UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
COELE — COORDENAÇÃO DE ENGENHARIA ELETRÔNICA

DJONES ALDIVO BONI

PONTE DE INTERFACES:

Integração de Barramento CAN com Interfaces RS-232 e RS-485

TRABALHO DE CONCLUSÃO DE CURSO

TOLEDO
2016

DJONES ALDIVO BONI

PONTE DE INTERFACES:

Integração de Barramento CAN com Interfaces RS-232 e RS-485

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de Trabalho de Conclusão de Curso 2 (TCC2), do Curso de Engenharia Eletrônica da Coordenação de Engenharia Eletrônica — COELE — da Universidade Tecnológica Federal do Paraná — UTFPR — Câmpus Toledo, como requisito parcial para obtenção do título de Bacharel em Engenharia Eletrônica.

Orientador: Prof. MSc. Claudinei de Jesus Donato

TOLEDO
2016

Ministério da Educação
Universidade Tecnológica Federal do Paraná
Câmpus Toledo
Coordenação do Curso de Engenharia Eletrônica

TERMO DE APROVAÇÃO

Título do Trabalho de Conclusão de Curso Nº 033

Ponte de Interfaces: Integração de Barramento CAN com Interfaces RS-232 e RS-485

por

Djones Aldivo Boni

Esse Trabalho de Conclusão de Curso foi apresentado às 13h do dia **22 de junho de 2016** como requisito parcial para a obtenção do título de **Bacharel em Engenharia Eletrônica**. Após deliberação da Banca Examinadora, composta pelos professores abaixo assinados, o trabalho foi considerado **APROVADO**.

_____
Prof. Dr. Fabio R. Coutinho
(UTFPR-TD)

_____
Prof. Me. Daniel Cavalcanti Jeronymo
(UTFPR-TD)

_____
Prof. Me. Claudinei de Jesus Donato
(UTFPR-TD)
Orientador

Visto da Coordenação

_____
Prof. M. Jorge A. V. Alves
Coordenador da COELE

A Folha de Aprovação assinada encontra-se na Coordenação do Curso de Engenharia Eletrônica

RESUMO

BONI, Djones A. **PONTE DE INTERFACES: Integração de Barramento CAN com Interfaces RS-232 e RS-485.** 2016. 71 p. Trabalho de Conclusão de Curso (Bacharel em Engenharia Eletrônica) – Universidade Tecnológica Federal do Paraná. Toledo, 2016.

A intercomunicação entre diferentes interfaces é importante em vários aspectos na área de sistemas embarcados distribuídos, possibilitando isolação dos canais de comunicação e fácil realização de testes no sistema. Uma ponte de interfaces deve apresentar comportamento determinístico para que possa ser utilizada na integração de sistemas de tempo real, de forma que não afete o determinismo do sistema completo, uma característica essencial para estes sistemas. Este trabalho apresenta um estudo sobre a implementação determinística de pontes de interfaces, analisando a integração de um barramento CAN com as interfaces RS-232 e RS-485. Foi identificada a relação entre o limite de mensagens que podem ser convertidas com sucesso com as taxas de transmissão das diferentes interfaces e a capacidade de armazenamento das filas de conversão. Para validar a implementação da ponte foram realizados testes de conversão, cujos resultados mostraram-se de acordo com os valores esperados.

Palavras-chave: Ponte de Interfaces. Integração de Sistemas Embarcados. Sistema Operacional de Tempo Real. CAN. Controller Area Network. RS-485. EIA/TIA-485. RS-232. EIA/TIA-232-F.

ABSTRACT

BONI, Djones A. **INTERFACES BRIDGE: Integrating a CAN Bus With RS-232 and RS-485 Interfaces.** 2016. 71 p. Completion of Course Work (Bachelor of Electronics Engineering) – Federal University of Technology - Paraná. Toledo, 2016.

The intercommunication among different interfaces is important in several aspects in the distributed embedded systems field, providing isolation of the communication channels and easy system testing. An interface bridge must present deterministic behavior to be used in the integration of real-time systems, so that will not affect the whole system determinism, a essential characteristic for these systems. This work presents a study on the deterministic implementation of interface bridges, analyzing the integration of a CAN bus with RS-232 and RS-485 interfaces. It was identified the relation among the limit of successfully converted messages with the transmission rates of the interfaces and the storage capacity of the conversion queue. Tests were performed to evaluate the interface bridge implementation, whose results were consistent with the expected values.

Keywords: Interface Bridge. Embedded Systems Integration. Real-Time Operating System. CAN. Controller Area Network. RS-485. EIA/TIA-485. RS-232. EIA/TIA-232-F.

LISTA DE FIGURAS

LISTA DE ALGORITMOS

LISTA DE ACRÔNIMOS

ACK	Acknowledgement
CAN	Controller Area Network
CANH	CAN High
CANL	CAN Low
CRC	Código de Redundância Cíclica
DLC	Data Length Code
EIA	Electronic Industries Alliance
FIFO	First In First Out
FILO	First In Last Out
IDE	Identifier Extension
IHM	Interface Homem Máquina
LLC	Logical Link Control
MAC	Medium Access Control
OSI	Open Systems Interconnection
RAM	Random Access Memory
RDA	Receive Data Available
RI	Receive Interrupt
RTC	Real Time Clock
RTOS	Real Time Operating System
RTR	Remote Transmission Request
RX	Receive
SD	Secure Digital
SMP	Sistema Monitor de Peso
SRR	Substitute Remote Request
THRE	Transmit Holding Register Empty
TI	Transmit Interrupt
TIA	Telecommunications Industry Association
TX	Transmit
UART	Universal Asynchronous Receiver/Transmitter
USB	Universal Serial Bus

SUMÁRIO

1 INTRODUÇÃO

Há pouco tempo os sistemas embarcados de controle eram formados por apenas um dispositivo centralizado e autocontido comunicando-se com seus periféricos, sensores e atuadores. Hoje ocorre a mudança onde os sistemas passam a ser distribuídos, havendo diversos dispositivos autônomos realizando suas tarefas de forma dedicada e comunicando-se uns com os outros para compartilhar suas informações.

Essa nova estrutura vem sendo adotada por diversos motivos, sendo os principais a eliminação de pontos únicos de falha e a possibilidade de inserir redundâncias em praticamente qualquer subsistema. Isso melhora a confiabilidade e estabilidade do sistema como um todo e permite que seja implementado de forma que tolere falhas de dispositivos.

Também percebemos a tendência na diversificação de interfaces e redes disponíveis para os sistemas embarcados, permitindo que se comuniquem com mais dispositivos, inclusive trocando informações o mundo externo, por exemplo na internet. Como exemplo temos as interfaces *ethernet* e *wireless* cada vez mais presentes nos sistemas embarcados.

No entanto, devido às limitações que os sistemas embarcados sofrem no tamanho físico, memória disponível e custos de componentes, não é viável implementar diversos periféricos e interfaces em cada um dos dispositivos presentes do sistema. A instalação da infraestrutura, como criar pontos de acesso adicionais para *wireless* e *ethernet*, também é caro e logo se torna difícil de gerenciar com tantos dispositivos.

Nesse cenário, para reduzir a complexidade do sistema, é preciso centralizar o ponto de entrada e saída de informações, permitindo que as várias interfaces que compõem o sistema possam comunicar-se entre elas e com o exterior de uma forma unificada.

Assim temos a necessidade de uma ponte de interfaces, um módulo que une logicamente um conjunto de interfaces de comunicação com diferentes padrões e permite a comunicação entre diversos dispositivos antes isolados por possuírem sistemas de comunicação incompatíveis.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é analisar os requisitos que uma ponte de interfaces deve atender para que seja adequada para interligar sistemas embarcados de controle, tendo como objetivos específicos:

- Implementar a conversão de protocolos;
- Implementar a recepção e transmissão de mensagens nas interfaces;
- Analisar os requisitos para garantir o determinismo da ponte;

- Identificar as limitações físicas;
- Implementar uma ponte de três interfaces (CAN, RS-232 e RS-485).

1.2 ORGANIZAÇÃO DO TEXTO

No capítulo 2 são discutidos conceitos básicos sobre determinismo, sistemas de tempo real e sistemas operacionais de tempo real, além de alguns recursos importantes para a implementação da ponte de interfaces.

As diferenças básicas entre as interfaces utilizadas são revisadas no capítulo 3. As diferenças entre as camadas definidas pelas normas das interfaces são analisadas e definimos então um protocolo para integração das interfaces.

São discutidas as formas básicas de controle de *hardware* no capítulo 5, seguidas pela implementação da transmissão e recepção de dados nas interfaces, utilizando as ferramentas fornecidas pelo sistema operacional.

A integração entre a recepção, conversão de protocolo e transmissão das mensagens, realizada pelas tarefas de recepção e transmissão, são explicadas no capítulo 6.

O capítulo 4 é analisada a relação entre recepção de mensagens em uma interface e seu envio em outra, que compõem uma relação produtor-consumidor. É realizada uma análise quantitativa dos limites entre produção e consumo de mensagens entre as interfaces.

Os resultados obtidos com a implementação da ponte de interfaces são discutidos no capítulo 7, mostrando tempos de atraso entre recepção e transmissão nos sinais e um teste de transmissão de rajadas de mensagens.

1.3 *HARDWARE* UTILIZADO

O *hardware* utilizado para implementação, mostrado na figura 1.1, foi idealizado e projetado na empresa Inove Tecnologia como um módulo centralizador de dados para o sistema de pesagem veicular SMP (Sistema Monitor de Peso). Entre as funcionalidades, o módulo é capaz de comunicar simultaneamente em dois barramentos CAN (*Controller Area Network*), um barramento RS-485 e um barramento RS-232, além de possuir RTC (*Real Time Clock*), memória *flash* externa, cartão SD (*Secure Digital Card*) e USB (*Universal Serial Bus*).

O microcontrolador utilizado é LPC2366 do fabricante NXP Semiconductors. Este microcontrolador possui um núcleo ARM7-TDMI, com 256 kB de memória de programa (*flash*), 56 kB de memória de dados (RAM – *Random Access Memory*). Entre os periféricos constam dois controladores CAN, quatro UART, RTC e USB.

