UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

Documentação de Projeto – Parte 2
Design, Estudo da Plataforma

Projeto: Controlador de simulação de 3 elevadores

Autores: Mariana Bittencourt e Henrique Mazzuchetti

Versão: 27-Nov-2022

Parte 2a – Design

1 Introdução

O objetivo deste documento é detalhar mais o que foi apresentado no primeiro documento (CONOPS), ou seja, detalhar mais da solução que será implementada no projeto, trazendo diagramas como o de arquitetura física, que contém todos os elementos que irão compor a solução e a relação entre eles.

2 Arquitetura Funcional

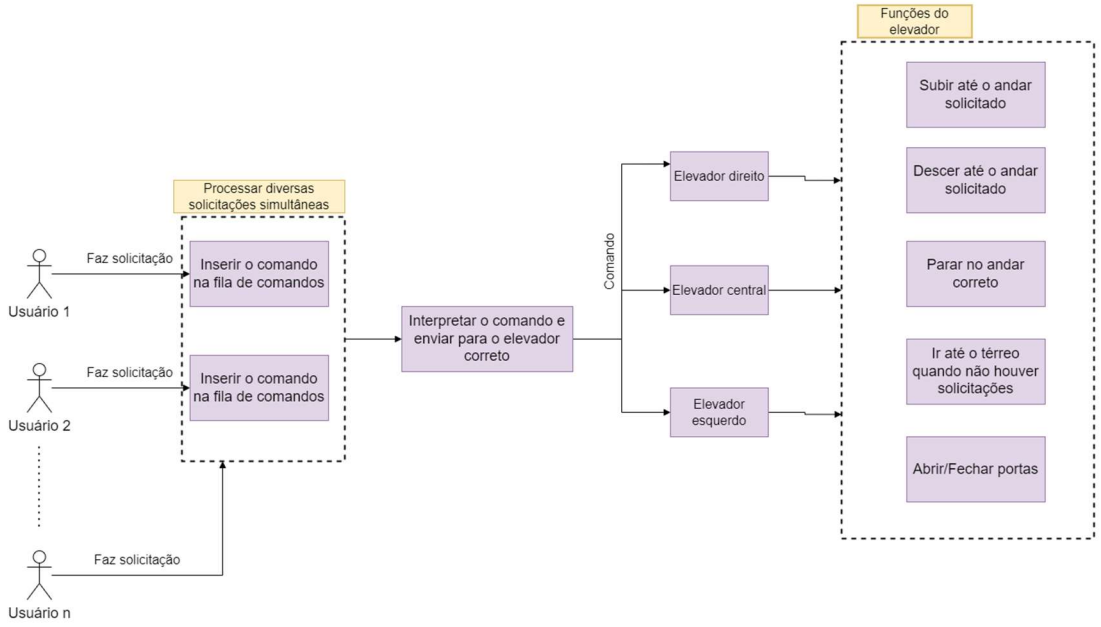


Figura 1 – Diagrama functional

3 Arquitetura Física

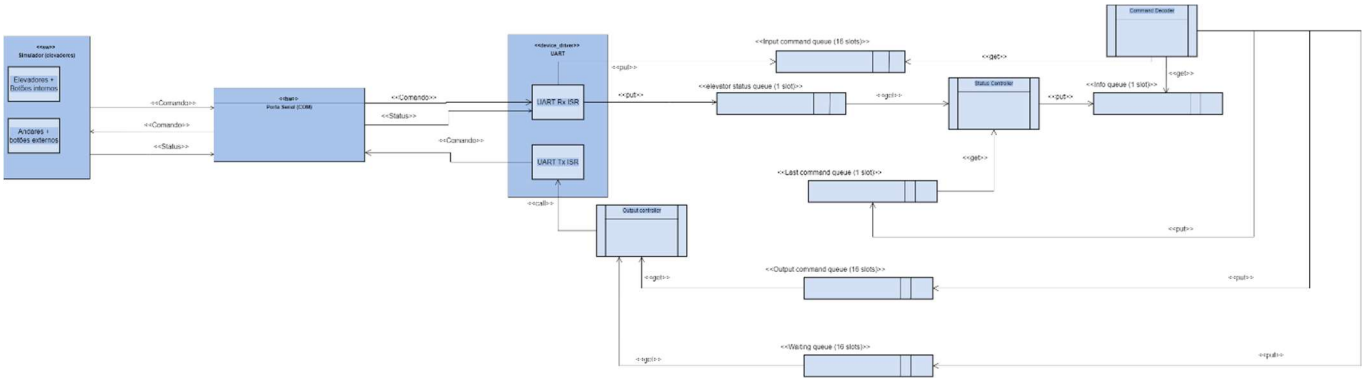


Figura 2 – Diagrama de arquitetura física

4 Interface com o Usuário

Vide Documento 1.

5 Mapeamento da Arquitetura Funcional à Arquitetura Física

	Inserir o comando na fila de comandos	Interpretar o comando e enviar para o elevador correto	Subir até o andar solicitado	Descer até o andar solicitado	Parar no andar correto	Ir até o térreo quando não houver solicitações	Abrir/Fechar Portas	Processar diversas solicitações simultâneas
