CANDIDATO: Miguel Lino Ferreira Neto
Link de repositório Github: https://github.com/miguelneto0/embedded_software

Prova Didática: Pesquisador I - Software Embarcado

Questão 1.

i. Arquitetura do software embarcado: apresento os requisitos funcionais e não-funcionais do sistema, bem como detalhes dos componentes e um esquemático de exemplo acompanhado de um protótipo com a ferramenta *wokwi.com* no corpo do texto explicativo.

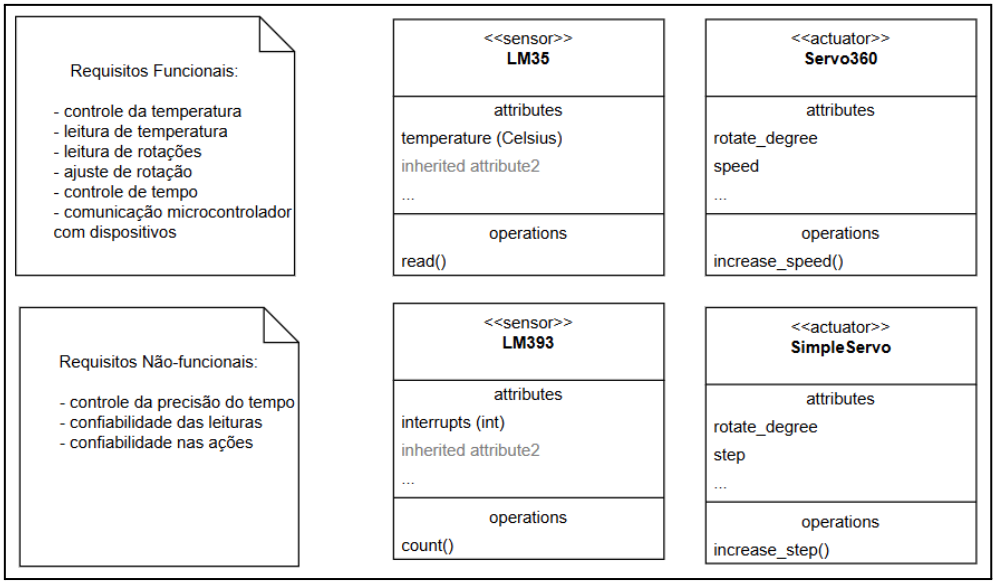

Figura 1: Requisitos do sistema e modelagem dos componentes.

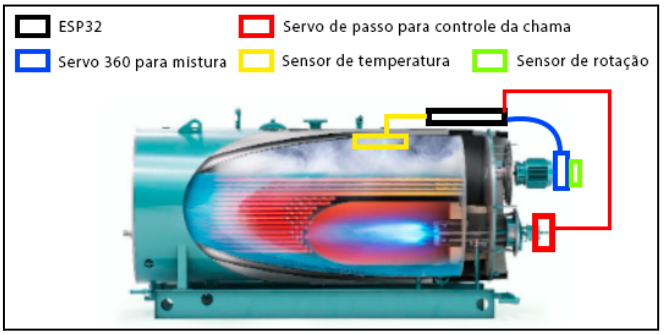

Figura 2: Esquemático exemplificando a distribuição dos componentes no sistema.

ii. Microcontrolador: ESP32 (detalhes no corpo do texto explicativo)
iii. Periféricos: sensores: sensor temperature (ex. LM35 analogico com 3 pinos), sensor de rotação (ex. LM393 digital com 4 a 5 pinos); atuadores: servo motor 360 (digital até 3 pinos) e um servo motor simples (4 pinos) junto com uma base servo de 2 pinos que são conectados ao ESP32.
iv. Pseudo-código + fluxograma: explicação do fluxo do código no corpo do texto explicativo.

```
#define TARGET_TEMP = [50, 65, 80]
#define TARGET_RPM  = [20, 30, 40]
#define TARGET_TIME = [1800, 1800, 3600]

int phase = 0;
int rpm = 0;
double temp = 0;

int activate_rotation(){ return rpm++; }
int activate_rotation(){ return temp++; }
void update_targets(){ phase++; }


function control_process():

    current_temp = 0
    current_rpm  = 0
    current_time = 0

    start_time = 0
    target_reached = False

    while ( !target_reached ) {
        current_rpm  = activate_rotation()
        current_temp = activate_heater()
        if (current_rpm != TARGET_RPM[phase] and current_temp != TARGET_TEMP[phase]){
            start_time = timer.getCurrentTime()
            target_reached = True
        }
    }

    while (current_time != target_time[phase]) {
        current_time = timer.getTick()
        if (current_time == target_time[phase] ) {
            update_targets()
        }
    }

function main():
    if ( phase > TARGET_RPM.size )
        return
    else {
        control_process()
    }
```

Figura 3: Pseudo-codigo do sistema de controle.