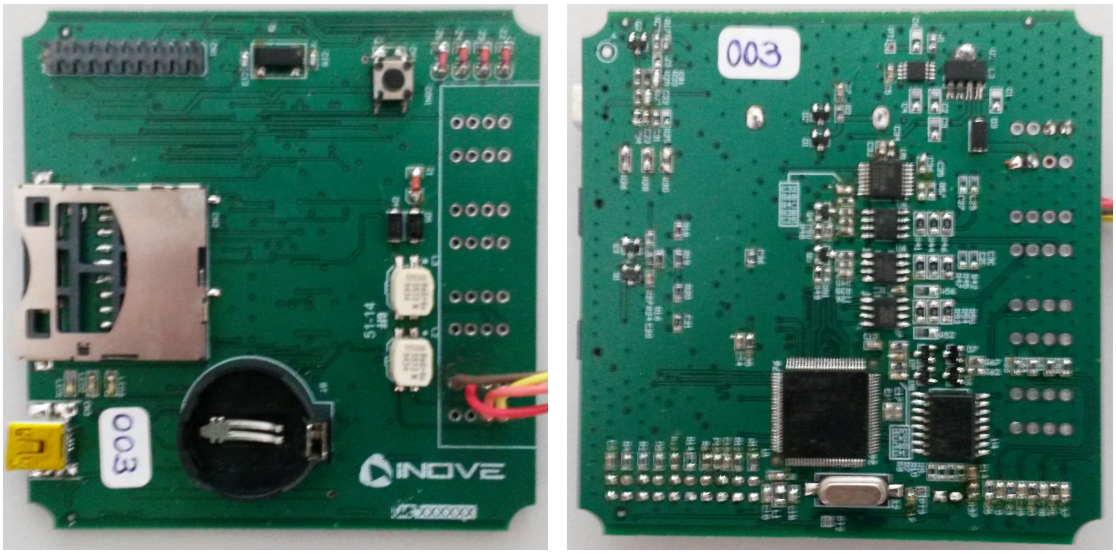

Figura 1.1: Placa utilizada para implementação da ponte de interfaces.
Fonte: Autoria própria.

2 SISTEMAS DE TEMPO REAL E DETERMINISMO

2.1 DETERMINISMO

Determinismo é a propriedade de um sistema de gerar resultados previsíveis, tanto no domínio do valor quanto no domínio do tempo, dado o estado inicial e os estímulos que agem sobre o sistema (KOPETZ, 2011). Dessa forma, o determinismo de um sistema não é definido apenas pelo seu resultado lógico ou o valor de sua saída, mas também pelo tempo que o sistema toma para responder aos estímulos. Se um ou ambos (resultado e tempo) não forem previsíveis, diz-se que o sistema é não-determinístico. Quando resultado e tempo são previsíveis, dadas as condições iniciais e os estímulos, diz-se que o sistema é determinístico.

Um sistema computacional de controle necessita sempre dessa previsibilidade, o que permite determinar se é capaz de controlar um processo, dentro das condições estabelecidas. Por isso o determinismo de um sistema é essencial para que este seja utilizado em qualquer método de controle.

Para exemplificar, é possível citar o caso de uma planta industrial que é controlada por uma malha de realimentação, para que sejam cumpridos os pontos de operação e outras necessidades do processo, como o controle da temperatura de uma reação química. Esta malha de realimentação precisa processar os dados dos sensores e decidir sobre que ações tomar, para que o processo seja mantido em controle. Uma falha na amplitude de saída da malha de realimentação pode acarretar em um desvio do ponto ideal de operação. Incertezas no tempo de processamento do estímulo de um sensor afeta diretamente o sinal de realimentação, podendo causar problemas.

A previsibilidade no domínio do valor e no domínio do tempo, portanto, são importantes para o processo de controle de uma planta industrial, caracterizando a necessidade de um sistema de controle, ou malha de realimentação, com característica determinística.

2.2 SISTEMA DE TEMPO REAL

"Um Sistema de Tempo Real é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos" (FARINES et al., 2000). De acordo com (KOPETZ, 2011), são considerados sistemas de tempo real os sistemas cujo funcionamento correto depende tanto das saídas geradas (resultados lógicos) quanto do tempo em que essas saídas demoram para serem produzidas.

Assim, um sistema de tempo real é um sistema que requer comportamento determinístico nos valores gerados e no tempo de resposta para que seja considerado funcional.

Há basicamente dois tipos de sistemas de tempo real: não-críticos e críticos. Em sistemas não-críticos, também chamado de *soft real time*, o estouro do prazo caracteriza uma falha do sistema, causando perda de eficiência no processo. Já em sistema críticos, ou *hard real time*, o estouro do prazo pode causar grandes prejuízos ou danos irreversíveis (TANENBAUM, 2009) (KOPETZ, 2011).

Como exemplos, podemos tomar o velocímetro de um veículo e o sistema de frenagem. O sistema que atualiza velocímetro do painel de um veículo pode ser considerado um sistema não-crítico, pois a falha ao atualizar a velocidade pode acarretar no recebimento de uma multa de velocidade. Já o sistema de frenagem é um sistema crítico, pois uma falha nele afeta diretamente a segurança do motorista, podendo causar acidentes, ferimentos graves ou até fatais.

2.3 SISTEMA OPERACIONAL DE TEMPO REAL

O sistema operacional utilizado em um sistema de tempo real precisa das características determinísticas citadas anteriormente, ou seja, em um sistema de tempo real, seja ele crítico ou não-crítico, deve-se utilizar um sistema operacional de tempo real (RTOS – *Real Time Operating System*). Um RTOS pode executar várias tarefas, sendo algumas críticas, outras não-críticas e possivelmente alguma que não se caracteriza como tempo real.

Um sistema operacional de tempo real, portanto, precisa garantir que os prazos de execução das suas chamadas de sistema sejam fixos e independentes da carga de processamento. Isso permite verificar se o sistema será capaz de atender os prazos exigidos pela aplicação, prevendo os tempos máximo e mínimo de execução de uma tarefa a partir do evento que a inicia.

Como exemplos, o tempo de escalonamento de uma tarefa deve ser constante, independente do número de tarefas ou de qual tarefa está sendo executada; os tempos de criação, travamento, destravamento e destruição de um semáforo devem ser constantes.

2.4 RECURSOS DO FREERTOS

O sistema operacional de tempo real utilizado para o desenvolvimento da ponte de interfaces foi o sistema operacional de código aberto FreeRTOS, versão 8.2.1. A configuração do sistema operacional utilizada foi de *kernel* preemptivo gerando mil interrupções de *timer* por segundo (*tick* do sistema operacional) para frequência de 72 MHz.

O FreeRTOS fornece diversos recursos que podem ser incorporados nas aplicações, sendo muitos deles importantes para a implementação da ponte de interfaces. A seguir explica-

mos brevemente os conceitos mais expressivos para nossa aplicação: tarefas, semáforos, *mutex* e filas.

Para mais informações sobre o FreeRTOS, o código fonte, documentação e licença de uso estão disponíveis no seu *web site*. Veja (REAL TIME ENGINEERS LTD., 2015).

2.4.1 Tarefa

Uma tarefa, ou *task*, é um contexto de execução. Possui prioridade, estado e regiões de memória para código, pilha e dados. A prioridade identifica a importância da execução dessa tarefa. No FreeRTOS a tarefa de menor prioridade aguarda as de maior prioridade serem executadas. O estado da tarefa informa qual seu estado em relação ao sistema operacional. Os estados geralmente são: em execução, pronta para executar, dormindo, bloqueada ou terminada.

Cada tarefa é executada em seu próprio contexto e é independente de outras tarefas do ponto de vista do sistema operacional. Se a aplicação exigir alguma dependência ou competição entre duas ou mais tarefas (ex: duas tarefas não devem acessar o mesmo recurso compartilhado ao mesmo tempo) fica a cargo do programador garantir a consistência dos dados.

2.4.2 Semáforos

Semáforos são a ferramenta de sincronização mais básica fornecida por um sistema operacional. Podem ser do tipo binário ou contador.

Um semáforo binário é uma variável que funciona como um cadeado, que pode estar travado ou destravado. Só pode ser travado uma vez, sendo isso possível apenas após ter sido destravado. Se uma tarefa travar um semáforo que não está disponível ela entra no estado bloqueado, até que ele seja destravado.

Além do semáforo binário existe o semáforo contador, que funciona de maneira similar, porém pode ser travado diversas vezes (geralmente com um limite) e destravado o mesmo número de vezes. Esses semáforos geralmente são utilizados para controlar recursos que possuem várias instâncias, sendo o seu limite o número de instâncias.

Semáforos são utilizados para sincronização, onde ficam travados a maior parte do tempo, com alguma tarefa aguardando que seja destravado por algum evento.

Em casos como a relação produtor-consumidor, a tarefa que consome fica bloqueada no semáforo até que o produtor sinalize que um novo dado está presente destravando o semáforo. Muitas vezes o produtor é uma interrupção e o consumidor é uma tarefa.

Como exemplo utilizamos novamente a porta serial: a interrupção de recepção é um produtor de dados, sinalizando a tarefa de recepção que um novo dado chegou; e a interrupção

de fim de transmissão é um consumidor, que transmite o próximo dado e sinaliza a tarefa de transmissão que um novo dado pode ser enviado.

2.4.3 Mutex

Assim como o semáforo, o *mutex* também é uma trava. Porém a finalidade do *mutex* é realizar a exclusão mútua entre diversas tarefas que precisam acessar um recurso compartilhado. Exclusão mútua é uma forma de resolver o problema de não-atomicidade das operações, fazendo com que do ponto de vista das tarefas as operações sob exclusão mútua sejam atômicas.

O *mutex* deve ser travado antes de acessar o recurso compartilhado e destravado ao finalizar. Ele mantendo as tarefas aguardando que ele seja destravado, liberando apenas uma das tarefas para prosseguir.

Um exemplo de recurso que deve ser compartilhado com exclusão mútua é a transmissão de uma porta serial. Várias tarefas podem transmitir na porta serial em momentos distintos, porém apenas uma deve fazê-lo por vez. Caso contrário, cada uma transmitirá uma porção do seu texto misturado com o texto de outra.

Porém quando tarefas de diferentes prioridades competem por um mesmo recurso protegido por um *mutex* observamos um efeito chamado de inversão de prioridade. A inversão de prioridade ocorre quando uma tarefa de prioridade B (mais baixa) trava o semáforo com sucesso e logo em seguida uma tarefa de prioridade A (mais alta) é escalonada e tenta travá-lo. Assim a prioridade de execução da tarefa A virtualmente se torna igual a prioridade da tarefa B, pois esta deve aguardar que B seja escalonada novamente e terminar o uso do recurso protegido pelo *mutex*.

Para reduzir os efeitos da inversão de prioridade, o *mutex* geralmente é implementado com um mecanismo de herança de prioridade. Dessa forma a tarefa B terá prioridade sobre as tarefas intermediárias entre A e B, sendo escalonada com maior frequência, finalizando seu processamento mais rapidamente. Isso apenas reduz o efeito da inversão de prioridade, pois a tarefa A ainda precisa aguardar uma tarefa de prioridade mais baixa ser executada, apesar da prioridade ser virtualmente a mesma.

2.4.4 Fila

No FreeRTOS uma fila é um semáforo contador que copia dados, organizando eles em ordem de chegada, geralmente implementado em forma de FIFO (*First In First Out*).