Simulador (elevadores)			X	X	X	X	X	
Porta Serial (COM)	X							
Device driver UART	X							
Status listener						X	X	
Command decoder		X						
Right Elevator								X
Center Elevator								X
Left Elevator								X

Figura 3 – Mapeamento de arquitetura física à arquitetura funcional

6 Arquitetura do Hardware

A solução de HW que será utilizada nesse projeto é o kit de desenvolvimento da Texas Instruments, Tiva C Series EK-TM4C1294XL.

A Tiva é uma plataforma de baixo-custo, que possui o microcontrolador TM4C1294NCPDT, além de botões, leds e suporte para outros periféricos.

Particularmente nesse projeto, será utilizada a porta COM associada à porta USB de debug da placa.

O simulador de elevadores também utiliza a mesma porta COM para se comunicar com a placa, portanto um software para "dividir" a porta COM em duas será necessário.

O software desenvolvido para a placa também irá contar com o uso de um RTOS, suportado pela TIVA.

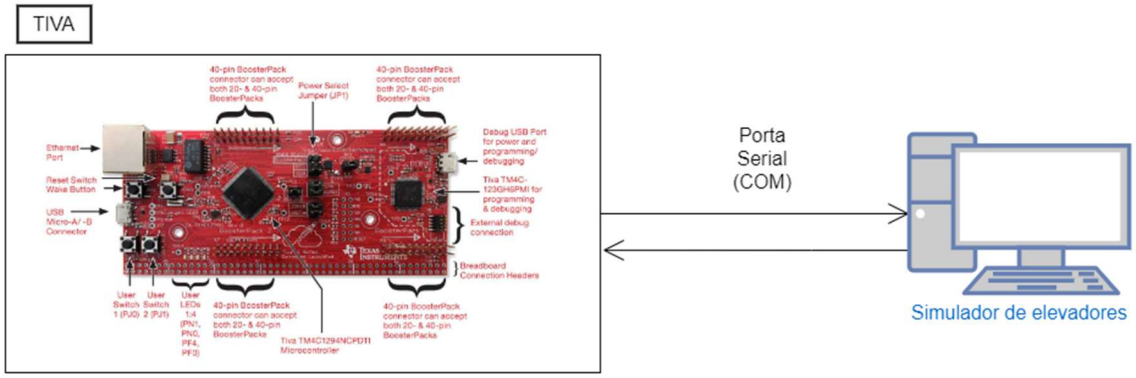
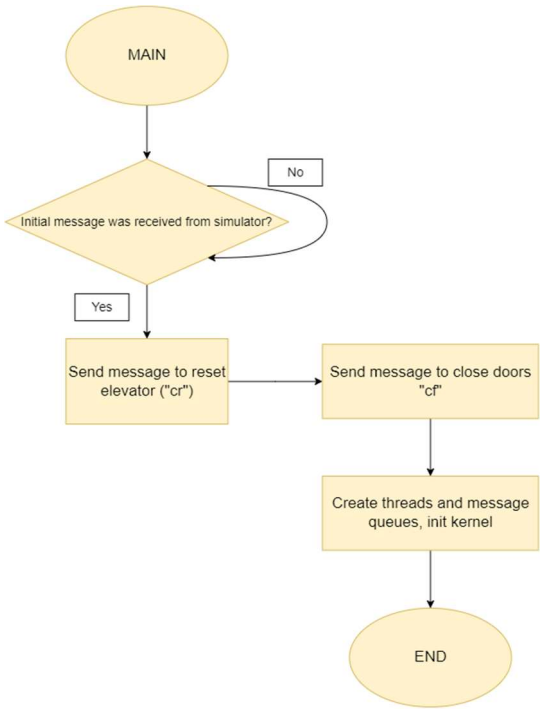