Texto explicativo:

De acordo com o problema apresentado, um sistema para realizar o controle da temperatura de uma caldeira e a rotação da mistura via software embarcado, implica na utilização de alguns periféricos que serão usados para leitura (sensores) de dados da temperatura e do componente de rotação, além de alguns componentes para controlar a chama para aquecer a caldeira e o sistema de rotação (que serão os atuadores).

Nesse caso, algumas soluções poderiam ser propostas partindo da escolha de um microcontrolador que suporte tais periféricos. Aqui, foi escolhido o ESP32-S da Espressif por ser um dispositivo bastante usado nesse tipo de projeto, possuindo 32 bits de processamento, além de envolver uma quantidade razoável de entradas e saídas (digitais e analógicas, visto que os sensores necessários para o projeto podem necessitar diferentes tipos). Além disso, tal dispositivo seria importante para uso em larga escala, por operar com sistemas de tempo real como o FreeRTOS, permitindo coletar dados precisos de tempo e uso de 2 núcleos de processamento. Além disso, tal microcontrolador vem de fábrica com módulos wifi e bluetooth para comunicação, se tornando robusto para este tipo de projeto. Embora tais recursos não sejam exigidos para as necessidades mencionadas, o ESP32 é um dispositivo bastante interessante e fácil de usar permitindo programação em linguagem

C e MicroPython. Uma alternativa seria usar um Arduino com a quantidade similar de portas e alguns modulos adicionais, mas por simplicidade a preferência foi por uma placa que exija menos componentes conectados, conforme ilustrado nos esquemáticos das figuras 2 e 4.

Quanto aos periféricos, dois sensores são propostos: um de temperatura que permita certa precisão e outro digital de leitura optica para contagem de interrupção no feichede luz indicando quantas voltas são calculadas a partir do servo motor usado; enquanto para os atuadores são definidos: um servo motor com capacidade de 360 graus para contagem de voltas e outro servo por passo para controlar o acionamento da chama (imaginando que é acionada por um tipo de dispositivo acionado por giro, este seria um servo mais simples sem função de rotação 360 graus). Para o sensor de temperatura, foi escolhido um sensor analógico preciso como o LM35 que pode trabalhar com uma precisão de 0.01 numa faixa de -50 a 150°C, diferente de sensores mais simples e usuais como o DHT 11 e 22. Tal medida seria suficiente considerando os valores indicados em cada etapa da questão. Para o sensor otico, um exemplo como o LM393 pode ser usado para realizar a contagem das voltas (embora na ilustração a seguir criada no wokwi.com seja usado o RY040 por ser a unico encontrado na ferramenta). Quanto aos atuadores, um servo do tipo stepper seria usado para acionar a chama e um servo 360 graus para monitorar a mistura. O modelo deveria acoplar ao ESP32 os 2 sensores (3 pinos analógicos para temperatura e 4 pinos digitais para a rotação) e os 2 atuadores, sendo o servo 360 usando 3 pinos digitais e o outro servo de passo usando 2. A figura 4 ilustra um protótipo das conexões.

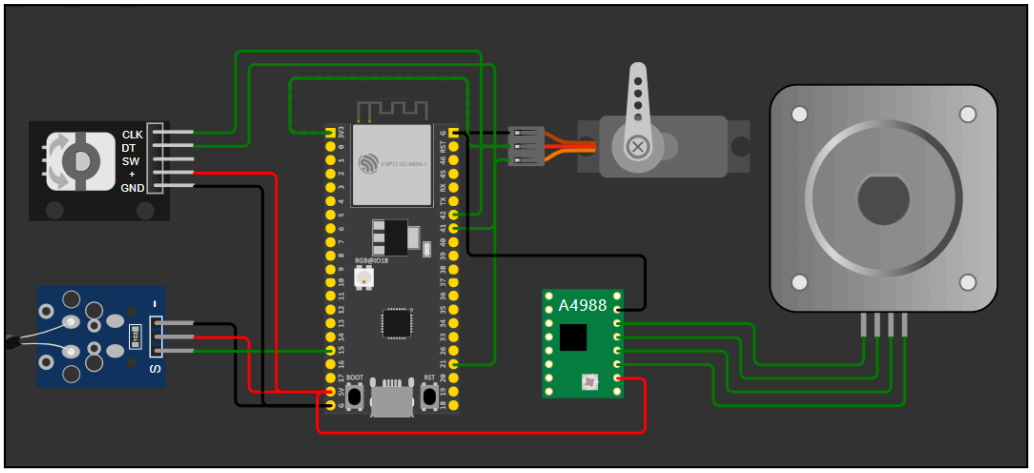

Figura 4: Esquemático das conexões dos componentes com a ESP32 usando wokwi.com.