Fila é a principal forma de intercomunicação entre tarefas e interrupção-tarefa, sendo bastante versátil e poderosa (REAL TIME ENGINEERS LTD., 2015). Filas facilitam muito

a implementação da relação produtor-consumidor, pois a fila pode ter um tamanho arbitrário, podendo haver um número de mensagens reservas que podem ser consumidas rapidamente e principalmente porque ambos o produtor e o consumidor podem bloquear nela. O produtor bloqueia aguardando um espaço na fila no caso dela estar cheia (consumo ser mais lento que a produção) e o consumidor bloqueia na fila se estiver vazia (consumo ser mais rápido que a produção).

Algumas implementações permitem inserir a mensagem tanto no início como no fim da fila, permitindo utilizar tanto FIFO como FILO (*First In Last Out*), dando flexibilidade de inserir as mensagens urgentes na frente.

Várias tarefas ou interrupções podem enviar mensagens para uma mesma fila e várias tarefas ou interrupções podem ler essas mensagens de forma segura e sem condições de corrida.

3 INTEGRAÇÃO DAS INTERFACES

Cada interface de comunicação de dados possui um conjunto de protocolos de transmissão e recepção de dados, enquadrando-se em no mínimo uma camada do modelo OSI (*Open Systems Interconnection*) (ALANI, 2014).

O modelo OSI define sete camadas de rede, que formam uma cadeia de processamento de dados. Na transmissão de dados cada camada processa os dados recebidos da camada anterior, adiciona dados relevantes à sua camada e encaminha os dados para a camada seguinte. Na recepção dos dados ocorre o caminho inverso, onde os dados são verificados pela camada, que remove a informação irrelevante para a camada seguinte. Para mais detalhes sobre o modelo OSI veja (ALANI, 2014) e (EDWARDS; BRAMANTE, 2009).

Para integrar duas interfaces diferentes, portanto, precisamos definir um protocolo que transforme as diferenças entre as camadas das interfaces de forma que os dados originais possam ser recuperados a partir dos dados convertidos.

A função de um protocolo de integração de interfaces é converter os dados de uma interface de comunicação para outra, subindo as camadas OSI de uma interface até a camada de aplicação para obter os dados, convertendo os dados de forma adequada e descer as camadas OSI da outra interface para permitir uma transmissão.

Também é função do protocolo de interfaces implementar camadas intermediárias que não estejam definidas em uma das interfaces, caso elas sejam necessárias.

3.1 PERIFÉRICO UART

UART (*Universal Asynchronous Receiver/Transmitter*) é um periférico de transmissão de dados em série, muito utilizado para comunicação entre sistemas embarcados, sistema embarcado e um computador e entre circuitos em uma mesma placa.

A UART define como ocorre a transmissão de um byte a partir de dois níveis de tensão (1/*mark* ou 0/*space*), sendo 1 o nível de repouso. A figura 3.1 mostra a sequência de bits para transmissão de um byte. *Start* e *Stop* são os bits de início e fim, b0 ... b7 são os bits de dados, do menos significativo até o mais significativo e PA é o bit opcional de paridade (AXELSON, 2007).

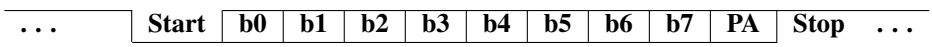

Figura 3.1: Transmissão de um byte na UART.
Fonte: Autoria própria.

3.2 INTERFACE RS-232

O padrão EIA/TIA-232-F, além da parte mecânica, como conectores, define os níveis de tensão da camada física e como os dados são transmitidos. É uma interface de comunicação serial baseada em um periférico UART, sendo baseado na transmissão e recepção de bytes (AXELSON, 2007). Por simplicidade nos referiremos a este padrão por RS-232.

Dessa forma, para realizar uma transmissão no padrão RS-232, é preciso um periférico UART e um *transceiver* para conversão de níveis de tensão. A transmissão é realizada em um barramento ponto-a-ponto, sendo possível realizar a comunicação de apenas dois nodos. Sendo um barramento *full-duplex* cada nodo é capaz de enviar e receber dados simultaneamente.

3.3 INTERFACE RS-485

Assim como o padrão RS-232, o padrão EIA/TIA-485 define os níveis de tensão da camada física e como os dados são transmitidos. Também é uma interface baseada em um periférico UART (AXELSON, 2007).Por simplicidade nos referiremos a este padrão por RS-485.

A transmissão é realizada em um barramento *multi-drop*, sendo possível conectar diversos nodos e todos serem capazes de comunicar.

O barramento pode ser *full-duplex* ou *half-duplex*. Em modo *full-duplex* há um nodo mestre que transmite em uma linha diferencial e todos os outros nodos transmitem em uma segunda linha diferencial, sendo *full-duplex* e *multi-drop* apenas do ponto de vista do nodo mestre, aparentando ser *full-duplex* e ponto-a-ponto sob o ponto de vista dos outros nodos.

Já em modo *half-duplex* o barramento consiste de apenas uma linha diferencial, onde todos os nodos podem transmitir. Como não há um nodo mestre definido fisicamente, o barramento é visto como *half-duplex* e *multi-drop* para todos os nodos.

Analisamos apenas a interface RS-485 em modo *half-duplex* neste trabalho.

3.4 INTERFACE CAN

CAN (*Controller Area Network*) é um protocolo de comunicação serial desenvolvido pela BOSCH para aplicações automotivas, permitindo controle em tempo real com alto nível de segurança (ROBERT BOSCH GMBH, 1991).

O padrão CAN especifica um barramento orientado a mensagens, com priorização intrínseca de mensagens, detecção e correção de erros, definindo as camadas físicas e de enlace (WATTERSON, 2012; CORRIGAN, 2008).

Enquanto os periféricos UART são baseados na transmissão e recepção de bytes, um periférico CAN é baseado no conceito de mensagem. Esta mensagem definida pelo padrão CAN possui um campo identificador da mensagem e um campo de dados, que pode ter comprimento de zero a 8 bytes.

O barramento consiste de uma linha diferencial, onde todos os nodos podem transmitir mensagens. Não há um nodo mestre, portanto o barramento é visto como *half-duplex* e *multi-drop* para todos os nodos.

Atualmente, CAN é o padrão de comunicação serial mais disseminado na indústria automotiva por ser seguro, confiável e barato (SOJKA et al., 2011). Apesar de não possuir tempos de transmissão constantes — o tempo de transmissão depende dos dados sendo transmitidos devido ao *bit-stuffing* utilizado para garantir a ressincronização —, uma rede CAN apresenta um comportamento bastante previsível e por isso é muito utilizada para comunicação em sistemas de tempo real.

A figura 3.2 mostra os campos que compõem uma mensagem CAN, também chamado de quadro CAN. Os principais campos são: identificador (padrão e extendido), tamanho dos dados (DLC – *Data Length Code*), dados e código de redundância cíclica (CRC). Os bits marcados com '0' ou '1' possuem esses valores fixos enquanto os demais variam de acordo com o tipo, o tamanho e o conteúdo da mensagem.

A seleção entre identificador de 11 ou 29 bits é realizada no momento da transmissão a partir dos bits SRR (*Substitute Remote Request*) e IDE (*Identifier Extension*) mostrados na figura. Quando ambos são 1 (recessivos) o identificador estendido é utilizado, adicionando 18 bits ao identificador. Porém quando o bit IDE é 0 (dominante) o identificador estendido não é utilizado, ficando apenas os 11 bits do identificador padrão.

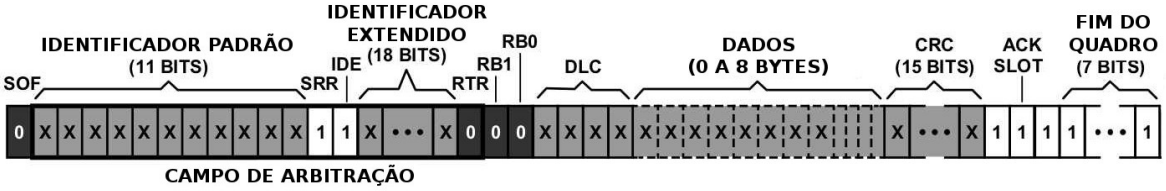

Figura 3.2: Quadro CAN.
Fonte: Adaptado de (WATTERSON, 2012).

3.5 CONVERSÃO RS-485 – RS-232

Tanto o padrão RS-485 quando RS-232 definem apenas a camada física do modelo OSI, estipulando as características físicas e elétricas dos barramentos e dispositivos. Ambos utilizam a transmissão byte a byte UART.

Dessa forma para realizar a conversão entre RS-485 e RS-232 é preciso apenas substituir uma camada física pela outra. Ou seja, ao receber um byte em uma interface, enviamos ele na outra.

Porém em um barramento RS-485 *half-duplex*, se houver algum outro nodo transmitindo em um certo instante e algum outro nodo que deseja transmitir, este deve aguardar até o final da transmissão para evitar colisão no barramento, o que causaria perda da mensagem em transmissão. No entanto, isso não fica a cargo do protocolo de conversão, mas da implementação da interface que veremos no capítulo a seguir.

Este detalhe não afeta a interface RS-232, por ser *full-duplex*, onde é possível enviar e receber simultaneamente sem corromper os dados.

3.6 CONVERSÃO CAN – UART

Nossa intenção ao criar um protocolo CAN – UART é definir um formato de mensagem UART (um bloco de bytes) que possa conter uma mensagem CAN sem que haja perda de informação da camada de aplicação.

A norma CAN define 4 quadros (tipos de mensagens) diferentes que podem ser transmitidos no barramento: quadro de dados (*data frame*), quadro remoto (*remote frame*), quadro de erro (*error frame*) e quadro de sobrecarga (*overload frame*). Destes, apenas o quadro de dados transmite informação. O quadro remoto é utilizado para requisitar dados com o identificador especificado.

O quadro de erro é utilizado para avisar os dispositivos que houve um erro na recepção de uma mensagem, destruindo a mensagem que está sendo enviada e forçando que a transmissão da mesma reinicie. O quadro de sobrecarga tem o propósito de atrasar a transmissão de uma nova mensagem, sendo não-destrutivo.

Percebe-se que os quadros de dados e remotos são essenciais para o funcionamento uma conversão CAN – UART, enquanto que os outros dois quadros não são desejáveis, pois exigiriam uma implementação extremamente complexa para realizar todas as verificações e rotinas, como a repetição ou atraso das mensagens.

Para que seja possível transmitir uma mensagem CAN em um formato byte a byte de forma a permitir a recuperação da mensagem original, é preciso identificar os campos e os dados que são transmitidos na mensagem e definir como eles serão divididos.

3.6.1 Quadro de Dados

O quadro de dados da CAN é formado pelos seguintes campos:

- Início de quadro (1 bit);
- Campo de arbitração (12 ou 32 bits);
- Campo de controle (4 bits);
- Campo de dados (0 a 64 bits);
- Campo de CRC (16 bits);
- Campo de ACK (2 bits); e
- Fim do quadro (7 bits).

Destes, os campos que contém informação são: arbitração (tipo e identificador da mensagem), controle (tamanho dos dados) e dados. Estes campos são os mesmos fornecidos pela camada de aplicação (camada 7 do modelo OSI), sendo os demais campos inseridos nas camadas 1 e 2, definidas pela norma CAN.