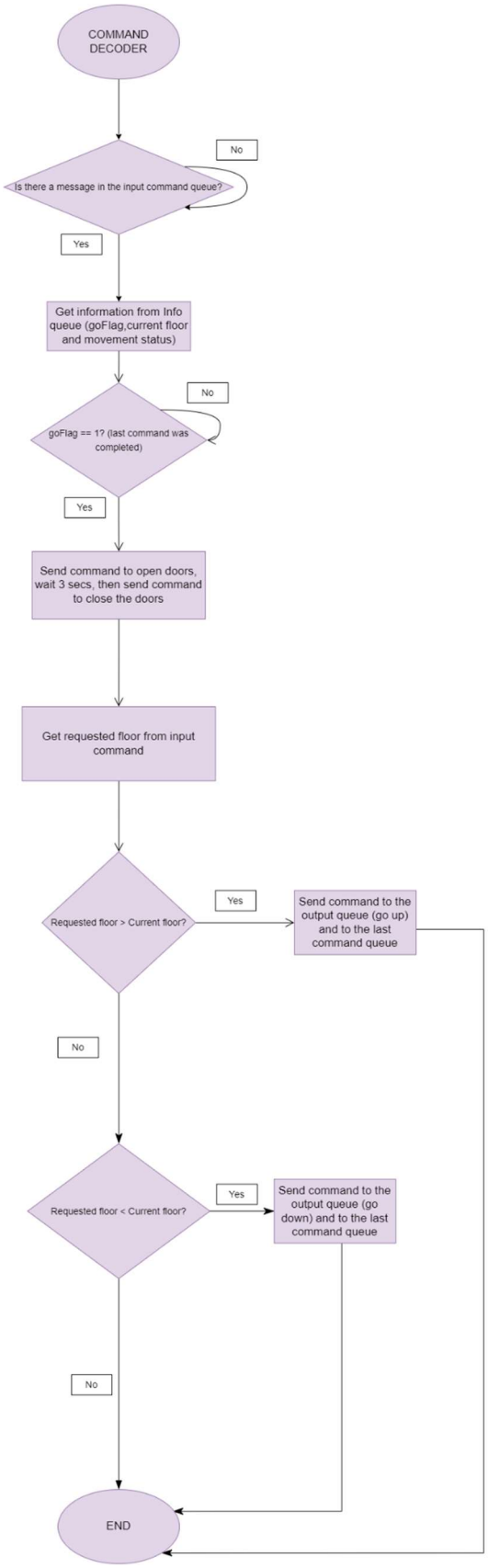
Figura 4 – Arquitetura do Hardware

7 Extras - Fluxogramas

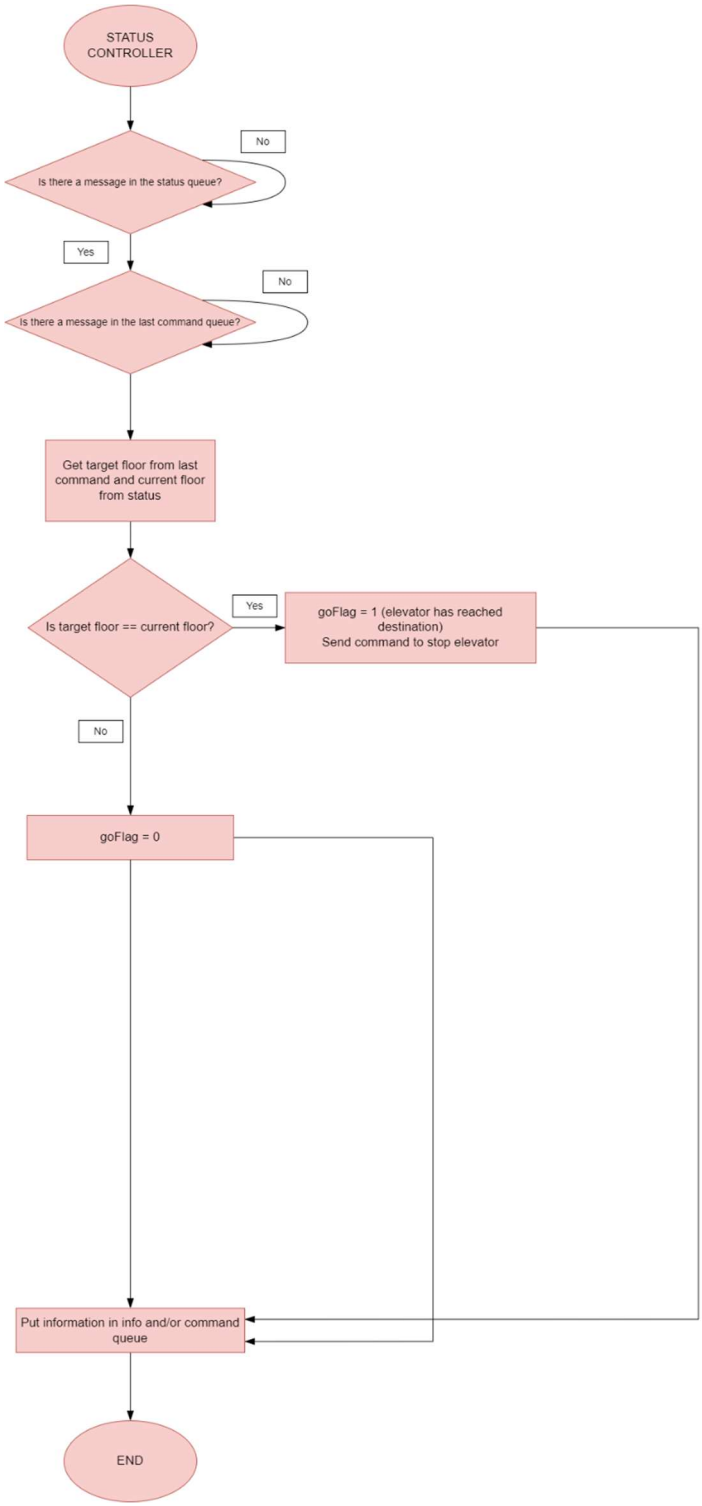