Quanto ao código e o fluxograma, a estrutura do programa deve iniciar com a definição das variáveis de alvo (target) que vão definir os valores para cada estágio, o estágio 1 teria: *target_temp* = 50 para indicar os 50°C, *target_rpm* = 20 para indicar 20 rpm e *target_time* = 1800s para indicar os 30 minutos em segundos (geralmente os ticks são em milissegundos mas aqui represento como segundos para simplificar). Em seguida são incrementados os valores de temperatura e rotação através do acionamento dos atuadores pela ESP32, para então verificar se as variáveis target são atingidas. Nesse caso, o fluxograma e o pseudocódigo abstraem a ideia de contagem do tempo via software por meio de *ticks* como as funções básicas das bibliotecas de timer (*timer.h*) da ESP32 e do Arduino, mas seria possível também usar temporizadores externos para melhor precisão. Assim, a cada segundo é verificado se os ticks alcançam o tempo definido na variável *target_time* para passar para o próximo estágio ou encerrar. É importante notar que o tempo inicial somente começa a ser contado quando os primeiros valores de temperatura 50 e o rpm 20 forem atingidos. E somente após isso deve iniciar o estágio.

No código, é definido uma lista (ou um array) para guardar cada valor de temperatura, rotação e tempo, sendo cada indice de temperatura e rpm incrementado a cada valor de tempo encontrado. O fluxo dos estágios é definido na função *control_process()* e é chamado no programa principal pela função *main()*. Conforme o estágio avança (incrementa pela função *updates_targets()* os valores são incrementados e obedecidos conforme definido. Para exemplificar, o processo inicia com as variaveis target_temp[0], target_rpm[0] e target_time[0], o que resultaria em 50, 20 e 1800, respectivamente. Em seguida inicia o aquecimento e a rotação, caso *target_temp* e *target_rpm* sejam atingidos (checagem do primeiro *while* no código), o tempo para cada estágio começa a ser contado. Assim, cada estágio é controlado por 2 loops *while* sendo o primeiro para verificar que a temperatura e a rotação seja alcançada e o segundo apra controlar que o tempo em que as variáveis se mantém no estado correto. Neste ponto é importante deixar claro que para tornar o processo mais robusto e preciso, sem varições, o idela seria implementar um controle PID (Proporcional, Integral e Diferencial) para ajustar as variáveis de contrle de forma correta e sem preceptações. Quando o estágio 3 ocorre, as estruturas estão com os dados do indice 2 da lista e na função main() é avliado se phase é maior que o tamanho da estrutura, e ao encerrar o estágio 3, phase 3 será maior que o indice 2, encerrando o programa (ou seja, parando os atuadores e desligando a placa).

No fluxograma, são usados retangulos para ilustrar estados do sistema, sendo o estado START e END usados somente para fins de orientação indicando inicio e fim do processo. As declaracoes de variaveis podem ser consideradas no quadro START. Cada losango indica uma condição que vai ter dois caminhos (*Yes* ou *No*) para indicar se a condição foi atendida. Assim, para ilustrar os 3 estágios nós teriamos: 1. inicia as variaveis com 50, 20, 1800 respectivamente. Inicia contagem do tempo (em ticks), verifica a cada segundo se 1800 foi atingido, se for, passa par ao estado de atualização de targets (caixinha *update targets*) que vai incrementar os indices, target_temp[1], target_rpm[1] e target_time[1], que serão respectivamente 65, 30 e 1800, zerando o timer. Na próxima iteração, similarmente, teriamos o indice 2 para cada estrutura que resultaria em 80, 40 e 3600. Ao final do terceiro estágio, como o fim das estruturas é encontrado o tamanho das três variaveis foi atingido, então isso quer dizer que os alvos devem ser zerados e encerrar (estado do losango *target cleared*), e o programa encerra.