O campo de arbitração pode ser dividido em 2 ou 4 bytes (dependendo do identificador ser de 11 ou 29 bits, respectivamente), para o campo de controle pode ser utilizado um byte e para o campo de dados de 0 a 8 bytes, dependendo do valor no campo de controle.

Assim, são necessários de 3 a 13 bytes de dados para transmitir uma mensagem da CAN por uma interface UART sem que haja perda de informação.

3.6.2 Quadro Remoto

O quadro remoto possui os mesmos campos que o quadro de dados, exceto pelo campo de dados. O que define se um quadro é remoto ou de dados é o bit RTR (*Remote Transmission Request*) recessivo no identificador, enquanto no quadro de dados este bit é dominante.

Os campos que contém informação são: arbitração (tipo e identificador da requisição) e controle (tamanho dos dados requisitados).

3.6.3 Protocolo de Conversão CAN – UART

É desejável que o receptor UART seja capaz de detectar o início da mensagem (primeiro byte), o fim da mensagem (último byte) e erros na transmissão da mensagem. As informações necessárias para isso são adicionadas pela camada de enlace (segunda camada do modelo OSI). Para isso podemos adicionar um byte de início (cujo valor é arbitrário) e um byte para o tamanho dos dados no cabeçalho. Isso garante a fácil identificação do início e do fim dos dados que são transmitidos em sequência.

Também adicionamos dois bytes para o campo de CRC no rodapé do quadro para realizar verificação de erros. Pode ser utilizado qualquer algoritmo de geração do CRC, desde

que seja utilizado o mesmo para o barramento. Os bytes de início e tamanho também devem entrar no cálculo do CRC.

A figura 3.3 mostra um exemplo para a implementação de quadro de transmissão de dados em uma interface UART. *DADOS* são os dados fornecidos pela camada de aplicação. *STX*, *TAM* e *CRC* são os campos de início de transmissão, tamanho dos dados e o código de redundância cíclica (camada de enlace).

Camada de aplicação			DADOS	
Camada de enlace	STX	TAM		CRC
Bytes	1	1	0 – 256	2

Figura 3.3: Quadro para transmissão de dados em uma interface UART. Fonte: Autoria própria.

No entanto, o byte de tamanho é redundante, pois é possível saber o tamanho dos dados a partir dos campos de arbitragem (número de bits do identificador) e de controle (número de bytes no campo de dados, se houver). Dessa forma é preciso verificar três bytes para descobrir o tamanho dos dados, sendo possível economizar um byte na transmissão dos dados. Ou seja, aumentamos o processamento para diminuir o número de bytes transmitidos por mensagem.

As figuras 3.4 mostra um exemplo para a implementação dos quadros de dados e remoto para mensagens CAN – UART. *ARB*, *CON* e *DADOS* são os campos de arbitragem, controle e dados (camada de aplicação). *STX* e *CRC* são os campos de início de transmissão e código de redundância cíclica (camada de enlace).

a) Quadro de dados

Camada de aplicação		ARB	CON	DADOS	
Camada de enlace	STX				CRC
Bytes	1	2 – 4	1	0 – 8	2

b) Quadro remoto

Camada de aplicação		ARB	CON	
Camada de enlace	STX			CRC
Bytes	1	2 – 4	1	2

Figura 3.4: Quadros de dados e remoto para mensagens CAN-UART. Fonte: Autoria própria.

Com esta implementação, utilizando identificadores de 29 bits, um quadro remoto ou um quadro com dados de tamanho zero é convertida com o uso de 8 bytes, reduzindo para 6 bytes para identificadores de 11 bits. Cada byte de dados adicionado ao quadro de dados aumenta um byte na mensagem convertida, alcançando um máximo de 16 bytes para uma mensagem.

Este protocolo permite que mensagens CAN sejam convertidas e transmitidas em uma interface UART, sendo ainda possível recuperar a mensagem CAN original.

A figura 3.5 mostra um exemplo de uma mensagem UART para transmissão de um quadro de dados CAN utilizando identificador estendido. A ordem de transmissão dos campos

na UART é da esquerda para a direita, transmitindo primeiro o campo SYNC e por último o CRC.

Para sinalizar se o identificador é de 11 ou 29 bits pode-se utilizar um dos bits mais significativos do primeiro byte do identificador UART, do qual uma polaridade denota dois bytes no identificador UART (11 bits do identificador CAN) e a outra polaridade denota quatro bytes no identificador UART (29 bits do identificador CAN).

	SYNC	**Identificador**				**Tamanho**	**Dados**			**CRC**	
Tamanho	1 Byte	4 Bytes				1 Byte	0 – 8 Bytes			2 Bytes	
Valor	0x02					0 – 8					
Offset	0	1	2	3	4	5	6	. . .	5+n	6+n	7+n

Figura 3.5: Mensagem UART para quadro de dados CAN.
Fonte: Autoria própria.

3.6.4 Conversão CAN – RS-232

De acordo com a própria norma, o padrão CAN define a primeira e a segunda camada do modelo OSI (camada física e enlace), incluindo as subcamadas *MAC* e *LLC* (controle de acesso ao meio e controle de *link* lógico) da camada de enlace. É formado por uma linha diferencial, *half-duplex* e multinodo.

Para que uma mensagem CAN possa ser transmitida na RS-232 é preciso que esta mensagem seja convertida para um formato, que possa ser transmitido byte a byte através de um periférico UART. Vemos na seção 3.6 um exemplo de protocolo de conversão de mensagens CAN para UART. Após convertida para um protocolo UART, a mensagem recebida pela CAN pode ser transmitida na interface RS-232.

A CAN é um barramento *half-duplex* (dois nodos não podem transmitir simultaneamente), porém diferente da RS-485 não há necessidade de verificar se o barramento está livre antes de enviar uma mensagem. A CAN faz uso da arbitragem de mensagens, parando a transmissão da mensagem de menor prioridade assim que o controlador CAN detecta que há outro nodo transmitindo uma mensagem de maior prioridade.

3.6.5 Conversão CAN – RS-485

Da mesma forma que na conversão para RS-232, a conversão da CAN para a RS-485 exige que as mensagens CAN sejam convertidas para um formato que possa ser enviado através de uma UART, ou seja, um formato divisível byte a byte.

No entanto, como RS-485 é um padrão *half-duplex* sem a facilidade da arbitragem de mensagens, o *hardware* não é capaz de detectar colisões, necessitando monitorar o barramento e aguardar que esteja livre para que seja iniciada uma transmissão.

Uma vez que esteja pronta a conversão de RS-232 para RS-485, é possível enviar a mensagem CAN já convertida para o protocolo UART, para a RS-232 e configurá-la para que envie nessa interface e que ela também realize a transmissão na RS-485.

4 RELAÇÃO PRODUTOR-CONSUMIDOR

Na implementação de uma ponte de interfaces encontramos o problema produtor-consumidor, também conhecido como problema do *buffer* limitado, na relação entre a recepção de uma interface com a transmissão de outra interface.

No problema produtor-consumidor, temos uma ou mais tarefas que produzem itens e uma ou mais que consomem esses itens. Os produtores inserem itens em um *buffer* limitado enquanto os consumidores removem itens deste *buffer*.

A menos que sejam realizadas de maneira bem planejada, estas inserções e retiradas concorrentes de itens no *buffer* facilmente causam condições de corrida. Este problema é solucionado facilmente através do uso de semáforos ou filas de mensagens, bloqueando a tarefa produtora caso tente inserir um item em uma fila cheia (produção mais rápida que consumo) ou bloqueando a tarefa consumidora caso tente retirar um item em uma fila vazia (consumo mais rápido que a produção) (TANENBAUM, 2009).

Para uma ponte de interfaces, a recepção de mensagens representa o produtor e a transmissão de mensagens representa o consumidor, havendo tantos produtores e consumidores quanto o número de interfaces.

Um ponto importante é que a produção de uma interface é tão rápida quanto o seu consumo. Portanto, quando temos várias interfaces com taxas de transmissão ou protocolos diferentes teremos na maioria dos casos produção e consumo desbalanceados entre as interfaces. Ou seja, a tarefa de recepção de uma interface mais rápida pode produzir mais mensagens do que a tarefa de transmissão de uma interface mais lenta pode consumir.

Porém uma interface não pode controlar o número de mensagens que recebe, como no problema produtor-consumidor padrão, citado acima. Isso significa que não é possível bloquear o produtor para impedi-lo de produzir. A principal consequência disso é que não importa quão grande seja o *buffer* de mensagens, se uma interface mais rápida receber uma rajada de mensagens suficientemente grande, haverá mensagens descartadas e consequentemente perda de eficiência do sistema.

Não há forma de contornar este problema, o que podemos fazer é dimensionar o *buffer* de forma que suporte uma certa rajada de mensagens. Outro ponto importante é discutir qual a ação mais adequada para o produtor quando mensagens são descartadas devido ao *buffer* estar cheio. Estes tópicos são abordados no restante deste capítulo.

4.1 CARGA NO CONSUMIDOR

É necessário verificar se o consumidor será capaz de transmitir a quantidade média de mensagens que são produzidas. Trata-se de analisar a taxa média de produção com a taxa

máxima de consumo.

A frequência máxima de produção de mensagens deve seguir a equação 4.1, onde f_{PROD} é a frequência média de produção de mensagens, f_{CONS} é a frequência máxima de consumo de mensagens, T_{PROD} é o tempo de produção de uma mensagem e T_{CONS} é o tempo de consumo de uma mensagem.

$$f_{PROD} \leq f_{CONS} = \frac{1}{T_{CONS}} \tag{4.1}$$

Sendo η_{PROD} a utilização do barramento da interface produtora, para que seja possível o consumo de todas as mensagens produzidas, a utilização média do barramento produtor deve ser limitada ao máximo que a interface consumidora é capaz de enviar, dado pela equação 4.3.

$$\eta_{PROD} = f_{PROD}\ T_{PROD} \tag{4.2}$$

$$\eta_{PROD} \leq T_{PROD}\ f_{CONS} = \frac{T_{PROD}}{T_{CONS}} \tag{4.3}$$

Se as equações 4.1 e 4.3 forem falsas então o consumidor não será capaz de transmitir todas as mensagens recebidas, sendo necessário diminuir o tempo de consumo ou aumentar o tempo de produção. Isso pode ser realizado alterando taxas de transmissão ou melhorando a eficiência do protocolo de transmissão do consumidor.

4.2 DIMENSIONAMENTO DO *BUFFER*

Antes de dimensionar o *buffer* precisamos conhecer como é realizada a comunicação nos barramentos. O tempo de transmissão da mensagem, o maior tamanho de rajada (número de mensagens enviadas em sequência) e a frequência em que as rajadas podem ocorrer são valores muito importantes para serem considerados nos barramentos mais rápidos. Nos barramentos mais lentos precisamos essencialmente do tempo de transmissão da mensagem.