Fluxograma da função "main":



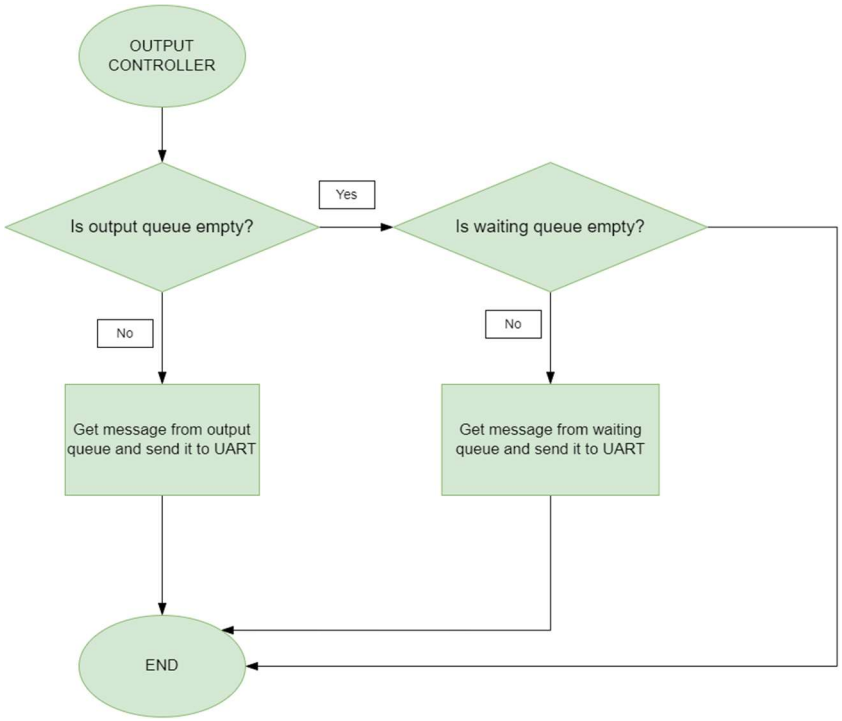
Fluxograma da thread CommandDecoder:



Fluxograma da thread StatusController:



Fluxograma da thread OutputController:



Parte 2b – Estudo da Plataforma

1 Sistema Operacional de Tempo Real (CMSIS-RTOS)

Para o desenvolvimento do projeto, será utilizado um RTOS (Sistema Operacional de Tempo Real) denominado CMSIS-RTOS.

O CMSIS-RTOS é um RTOS que segue a especificação de interface padrão para microcontroladores Cortex-M da ARM (CMSIS).

Esse RTOS possui diversos elementos, e serão listados neste documento aqueles elementos relevantes para o projeto e que farão parte da solução final.

1.1) Threads

São a unidade básica de execução do RTOS. São similares a funções em C, mas possuem algumas diferenças fundamentais.

Por exemplo, nas funções em C sempre há um retorno para a função principal. Já as threads iniciam sua execução e nunca retornam.

Um programa de RTOS é feito de várias threads, que são controladas pelo escalonador do RTOS. O escalonador usa o SysTick para gerar uma interrupção periódica como base de tempo, pois assim ele pode delegar um certo período para cada thread ser executada. Esse período é curto e imperceptível aos usuários, que têm a sensação de que as threads rodam ao mesmo tempo.

Quando o RTOS substitui a execução de uma thread por outra, o contexto da thread anterior deve ser salvo e o da thread atual ser recuperado da memória.

Todas as threads possuem prioridade. Se uma thread de maior prioridade fica pronta para executar, o escalonador do RTOS irá parar a thread atual e iniciar a execução da thread de maior prioridade.

No CMSIS-RTOS, para criar threads, deve-se primeiro inicializar o Kernel do OS, chamando a função:

osKernelInitialize();

Depois disso, pode-se criar as primeiras threads, utilizando as seguintes funções (exemplo):

```
osThreadId thread1_id;

void thread1 (void const *argument);

osThreadDef(thread1, osPriorityNormal, 1, 0);
```

E então:

Thread1_id = osThreadCreate(osThread(thread1), NULL);

Essa última chamada cria a thread e inicia sua execução.

Podem ser criadas uma ou mais threads, tanto no início quanto durante a execução.

Depois disso a seguinte função, que inicia o Kernel, pode ser chamada:

osKernelStart();

1.2) Comunicação inter-threads

As threads rodam "ao mesmo tempo", então alguma maneira de comunicação entre elas deve existir para garantir a organização.

Existem diversos elementos de comunicação entre threads, entre eles: Semáforos, mutexes e filas de mensagens.

Semáforos

É um container que possui um número de tokens. Caso uma thread queira executar um certo trecho de código protegido, irá tentar adquirir um token do semáforo. Caso o semáforo possua esse token, a thread irá continuar a executar e o número de tokens irá ser decrementado. Se não existirem tokens no semáforo quando a thread tenta recuperar um token, essa thread irá ser colocada em um estado de espera, até que o token esteja disponível.