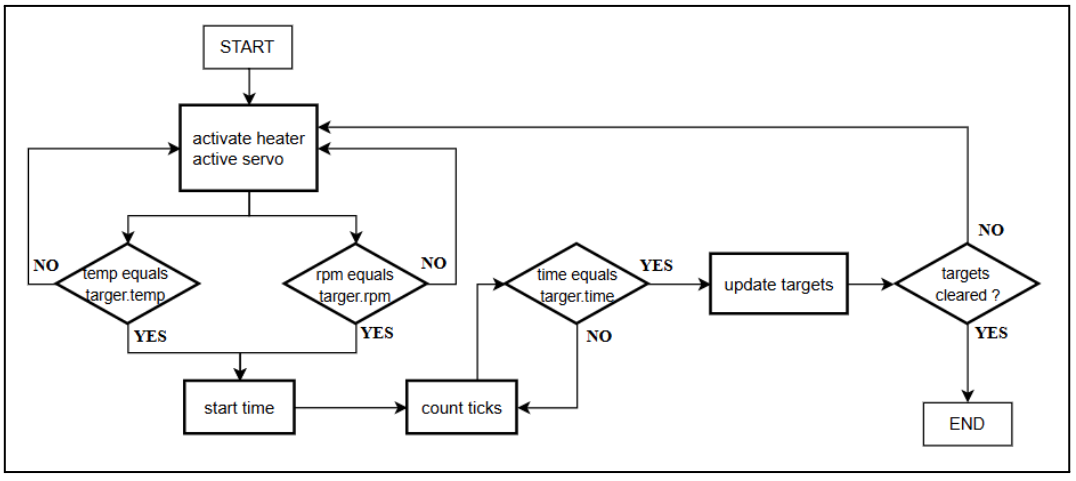

Figura 5: Fluxo-grama do sistema de controle.

Questão 2.

O ROS (Robot Operating System) é um sistema opercional amplamente utilizado em sistemas robóticos embarcados por sua flexibilidade e infraestrutura robusta (mesmo a versão Humble sendo desenvolvida para rodar a partir do Ubuntu 22). Um dos motivos de sua relevância é o suporte a comunicação entre componentes de um sistema robótico, que facilita o desenvolvimento de aplicações distribuídas, integrações com sensores, e controle de atuadores em tempo real.

Conforme solicitado no enunciado a questão, o ROS deve ser criado como imagem Docker para atender às condições de: monitoramento do uso de memória, definido no pacote 1, a geração e coleta de dados de sensores com base na média e na criação de interfaces descritos no pacote 2, além de uma função de busca de dados como no pacote 3.

Assim, o Dockerfile foi criado na pasta refetente a questao2 no repositório github e os pacotes separados em cada pasta com os respectivos arquivos .xml, .py, .cfg dentro de workspace que correspondem respectivamente a definição do pacote, o setup, e a configuração de cada pacote. É usada a ferramenta *colcon* para construção dos pacotes, o *curl* para de linha de comando e o *python3* como interpretador. No corpo do Dockerfile o comando RUN apt-get destaca cada componente e suas respectivas dependencias.

Questão 3.

Neste problema é explorado o uso da criptografia via algoritmo AES 128bits, usando bibliotecas de criptografia do python a solução se torna mais simples. O ponto a se atentar é que o AES trabalha com *padding* e *unpadding* de dados (remoção e adição de simbolos para preenchimento das partes em que o algoritmo vai atuar tendo que limitar a 16bytes por bloco devido o modo ECB - *eletronic codebook* - descrito na questão). Uma função de *unpadding* é usada para remover dados não compatíveis com a codificação UTF-8, referente ao modo ECB que, pesquisando pela internet, descobri ter referência com um tipo de remoção chamada de PKCS7, facilitando a decriptografia e a garantia dos testes.

Na pasta referente à questão 3 no repositório github encontra-se o arquivo q3_algoritmoAES.py que implmenta o algoritmo e decifra a setença, e o codigo referente aos testes unitários q3_unit_test.py que compõe 4 cenários possíveis, como segue:

i. Código python para decifrar a mensagem usa a biblioteca *Cripto.Cipher* é possível resolver, gerando a mensagem: “*Sistemas Embarcados*”.

```
Miguel@DESKTOP-DDGT64I MINGW64 ~/Documents/ufsc 2024.2/Seletivo 2024/embedded_software/questao3 (main)
$ python q3_algoritmoAES.py
Mensagem decifrada: Sistemas Embarcados
```

Figura 6: Executando código python do algoritmo AES e imprimindo mensagem decifrada.

ii. Codigo de testes unitários usando a biblioteca *unittest* do python pela opção TestCase, sendo: o primeiro teste para o cenário em que a mensagem é decifrada corretamente com a chave correta; o segundo testa o cenário quando a chave de criptografia é incorreta e deveria gerar uma resposta diferente (usando *assertNotEqual* para garantir isso); o terceiro teste verifica o cenário quando a mensagem tem tamanho insuficiente devendo gerar uma *exception ValueError*; e, o quarto, também deve gerar tal *exception* mas para o caso em que a chave é curta.

```
Miguel@DESKTOP-DDGT64I MINGW64 ~/Documents/ufsc 2024.2/Seletivo 2024/embedded_software/questao3 (main)
$ python -m unittest q3_unit_test.py
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
Mensagem decifrada: Sistemas Embarcados
```

Figura 7: Executando testes unitários e garantindo que todos passam para os 4 cenários explorados.