As análises realizadas a seguir consideram que os barramentos estejam completamente livres no momento em que precisam realizar uma transmissão e que o tempo de processamento é desprezível em comparação com os tempos de transmissão e recepção.

4.2.1 Produtor Lento e Consumidor Rápido

A figura 4.1 mostra os tempos de recepção de uma interface T_{PROD} e de transmissão de outra interface T_{CONS}. A interface mais lenta recebe os dados, sendo um produtor lento, enquanto a interface mais rápida transmite esses dados, sendo um consumidor rápido.

Na figura há vários instantes importantes, identificados pelas linhas verticais. O primeiro instante T_1 é o início da recepção da produção. O segundo T_2 é o fim da produção, o início do consumo e também há um novo início de produção. O terceiro instante T_3 marca o fim do consumo, ficando o consumidor em estado bloqueado até o próximo instante, quando o produtor finaliza. No quarto instante T_4 é o fim da produção e o início do consumo.

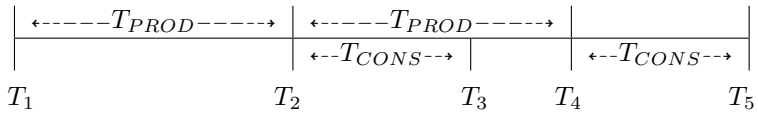

Figura 4.1: Comparação dos tempos de produtor lento e consumidor rápido. Fonte: Autoria própria.

Note que o consumidor fica aguardando uma nova produção durante o tempo $T_{PROD} - T_{CONS}$. Isso mostra que, quando uma interface demora mais para produzir do que outra interface demora para consumir, independente do tamanho da rajada que ocorre na produção não é necessário uso de um *buffer*, desde que o barramento consumidor esteja livre no momento da transmissão.

Geralmente não há garantia de que o barramento estará livre no momento de início da transmissão e, mesmo que mais rápida, esta interface precisará de um *buffer* para armazenar as mensagens. O tamanho do *buffer* precisa ser calculado de maneira semelhante ao caso a seguir (seção 4.2.2), levando em conta o maior tamanho de rajada que ocorre no barramento.

4.2.2 Produtor Rápido e Consumidor Lento

No caso onde o produtor é rápido e o consumidor é lento a dinâmica se torna um pouco diferente. O produtor pode produzir uma nova mensagem antes que o consumidor seja capaz de terminar a primeira, exigindo a existência de um *buffer* para armazenar estas mensagens temporariamente.

A figura 4.2 mostra a transmissão de $N + 1$ mensagens e compara os tempos de recepção T_{PROD} e de transmissão T_{CONS} de duas interfaces, onde a interface rápida recebe dados e envia eles para a interface lenta (produção mais rápida que o consumo). A variável N é o tamanho da rajada que pode ser recebida sem que haja perda de mensagens, T_{PROD} é o tempo de produção e T_{CONS} é o tempo de consumo.

As linhas verticais da figura identificam os instantes mais importantes para a análise. O primeiro instante T_1 mostra o início da produção da primeira mensagem e o segundo instante T_2 mostra o fim da produção e início do consumo da primeira mensagem. No segundo também é iniciada a produção de uma rajada de N mensagens (iniciando a utilização do *buffer*), finalizando a produção no terceiro instante T_3. No quarto instante T_4, o consumidor termina de

consumir as mensagens do *buffer* (fim da rajada) e no quinto T_5 termina a transmissão da última mensagem.

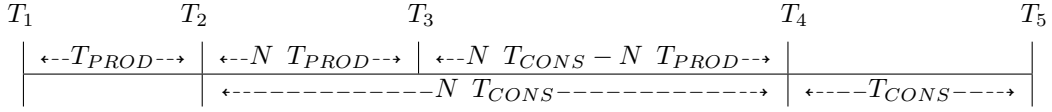

Figura 4.2: Comparação dos tempos de produtor rápido e consumidor lento. Fonte: Autoria própria.

Os instantes citados formam quatro intervalos. No primeiro intervalo (T_1 até T_2) apenas a produção está ativa. No segundo intervalo (T_2 até T_3) a produção e o consumo estão ativos. No terceiro e quarto intervalos (T_3 até T_4 e T_4 até T_5, respectivamente) apenas o consumidor está ativo. O *buffer*, no entanto, é utilizado apenas no segundo e terceiro intervalos.

No segundo intervalo a utilização do *buffer* cresce em uma taxa menor do que $1/T_{PROD}$, devido ao consumidor; essa taxa é igual a $1/T_{PROD} - 1/T_{CONS}$. No terceiro intervalo a utilização do *buffer* decresce na taxa $1/T_{CONS}$.

O ponto onde ocorre a máxima utilização do *buffer* ocorre no terceiro instante, ou seja, no fim da produção. Já o ponto onde o *buffer* volta a ficar livre ocorre no final do consumo, o quarto instante. Para que não haja perda de mensagens, este intervalo de tempo precisa ser limitado.

O limite desse intervalo é igual ao tempo que o consumidor demora para transmitir todas as N_{BUFF} mensagens aguardando no *buffer*. Assim chegamos a equação 4.4, onde N é o tamanho da rajada transmitida e N_{BUFF} é o tamanho do *buffer*.

$$N\ T_{CONS} - N\ T_{PROD} < N_{BUFF}\ T_{CONS} \tag{4.4}$$

O número máximo N de mensagens que podem ser enviadas em uma rajada depende do tamanho da fila de envio N_{BUFF} do consumidor e dos tempos de envio das mensagens T_{PROD} e T_{CONS}. A equação 4.5 mostra a relação entre essas variáveis.

$$N < \frac{N_{BUFF}}{1 - T_{PROD}/T_{CONS}} \tag{4.5}$$

Nota-se duas coisas: se $T_{PROD} \ll T_{CONS}$ teremos $N \approx N_{BUFF}$, pois o consumidor não será capaz de consumir mensagens enquanto o produtor enfileira o restante das mensagens no *buffer*; e se o consumidor estiver pronto para consumir uma mensagem no momento em que o produtor recebê-la será possível enviar uma mensagem a mais com sucesso.

A equação 4.5 governa qualquer relação produtor-consumidor onde o consumidor seja mais lento e não tem o controle sobre o produtor, utilizando um *buffer* para armazenar os objetos excedentes temporariamente.

Ainda podemos concluir que se o consumidor estiver a consumir os objetos do *buffer*, o produtor pode então produzir até N objetos em uma rajada, sendo N calculado com N_{BUFF}

igual ao número de espaços livres no *buffer*.

Assim, a partir dos tempos de produção e consumo (ou seja, os tempos de recepção e transmissão de mensagens), é possível definir o tamanho do *buffer* necessário para transmitir uma rajada de tamanho arbitrário ou conhecer a maior rajada suportada por um *buffer* de tamanho arbitrário.

4.3 DESCARTE DE MENSAGENS

Dimensionar o *buffer* para que ele suporte uma certa rajada de mensagens é essencial para garantir a performance de uma ponte de interfaces. No entanto, é possível que em algum momento seja necessário descartar uma ou mais mensagens. É necessário tomar alguns cuidados em relação a como descartá-las.

A forma mais simples de realizar o descarte de mensagens é manter o produtor bloqueado ao tentar escrever em um *buffer* cheio, de forma que ele aguarde um espaço no *buffer* ser liberado através de uma leitura do consumidor. Temos neste caso um problema, pois mensagens que então chegam ao produtor não serão processadas por nenhuma das interfaces consumidora apesar de apenas uma estar cheia.

Portanto, bloquear o produtor gera um efeito colateral indesejado nas outras interfaces. O mesmo produtor pode enviar mensagens para mais de um consumidor e bloquear ao escrever no *buffer* de um consumidor impede que o outro consumidor receba mensagens, fazendo que todas as interfaces consumidoras tenham o mesmo limite de performance.

Ou seja, um consumidor com *buffer* mal dimensionado ou com barramento muito ocupado (impedindo a transmissão imediata) vai forçar perdas de mensagens em outro consumidor e impedir que ele alcance seu máximo desempenho.

Por outro lado esta abordagem permite que todas as interfaces consumidoras transmitam as mesmas mensagens, independente do tamanho de suas filas, o que em alguns casos pode ser desejável.

Outra abordagem, que foi a opção utilizada neste trabalho, é descartar a mensagem para aquele consumidor cujo *buffer* está cheio e seguir distribuindo a mensagem para os outros consumidores. Isso implica que diferentes consumidores poderão receber um conjunto diferente de mensagens quando estiverem operando nos limites dos barramentos, seja devido às rajadas de produção ou pelo uso do barramento por outros módulos.

Isso é mais adequado quando não se conhece os detalhes dos consumidores de antemão, podendo por exemplo alterar sua taxa de transmissão, o tamanho da mensagem ou a carga do barramento, o que interfere diretamente nas características do dimensionamento do *buffer*.

4.4 FILTRO DE MENSAGENS

Em muitas aplicações nem todas as mensagens precisam ser copiadas de uma interface para outra, sendo possível utilizar um filtro de mensagens para selecionar aquelas que devem ser enviadas pela interface.

A filtragem das mensagens pode ser realizada em dois pontos: após ser retirada do *buffer* pelo consumidor ou antes de ser colocada no *buffer* pelo produtor. Utilizamos filtragem das mensagens pelo consumidor neste trabalho.

A filtragem realizada pelo consumidor afeta o tempo de resposta apenas do próprio consumidor e de outros consumidores de menor prioridade, permitindo melhores latências para os consumidores de maior prioridade. Realizar a filtragem após retirá-las do *buffer*, se houver um, permite um aumento na margem da frequência média de produção, ou seja, diminuindo a frequência de produção eficaz. Porém não melhora significativamente o tamanho das rajadas, pois a filtragem é realizada após a adição da mensagem no *buffer*, ocupando o espaço mesmo se não for necessário transmiti-la.

Realizando a filtragem no produtor antes de enfileirar a mensagem no *buffer*, por sua vez, causa atrasos variáveis no enfileiramento das mensagens de outros consumidores; isso causa a perda do determinismo do produtor, porém os atrasos geralmente são muito pequenos comparados com o tempo de envio de uma mensagem, não afetando o desempenho de forma significativa. Portanto, realizar a filtragem antes da inserção no *buffer* diminui a taxa de produção eficaz de mensagens, e melhora significativamente o tamanho das rajadas possíveis, pois as mensagens das rajadas são filtradas antes de serem inseridas no *buffer*.

Caso o tempo de filtragem das mensagens for da ordem do tempo de consumo de qualquer consumidor o determinismo do produtor pode ser necessário. Nesse caso a saída é utilizar um *buffer* e tarefa intermediária para realizar a filtragem após a produção, sendo esta tarefa o consumidor das recepções das interfaces e o produtor para a transmissão da interface, gerando uma relação produtor-consumidor adicional, porém com consumo muito mais rápido que a interface, possuindo as mesmas vantagens citadas no parágrafo anterior.

