, um local adequado para inserir uma chamada de economia de energia (sleep) no rádio CYW43 entre notificações.

Resposta: Com base na implementação atual, um local adequado para inserir uma chamada de economia de energia (sleep) no rádio CYW43 seria dentro da função heartbeat_handler (tanto em server.c quanto em server_with_wifi.c). Especificamente, após a lógica de atualização do LED e antes da linha que reagenda o timer/worker (btstack_run_loop_add_timer(ts); ou async_context_add_at_time_worker_in_ms(context, &heartbeat_worker, HEARTBEAT_PERIOD_MS);). Poderia ser verificado se não há conexões ativas (con_handle == HCI_CON_HANDLE_INVALID) ou se as notificações não estão habilitadas (!le_notification_enabled), e se o dispositivo não está no meio de uma operação crítica. O sleep_ms() usado no loop while(true) do main() já sugere uma forma de economia de energia para o core, mas uma gestão mais fina do rádio entre anúncios ou notificações poderia ser implementada no heartbeat.

6. Cliente BLE

a. No client.c, qual é o critério para filtrar anúncios de BLE antes de conectar?

Resposta: No arquivo client.c, o critério principal para filtrar anúncios BLE antes de tentar uma conexão é a presença de um UUID de serviço específico no pacote de anúncio. Dentro da função hci_event_handler(), ao receber um GAP_EVENT_ADVERTISING_REPORT, o código chama a função advertisement_report_contains_service(ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING, packet). Somente se esta função retornar true, indicando que o dispositivo anunciado expõe o Serviço de Sensoriamento Ambiental (UUID 0x181A), o cliente prosseguirá para parar a varredura e iniciar uma tentativa de conexão com gap_connect().

b. Detalhe as etapas de descoberta de serviço e característica no cliente, e explique como o CCCD é escrito para habilitar notificações.

Resposta: No client.c, após uma conexão ser estabelecida (evento HCI_SUBEVENT_LE_CONNECTION_COMPLETE), a descoberta de serviços é iniciada com gatt_client_discover_primary_services_by_uuid16(), filtrando pelo UUID ORG_BLUETOOTH_SERVICE_ENVIRONMENTAL_SENSING. Quando o resultado da descoberta de serviço chega (GATT_EVENT_SERVICE_QUERY_RESULT), as informações do serviço são armazenadas em server_service. Em seguida, o cliente inicia a descoberta de características para este serviço com gatt_client_discover_characteristics_for_service_by_uuid16(), buscando pela característica de Temperatura (ORG_BLUETOOTH_CHARACTERISTIC_TEMPERATURE). Ao receber GATT_EVENT_CHARACTERISTIC_QUERY_RESULT, a característica é armazenada em server_characteristic. Finalmente, para habilitar as notificações, o cliente escreve o valor GATT_CLIENT_CHARACTERISTICS_CONFIGURATION_NOTIFICATION no Client Characteristic Configuration Descriptor (CCCD) da característica de temperatura usando a função gatt_client_write_client_characteristic_configuration().

- GitHub: https://github.com/ManoelFelipe/Embarcatech_37/tree/main/Unidade_02/Cap_01/Atividade_01

O projeto está dividido em três partes principais:

Este projeto está dividido em três programas principais (ou "alvos de compilação") que podem ser carregados no Pico W, cada um com sua própria função int main():

1) Cliente BLE (client.c):
   Este programa configura o Pico W para atuar como um cliente Bluetooth Low Energy.
2) Servidor BLE Básico (server.c):
   Este programa configura o Pico W para atuar como um servidor Bluetooth Low Energy.
3) Servidor BLE com Wi-Fi (server_with_wifi.c):
   Este programa expande o servidor BLE básico, adicionando funcionalidade Wi-Fi.

   Componentes Comuns e de Configuração:
   - Lógica Comum do Servidor (server_common.c, server_common.h): Contém o código compartilhado entre os dois programas de servidor (server.c e server_with_wifi.c).
   - Arquivo de Cabeçalho do Sensor (temp_sensor.h - implícito): Embora não fornecido diretamente, este arquivo é incluído por server_common.c e é crucial, pois espera-se que defina os handles GATT para a característica de temperatura.
   - Arquivos de Configuração (btstack_config.h, lwipopts.h):
     - btstack_config.h: Configura a pilha Bluetooth (BTstack), definindo recursos como papéis (periférico/central), tamanhos de buffer e funcionalidades do HAL.
     - lwipopts.h: Configura a pilha TCP/IP (LwIP) para a funcionalidade Wi-Fi, ajustando o gerenciamento de memória, protocolos e buffers.

Essencialmente, o projeto oferece três "personalidades" distintas para o Pico W, reutilizando código comum onde aplicável e permitindo que o desenvolvedor escolha qual firmware compilar e carregar.