Para usar um semáforo no CMSIS-RTOS, deve-se declarar um container de semáforo com as funções (Exemplo):

osSemaphoreId sem1;

osSemaphoreDef(sem1);

Em seguida, inicializar o semáforo com um certo número de tokens:

Sem1 = osSemaphoreCreate(osSemaphore(sem1),SIX_TOKENS);

Após essa inicialização, uma thread pode ser bloqueada até que um token esteja disponível, com a função:

osStatus osSemaphoreWait(osSemaphoreId semaphore_id, uint32_t milisec);

E, caso esteja utilizando o recurso do semáforo, uma thread pode liberar o recurso enviando um token para o semáforo utilizando a função:

osStatus osSemaphoreRelease (osSemaphoreId semaphore_id);

Mutexes

Mutex é uma abreviação para "Mutual Exclusion". Na verdade, o Mutex é apenas uma versão de um semáforo, descrito acima. A diferença é que o mutex pode conter apenas um token que não pode ser criado nem destruído. O princípio do mutex é proteger e controlar o acesso a um recurso como um periférico. Fora isso, funciona da mesma forma que um semáforo, porém as funções para manipular um mutex são diferentes.

Antes de tudo, é preciso declarar o mutex e inicializá-lo (Exemplo):

osMutexId uart_mutex;

osMutexDef(uart_mutex);

Depois, o mutex deve ser criado dentro de uma thread:

Uart_mutex = osMutexCreate(osMutex(uart_mutex));

Então, as próximas threads que quiserem acessar o recurso devem adquirir o token do mutex primeiro:

osMutexWait(osMutexId mutex_id, uint32_t millisec);

Após o uso do recurso, o mutex deve ser liberado:

osMutexRelease(osMutexId mutex_id);

Fila de mensagens

Para transferir dados entre threads, o CMSIS-RTOS oferece a opção das filas de mensagens.

Para configurar uma fila de mensagens, primeiro é preciso alocar os recursos de memória (Exemplo):

osMesssageQId Q_LED;

osMessageQDef(Q_LED,16_Message_Slots, unsigned int);

A função acima define uma fila de mensagens com 16 elementos inteiros sem sinal.

Em uma thread, é possível então declarar a fila de mensagens:

Q_LED = osMessageCreate(osMessageQ(Q_LED), NULL);

Assim, é possível colocar dados na fila, dentro de uma thread:

osMessagePut(Q_LED,0x0,osWaitForever);

E então ler essa mensagem em outra thread:

Result = osMessageGet(Q_LED,osWaitForever);

LED_data = result.value.v;

2 UART Driver

Para esse projeto, a interface entre o simulador e o software embarcado será a porta serial UART da placa TIVA. Por isso, um driver de UART será desenvolvido para facilitar a configuração e o envio e recebimento de mensagens pela porta serial.

Para o desenvolvimento do driver, será utilizada a biblioteca TivaWare, que já possui funções elaboradas para a gestão da porta serial UART. As principais funções são:

```
void
UARTConfigSetExpClk(uint32_t ui32Base,
                    uint32_t ui32UARTClk,
                    uint32_t ui32Baud,
                    uint32_t ui32Config)
```

- Função de configuração da UART para operação. Configura a fonte de clock, o baud rate, o número de bits, o número de stop bits e a paridade.

```
void
UARTIntEnable(uint32_t ui32Base,
              uint32_t ui32IntFlags)
```

- Função que habilita a interrupção de uma determinada UART, e determina o tipo de interrupção.

```
uint32_t
UARTIntStatus(uint32_t ui32Base,
              bool bMasked)
```

- Função que retorna o status de interrupção para uma UART específica. Permite saber qual o tipo de interrupção que ocorreu.

```
void
UARTIntClear(uint32_t ui32Base,
             uint32_t ui32IntFlags)
```

- Função que limpa o flag de interrupção para uma determinada UART e um determinado tipo de interrupção (por exemplo, Rx ou Tx).

```
int32_t
UARTCharGet(uint32_t ui32Base)
```

- Função que captura um caractere da fila de recepção na UART especificada. Aguarda (bloqueia) até que um caractere esteja disponível.

```
void
UARTCharPut(uint32_t ui32Base,
            unsigned char ucData)
```

Função que coloca um caractere na fila de transmissão da UART especificada.