4.5 MÚLTIPLOS *BUFFERS*

Apesar de não terem sido utilizados neste trabalho é importante citar a possibilidade de utilizar múltiplos *buffers*, pois na maioria dos barramentos de sistemas de tempo real há no mínimo dois conjuntos de mensagens, selecionadas por prioridade de transmissão.

As mensagens de maior prioridade são filtradas e enfileiradas em um *buffer* separado, que possui maior prioridade de transmissão.

Dessa forma as mensagens presentes no *buffer* de alta prioridade são desenfileiradas e transmitidas antes das mensagens presentes no *buffer* de baixa prioridade, garantindo o desempenho da comunicação de tempo real e permitindo mensagens de maior latência aguardarem.

5 IMPLEMENTAÇÃO DAS INTERFACES

Uma das partes mais importantes na implementação de uma ponte de interfaces é a programação das rotinas que controlam o *hardware*. Discutimos os vários métodos que realizam estas rotinas, desenvolvendo seus pontos fortes e fracos.

Os periféricos de interesse no *hardware* para a implementação da ponte são duas UARTs, ligadas a *transceivers* RS-232 e RS-485, e uma CAN. Cada periférico é capaz de receber e transmitir dados, possuindo suas peculiaridades, que serão analisadas em relação aos métodos de controle de *hardware*.

5.1 CONTROLE DE HARDWARE

Após dar um comando ao *hardware*, há basicamente duas formas de verificar se a operação finalizou. Estas são *polling* e interrupção, que são discutidas nesta seção. Para simplificar os exemplos utilizamos uma ponte RS-232 para RS-485, transmitindo os dados recebidos um a um.

5.1.1 Polling

O método mais básico de controle de *hardware* é o chamado *polling*, consiste em checar as *flags* de recepção dos periféricos constantemente e ao transmitir aguardar o comando terminar através da verificação de *flags* nos registradores do periférico.

Como vantagens desta abordagem são a simplicidade do código e em casos simples também a velocidade de execução. O código gerado é simples e bastante sequencial, com acesso a poucos registradores dos periféricos, permitindo fácil leitura e compreensão do fluxo de execução. A velocidade execução pode ser maior pois não é necessário salvar e restaurar o contexto da aplicação ao início e fim de cada interrupção[1], necessários para manter os registradores de trabalho intactos para a aplicação. Essa velocidade de execução rapidamente se dissolve quando o número de rotinas utilizando *polling* aumenta.

Uma das formas de *polling* mais comuns é a utilização de um laço de espera ocupada. A principal desvantagem dessa implementação é que ela impede que outras rotinas sejam executadas ou outros periféricos sejam verificados enquanto a operação não finaliza. Outro

[1]Uma troca de contexto consiste em salvar os registradores de trabalho e *status* na pilha antes de executar o tratamento da interrupção e devidamente restaurar os valores salvos para os registradores após a execução. Trocas de contexto também são utilizadas por sistemas operacionais para escalonar diferentes tarefas sem que seus dados sejam corrompidos e sem que as tarefas precisem estar cientes do escalonamento.

problema ocorre quando há outras rotinas que também precisam ser executadas pela aplicação, como monitorar e atualizar IHM (Interface Homem Máquina), atrasando o tratamento dos dados recebidos, o que pode causar perda de dados recebidos e perda de tempo de transmissão. Este método é muito utilizado em sistemas simples, onde há apenas uma ou poucas funções que precisam serem realizadas.

Um método mais adequado é um laço com diversas verificações, uma para cada função, que será executada apenas quando necessário. Esta técnica é mais utilizada quando há várias funções que precisam ser executadas, como recepção, processamento, transmissão.

O algoritmo 5.1 mostra um exemplo de código utilizando *polling*. Na linha 1 temos uma função que verifica se há dados recebidos na interface RS-232, a função da linha 6 obtém os dados recebidos pela RS-232 e na linha 11 temos a função de transição na interface RS-485. Na linha 17 tempos a função principal que realiza a recepção e transmissão dos dados entre as interfaces. É fácil perceber que não é adequado para a aplicação, devido à espera ocupada na linha 13 na função de transmissão na RS-485. Se a interface RS-232 for mais rápida que a RS-485 ocorrerá perda de dados, pois receberia dois ou mais dados no intervalo em que outra transmite, causando o descarte de dados.

Algoritmo 5.1 Código exemplo de *polling*.

```
1  int verifica_rx_rs232(void) {
2    /* Retorna estado da flag. */
3    return UART0_STATUS & RX;
4  }
5
6  char recebe_rs232(void) {
7    /* Retorna caractere recebido. */
8    return UART0_RXDATA;
9  }
10
11 void transmite_rs485(char ch) {
12   /* Aguarda transmissão atual finalizar e envia. */
13   while(UART1_STATUS & TX) {}
14   UART1_TXDATA = ch;
15 }
16
17 void ponte(void) {
18   char ch;
19   if(verifica_rx_rs232()) {
20     ch = recebe_rs232();
21     transmite_rs485(ch);
22   }
23   /* ... */
24 }
```

É possível remover este problema e também remover em parte a espera ocupada na transmissão utilizando uma variável para indicar que a interface está em modo de transmissão, outra variável para guardar o dado que deve ser transmitido e uma função para verificar se a transmissão anterior já finalizou. A espera ocupada ainda existe, porém não acontece mais do

processador ficar preso em um laço verificando apenas uma interface.

Vejamos agora o algoritmo 5.2, que utiliza funções definidas no algoritmo 5.1. A função na linha 1 é utilizada para verificar se a interface RS-485 está transmitindo e a função na linha 6 inicia uma transmissão de um byte e retorna imediatamente, sem aguardar o fim dessa transmissão. Nesse caso, ao receber um dado na interface RS-232, a interface RS-485 é colocada em modo de transmissão e o dado recebido é colocado na variável de transmissão. Ao verificar que a interface RS-485 está em modo de transmissão, verifica-se se a transmissão anterior já finalizou, transmite o dado e tira a interface do modo de transmissão.

Algoritmo 5.2 Código exemplo de *polling*.

```
1  int transmitindo_rs485(void) {
2    /* Retorna estado da flag. */
3    return UART1_STATUS & TX;
4  }
5
6  void transmite_rs485(char ch) {
7    /* Inicia transmissão e retorna. */
8    UART1_TXDATA = ch;
9  }
10
11 void ponte(void) {
12   static char estado485=0, ch485;
13   if(verifica_rx_rs232()) {
14     estado485 = 1;
15     ch485 = recebe_rs232();
16   }
17   /* ... */
18   if(estado485 && !transmitindo_rs485()) {
19     transmite_rs485(ch485);
20     estado485 = 0;
21   }
22   /* ... */
23 }
```

Fazendo isso para todos os periféricos podemos garantir que todos os dados recebidos serão processados, desde que o processador seja capaz de realizar o processamento. No entanto, dados ainda podem não ser transmitidos de uma interface para outra devido a diferenças de velocidade de transmissão, podendo facilmente perder dados que fazem parte de um bloco maior, corrompendo a mensagem em transmissão.

Este problema pode ser minimizado substituindo a variável com o dado a ser transmitido por uma fila circular[2]. Por exemplo, ao receber um dado na interface CAN, a interface RS-232 é colocada em modo de transmissão e o dado recebido é inserido na fila. Ao identificar que a interface RS-232 está em modo de transmissão, é verificado se a transmissão anterior já

[2]Uma fila circular é uma estrutura de dados capaz de armazenar vários dados do mesmo tipo, sendo recuperados na mesma ordem em que foram inseridos. Uma fila possui duas operações básicas: *push* e *pop*, que inserem e removem um item da lista, respectivamente. Após a remoção de um dado o espaço que ele ocupava pode ser utilizado novamente pela inserção de um novo dado.

finalizou, o dado mais antigo da fila é recuperado da fila e transmitido, retirando a interface do modo de transmissão caso a fila esteja vazia.

Ainda não há garantia que todos os dados recebidos serão transmitidos nas outras interfaces, porém é possível garantir que uma certa quantia de dados recebidos em sequência serão transmitidos. A quantidade de mensagens em sequência que pode ser transmitida depende das velocidades de transmissão das interfaces e do tamanho da fila, como veremos na seção 4.

5.1.2 Interrupção

O método de controle de *hardware* baseado em interrupções consiste em configurar o periférico para gerar uma interrupção no processador quando algum evento importante acontece, como a recepção de um dado ou o fim da transmissão, desviando o fluxo de execução para o *handler* (rotina de tratamento) da interrupção, uma função especial que checa as *flags* do periférico para verificar que evento gerou a interrupção, realiza o tratamento e limpa a interrupção.

Utilizar interrupções para controle do *hardware* é um pouco mais trabalhoso, pois envolve uma etapa mais complexa de configuração, especialmente quando o tratamento da interrupção interagem com o restante da aplicação, que pode gerar condições de corrida e outros problemas.

A principal vantagem de utilizar interrupções é que estas são tratadas assim que o evento ocorre (exceto por um atraso devido ao desvio do fluxo de execução). Não há necessidade de perder tempo de processamento verificando as *flags* dos periféricos e também é possível que a aplicação entre em um estado de baixo consumo, aguardando pela ocorrência de uma interrupção.

Como desvantagem, há um pequeno atraso no início do tratamento da interrupção devido ao desvio do fluxo de execução e outro atraso ao retornar o fluxo de execução para a aplicação.

Em casos onde o periférico possui alta velocidade, de forma que o tempo de entrada e saída da interrupção influenciem na performance, e precisa ser acessado com maior velocidade possível, porém não é necessária uma garantia de desempenho pode-se utilizar o método *polling*. Porém atualmente os periféricos de alta velocidade geralmente possuem filas de transmissão e recepção, de forma que vários dados possam ser lidos e escritos em um único tratamento de interrupção, reduzindo a influência dos atrasos de entrada e saída da interrupção.

O código fonte, por sua vez, fica mais complexo devido as interrupções poderem ocorrer a qualquer instante, desviando o fluxo de execução temporariamente para tratar o evento que a causou. Quando as interrupções interagem diretamente com a aplicação, ou seja, ambos modificam as mesmas variáveis, se não houver um planejamento cuidadoso do uso das variáveis com acesso concorrente podem ocorrer condições de corrida, gerando resultados imprevisíveis.

O algoritmo 5.3 mostra um exemplo de código utilizando interrupção em conjunto com filas de transmissão e recepção. Na linha 1 temos a função de tratamento de interrupções da RS-232, tratando a recepção de dados (linha 2, colocando o dado recebido na fila de recepção da interface) e a transmissão de dados (linha 6, colocando um novo dado a ser transmitido na fila de transmissão). Na linha 12 temos a função de inicialização da transmissão na interface RS-485, que verifica se uma transmissão está sendo realizada e escreve no *buffer* de transmissão caso não estiver. O código do *handler* da RS-485 é omitido, pois é semelhante ao código do *handler* da RS-232.

Algoritmo 5.3 Código exemplo de interrupção.

```
1  int uart0_handler(void) {
2    if(UART0_STATUS & RX) {
3      /* Recebeu dado. Enfileirar. */
4      fila_RX_rs232_push(&UART0_RXDATA);
5    }
6    else if(UART0_STATUS & TXIF){
7      /* Fim de transmissão. Envia próximo dado. */
8      fila_TX_rs232_pop(&UART0_TXDATA);
9    }
10 }
11
12 void inicia_transmissao_rs485(void) {
13   /* Inicia a transmissão se não estiver
14     transmitindo. */
15   __DESABILITA_INTERRUPCAO();
16   if(!transmitindo_rs485())
17     fila_TX_rs485_pop(&UART1_TXDATA);
18   __HABILITA_INTERRUPCAO();
19 }
20
21 void ponte(void) {
22   char ch;
23   if(fila_RX_rs232_pop(&ch)) {
24     fila_TX_rs485_push(&ch);
25     inicia_transmissao_rs485();
26   }
27   /* ... */
28 }
```

Por exemplo, ao receber um dado na interface CAN, o *handler* de interrupção é executado e o dado recebido é enfileirado pela função *push* na fila de recepção da interface CAN. A aplicação continua sua execução, tentando desenfileirar um dado da fila de recepção da interface CAN através da função *pop*. Em caso de sucesso, ou seja, havia um dado na fila, este dado é enfileirado na fila de transmissão da interface RS-232. Em seguida é verificado se a interface RS-232 está em transmissão. Se não estiver em transmissão um dado da fila é retirado e enviado. Caso contrário, a interrupção do fim da transmissão atual tratará de enviar o dado. Ao finalizar a transmissão, o *handler* da interface RS-232 é executado e tenta desenfileirar e enviar outro dado da fila de transmissão.

Percebemos que com esta abordagem o atraso de recepção de um dado (tempo entre a recepção e a leitura) não depende da aplicação, fazendo com que não haja perda de dados recebidos, desde que haja espaço na fila de recepção. O atraso de transmissão também não depende da aplicação, evitando períodos sem dados sendo transmitidos.

Para evitar condições de corrida, as funções *push* e *pop* precisam ser atômicas, ou seja, devem ser não-divisíveis. Outra operação que precisa ser atômica é testar se a interface está transmitindo e, caso não estiver, iniciar a transmissão. No algoritmo 5.3 a atomicidade desta operação é obtida desabilitando as interrupções.

A estratégia que utiliza interrupções geralmente é a mais adequada do ponto de vista de tempo de resposta aos eventos, permitindo que o tempo entre a ocorrência e o tratamento dos eventos seja o mais constante possível.

No entanto, ainda não garante que a aplicação seja capaz processar os dados assim que forem enfileirados, pois ela pode estar executando algum processamento intenso, para só então verificar as filas das interfaces.

5.1.3 Interrupção com Sistema Operacional

Utilizando interrupções para identificar os eventos do *hardware* já é capaz de garantir que não haja perda de dados recebidos. Com um sistema operacional podemos dividir as rotinas da aplicação em diferentes tarefas, atribuindo prioridades diferentes para elas. Assim podemos garantir que os dados recebidos serão tratados pela aplicação e enviados nas outras interfaces, realizando processamento intenso em tarefas de baixa prioridade.

Um sistema operacional fornece uma série de ferramentas para auxiliar o desenvolvimento da aplicação e a interação aplicação-interrupção. Em primeiro lugar o sistema operacional fornece o conceito de tarefa, ou contexto de execução, e possuindo prioridade, estado e suas regiões de memória para código, pilha e dados. Várias tarefas podem estar em execução simultaneamente, dividindo o tempo de processamento entre elas, de acordo com a regra de escalonamento. Assim, várias tarefas podem ser criadas para que cada uma realize uma função.

Com isso é possível segmentar o código de forma que cada tarefa execute uma função bem definida, permitindo que o código da aplicação seja muito mais compreensível.

No entanto a interação entre tarefas e tarefa-interrupção torna-se mais complexa, aumentando as possibilidades de condição de corrida[3], devido aos diversos contextos de aplicação. Para evitar estes problemas podemos utilizar filas e semáforos.

[3]Quando há vários contextos de execução simultâneos (interrupções e/ou várias tarefas de sistemas operacionais) pode ocorrer que o resultado das operações dependa da ordem de execução. Esse efeito indesejável é chamado condição de corrida, sendo resolvido realizando as operações atomicamente. Um exemplo de condição de corrida é quando uma tarefa inicia a retirada de um item de uma fila com operações não atômicas e, antes que finalize, uma interrupção ocorre e insere um novo item na fila, corrompendo o estado da fila.

Filas e semáforos são ferramentas importantes fornecidas pelos sistemas operacionais. Filas são utilizadas para troca de dados entre contextos (tarefas ou interrupção-tarefa), geralmente são implementadas como FIFO. Semáforos servem para sincronização entre tarefas ou interrupção-tarefa, onde um contexto aguarda em estado bloqueado até que outro contexto libere o semáforo. Semáforos também podem realizar exclusão mútua entre tarefas para controlar o acesso a recursos compartilhados, onde apenas o contexto que liberou o semáforo pode acessá-lo.

Com estas ferramentas podemos realizar implementações que não consomem tempo de processamento até que haja algum dado a ser processado. Ou seja, não necessita verificar constantemente se há algum dado a ser processado.

O algoritmo 5.4 mostra um exemplo de código utilizando sistema operacional e filas (*queue*) em conjunto com interrupções. As funções das linhas 1 e 12 são o tratador de interrupções da RS-232 e a função de inicialização de transmissão da RS-485, respectivamente, e funcionam identicamente ao algoritmo 5.3. Porém ao invés de utilizar uma função única para tratar todas as interfaces, utiliza-se uma tarefa do sistema operacional para monitorar cada uma das filas de recepção, veja a tarefa de recepção da RS-232 na linha 21. Esta tarefa aguarda que algum dado seja escrito na fila de recepção da interface através da função *queue_receive_pend*, que mantém a tarefa bloqueada até que um dado esteja disponível na fila, então envia os dados recebidos à fila de transmissão das outras tarefas. Assim, cada interface possui um tarefa de recepção monitorando sua fila, que envia os dados recebidos às filas de transmissão. Não utilizamos semáforos neste exemplo, porém ele é importante quando a tarefa precisa aguardar o término da transmissão. O código do *handler* da RS-485 é omitido, pois é semelhante ao código do *handler* da RS-232.

Há duas funções de fila utilizadas no exemplo: *send* e *receive*. A primeira insere na fila o dado apontado e a segunda remove da fila um dado e o escreve ele no endereço apontado. As filas são gerenciadas pelo código do sistema operacional, que deve garantir a atomicidade no acesso as filas para evitar condições de corrida.

Também utilizamos duas variações da função *receive*: a versão bloqueável (utilizada na tarefa) e a versão não-bloqueável (utilizada na interrupção). Utilizando a versão bloqueável, caso houver dados na fila a função retorna imediatamente, de forma que o dado possa ser processado. Caso não houver nenhum dado a tarefa entra em estado bloqueado, voltando à execução quando algum dado for inserido na fila.

Algoritmo 5.4 Código exemplo com sistema operacional.

```
1  int uart0_handler(void) {
2    if(UART0_STATUS & RX) {
3      /* Recebeu dado. Enfileirar. */
4      queue_send(queue_RX_rs232, &UART0_RXDATA);
5    }
6    else if(UART0_STATUS & TXIF){
7      /* Fim de transmissão. Envia próximo dado. */
8      queue_receive(queue_TX_rs232, &UART0_TXDATA);
9    }
10 }
11
12 void inicia_transmissao_rs485(void) {
13   /* Inicia a transmissão se não estiver
14     transmitindo. */
15   __DESABILITA_INTERRUPCAO();
16   if(!transmitindo_rs485())
17     queue_receive(queue_TX_rs45, &UART1_TXDATA);
18   __HABILITA_INTERRUPCAO();
19 }
20
21 void task_rs232(void) {
22   char ch;
23
24   /* Esta tarefa fica bloqueada até que a fila receba
25     um dado. */
26   queue_receive_pend(queue_RX_rs232, &ch);
27
28   queue_send(queue_TX_rs485, &ch);
29   inicia_transmissao_rs485();
30 }
```

A versão não-bloqueável retorna imediatamente independente de haver ou não dados na fila. É obrigatório utilizar a versão não-bloqueável na interrupção, pois o sistema operacional não pode salvar o contexto de interrupção e bloquear a interrupção. Isso só pode ser feito com tarefas.

A versão não bloqueável de *send* é utilizada na tarefa ao enviar dados para uma fila de transmissão, para que ela não entre em estado de bloqueio caso a fila de alguma interface esteja sem espaço.

Cada interface tem sua tarefa de recepção, fila de recepção e fila de transmissão. A tarefa de recepção fica bloqueada na fila de recepção até que algum dado seja recebido e enfileirado pelo *handler* da interrupção. Ao desbloquear da fila a tarefa passa a enfileirar os dados nas filas de transmissão das outras interfaces, sempre verificando se há alguma transmissão em andamento para que inicie a transmissão quando necessário. Ao final de uma transmissão a interrupção retira um dado da fila e, se ela não estiver vazia, o transmite.

Como citado anteriormente, é possível configurar prioridades para as tarefas, de forma que a tarefa de uma interface de maior importância tenha prioridade sobre as outras, evitando que esta precise aguardar outras tarefas realizarem seu processamento. Isso é importante para

as interfaces mais rápidas, diminuindo o tempo gasto para início de transmissão e evitando que o barramento fique desocupado sem necessidade.

5.2 IMPLEMENTAÇÃO DAS INTERFACES RS-232 E RS-485

Os periféricos UART, que realizam a transmissão nas interfaces RS-232 e RS-485, foram os primeiros a ser implementados, pois sua transmissão é mais simples e é possível realizar testes de comunicação com o computador a partir de um *hardware* USB-Serial adequado para cada interface.

Para a implementação do tratamento de interrupções (*handler*) e da transmissão foram utilizados um semáforo e um *mutex* (recursos do sistema operacional FreeRTOS). O semáforo é utilizado na transmissão para sincronizar a tarefa e a interrupção de fim de transmissão. Já o *mutex* é utilizado para garantir exclusão mútua das tarefas na função de transmissão.

A função de recepção é definida pela aplicação e será descrita a seguir.

5.2.1 Configuração

É desejável que a interface possa ser reconfigurada em tempo de execução. Por esse motivo foi implementada uma função de configuração que recebe parâmetros genéricos, configurando os registradores do periférico a partir deles.

Para configurar a UART a aplicação precisa informar:

- A taxa de dados;
- A configuração da linha (bits por dado, tipo de paridade e número de bits de parada);
- A função de tratamento de erros (chamada quando ocorre algum erro de recepção); e
- A função de recepção de dados (chamada quando um ou mais dados são recebidos).

Isso permite uma flexibilidade muito grande para a aplicação, permitindo que diversas configurações de linha, taxas de dados e protocolos de comunicação possam ser utilizados com uma mesma implementação da UART, sendo possível inclusive alterar as configurações em tempo de execução desativando a interface e chamando a função de configuração novamente, porém com os novos parâmetros.

Também é preciso informar as funções para configuração do *transceiver* (que depende da interface utilizada):

- Uma função para configurar a direção das portas (chamada durante a configuração);
- Uma função para habilitar a transmissão (chamada logo antes do início da transmissão); e
- Uma função para habilitar a recepção (chamada logo após o fim da transmissão).

Estas funções também possibilitam uma maior flexibilidade, permitindo que a mesma função de transmissão na UART seja utilizada tanto para RS-232 quanto para RS-485.

5.2.1.1 Configuração do *Transceiver* RS-232

A interface RS-232 é *full-duplex* e ponto-a-ponto (apenas dois nodos), permitindo que tanto o *driver* (transmissor) quanto o *receiver* (receptor) fiquem ativos o tempo todo.

Dessa forma, a configuração do *transceiver* da interface RS-232 só precisa ser realizada na configuração da interface. As portas do microcontrolador que controlam o estado do *transceiver* devem ser configuradas como saída e nos níveis adequados para o funcionamento dele.

Ou seja, apenas a primeira função é necessária (configurar a direção das portas), podendo definir as outras duas (habilitar a transmissão e habilitar a recepção) com o corpo vazio.

5.2.1.2 Configuração do *Transceiver* RS-485

A interface RS-485 disponível é *half-duplex*, impedindo que mais de um nodo habilite seu *driver* sem o risco de corromper dados sendo transmitidos. Isso exige que as funções habilitar a transmissão e habilitar a recepção também sejam definidas.

Na configuração da interface as portas do microcontrolador que controlam o estado do *transceiver* devem ser configuradas como saída e com os níveis de tensão para habilitar o estado de recepção (configurar a direção das portas). Logo antes da transmissão deve-se configurar o *transceiver* para o estado de transmissão, habilitando o *driver* (habilitar a transmissão) e logo após a transmissão deve-se configurar o *transceiver* para o estado de recepção, desabilitando o *driver* (habilitar a recepção).

O *receiver* pode ser desabilitado no estado de transmissão; nesse caso é preciso reabilitar no estado de recepção. No entanto, manter o *receiver* habilitado permite o monitoramento do barramento e verificação dos dados transmitidos, possibilitando retransmissão da mensagem inteira em caso de erro.

5.2.2 Recepção

A recepção dos dados é realizada através da interrupção RDA (*Receive Data Available)*. A ocorrência desta interrupção indica que há ao menos um byte no *buffer* de recepção.

O tratamento desta interrupção consta em ler os dados do *buffer* até que este esteja vazio ou que um limite de dados tenha sido lido e chamar a função de recepção, informando o número de bytes recebidos e um ponteiro para os dados.

O protótipo da função de recepção utilizado é

```
void UART_Receive(uint8_t *Buff, int Size, int *HiPrioTaskWoken);
```

onde o primeiro argumento é um ponteiro para os dados lidos do *buffer* de recepção, o segundo é o número de bytes lidos e o terceiro aponta para uma variável que indica se é necessário uma troca de contexto.

Como é chamada em contexto de interrupção, é essencial que a função de recepção não realize processamentos longos, para reduzir ao máximo atrasos no serviço de outras interrupções. O ideal é enviar os dados recebidos para uma fila e deixar que uma tarefa faça o processamento fora do contexto de interrupção.

O algoritmo 5.5 mostra de forma simplificada a implementação da função de tratamento de interrupção da UART, utilizando as ferramentas do sistema operacional, como filas e semáforos. A variável *HiPrioTaskWoken* é utilizada por funções que podem acordar tarefas para verificar se uma troca de tarefa é necessária ao fim da interrupção. Na linha 6 temos o tratamento de interrupção de recepção de dados, onde até 16 bytes são removidos do *buffer* de recepção (fornecido pelo *hardware* do periférico UART), chamando então a função *UART_Receive*, que envia os dados recebidos para a fila de recepção da interface, onde a tarefa da interface aguarda os dados recebidos. O tratamento de fim de transmissão é realizado na linha 26, desbloqueando o semáforo onde a tarefa de transmissão da interface aguarda, sinalizando que mais dados podem ser transmitidos.

5.2.3 Transmissão

A transmissão dos dados é realizada a partir do funcionamento conjunto da interrupção *THRE* (*Transmit Holding Register Empty*) e da função de transmissão.

O protótipo da da função de transmissão é

```
void UART_Send(uint8_t *Buff, int Num);
```

e os parâmetros são um ponteiro para o *buffer* de dados e a quantidade de dados a ser transmitida.

A função de transmissão inicia travando o *mutex* de transmissão, evitando que múltiplas tarefas acessem o registrador de transmissão, então chama a função para habilitar transmissão, transmite os dados (veja abaixo), chama a função para habilitar recepção e desbloqueia o *mutex* obtido no início da chamada.

A transmissão é realizada escrevendo no registrador de transmissão até preencher o *buffer* FIFO (tamanho 16), ou acabarem os dados, e bloqueia ao tentar travar no semáforo de

Algoritmo 5.5 Código para tratamento de interrupções da UART.

```
1  void UART_Handler(void)
2  {
3    /* Inicializa */
4    int HiPrioTaskWoken = 0;
5
6    if(UART_INTERRUPT_RDA())
7    {
8      /* Dados recebidos */
9
10     int Size = 0;
11     uint8_t Buff[16];
12
13     while(UART_DATA_ON_RXBUFF())
14     {
15       Buff[Size] = UART_RXDATA;
16       Size++;
17
18       /* Limite de dados */
19       if(Size >= 16)
20         break;
21     }
22
23     UART_Receive(Buff, Size, &HiPrioTaskWoken);
24   }
25
26   else if(UART_INTERRUPT_THRE())
27   {
28     /* Fim de transmissão. Destrava o semáforo. */
29     xSemaphoreGiveFromISR(SemaforoTX_UART, &HiPrioTaskWoken);
30   }
31
32   /* Realiza troca de contexto se necessário */
33   if(HiPrioTaskWoken == 1)
34     portYIELD_FROM_ISR();
35 }
```

sincronização. Este semáforo é destravado quando ocorre a interrupção THRE (veja o algoritmo 5.5). Então a tarefa pode enviar o resto dos dados ou sair se já tiver enviado todos.

A função de transmissão deve ser chamada apenas em contexto de tarefa, pois ela pode bloquear em um *mutex* ou em um semáforo.

O algoritmo 5.6 mostra o código da função de transmissão da UART que deve ser chamada pelas tarefas. Na linha 4 tenta-se travar um *mutex*, que protege os registradores de transmissão contra acessos simultâneos, aguardando até que o *mutex* seja destravado por outra tarefa que possa estar utilizando ele. Este semáforo é destravado pela função na linha 20. Nas linhas 5 e 19 são chamadas as funções que habilita e desabilita o *driver* do *transceiver* da interface. Nas linhas 10 ou 14 é chamada a função de transmissão de dados na UART. São chamas funções diferentes para as interfaces RS-232 e RS-485 (estas funções estão definidas a seguir). RS-485 necessita que o último byte transmitido cause uma interrupção de fim de transmissão atrasada, enquanto para RS-232 isso não é necessário. Este comportamento é definido

em tempo de compilação.

Algoritmo 5.6 Código para transmissão UART.

```
1  void UART_Send(uint8_t *Buff, int Num)
2  {
3    /* Tenta travar mutex e configura hardware para transmissão */
4    xSemaphoreTake(MutexTX_UART, portMAX_DELAY);
5    UART_Config_TX();
6
7    /* Transmite na UART */
8    #if (HAB_RS232 == 1)
9    {
10     UART_Send_232(Buff, Num);
11   }
12   #else
13   {
14     UART_Send_485(Buff, Num);
15   }
16   #endif
17
18   /* Configura hardware para recepção e destrava mutex */
19   UART_Config_RX();
20   xSemaphoreGive(MutexTX_UART);
21 }
```

5.2.3.1 Interrupção THRE

De acordo com o *datasheet* do microcontrolador utilizado, a interrupção THRE apresenta dois comportamentos diferentes dependendo de como é realizada a transmissão dos dados: normal e atrasada.

A interrupção THRE normal ocorre assim que o *buffer* de transmissão estiver completamente vazio, ou seja, ocorre assim que é possível escrever no registrador de transmissão 16 vezes (tamanho da fila de transmissão do *hardware*) sem que haja dados descartados ou sobrescritos. Já a interrupção atrasada aguarda o tempo extra da transmissão de um byte. Para que ocorra uma interrupção atrasada é preciso que apenas um byte seja escrito no *buffer* de transmissão desde a última interrupção THRE (NXP SEMICONDUCTORS, 2012).

Essa interrupção atrasada é implementada em *hardware* para que haja apenas uma linha de interrupção ocupada pela recepção do periférico, ao invés de duas. Seu uso é necessário em interfaces que precisam ativar e desativar o *driver*[4] do *transceiver*, que deve ocorrer apenas depois do final da transmissão, como é o caso da interface RS-485.

[4] Ativar e desativar a escrita no barramento.

5.2.3.2 Transmissão RS-232

A transmissão para a interface RS-232 é bastante simples, pois não é necessário garantir que haja interrupções atrasadas (tanto o *driver* quanto o *receiver* ficam ativos o tempo todo). No entanto é interessante evitar que ocorram interrupções atrasadas para permitir maior taxa de dados e aproveitamento da interface.

O *buffer* FIFO de transmissão possui tamanho 16. Portanto é possível enviar até 16 bytes de dados para cada bloqueio no semáforo. Quando o último envio for apenas um byte ocorre uma interrupção atrasada.

Portanto se houver 17 bytes restantes para transmitir devemos transmitir apenas 15, para que na próxima transmissão possamos transmitir 2 bytes, evitando a interrupção com atraso. Dessa forma só haverá atraso na interrupção quando uma tarefa enviar apenas um byte. O algoritmo 5.7 mostra a implementação. Na linha 5 temos o laço principal da função, que repete seu bloco até que todos os dados sejam transmitidos. O envio dos dados par o *buffer* de transmissão ocorre entre as linhas 10 e 21. É verificado se o número de dados restantes é igual a 17, realizando o envio de 15 bytes nesse caso, sobrando 2 para o última transmissão. Na linha 9 as interrupções são desabilitadas para que o *buffer* de transmissão seja preenchido atomicamente, habilitando-as novamente na linha 22. Na linha 25 a função trava no semáforo de transmissão, aguardando a interrupção de fim de transmissão, continuando a transmissão caso ainda haja dados a ser transmitidos.

5.2.3.3 Transmissão RS-485

Na interface RS-485 é preciso garantir que o último byte seja enviado sozinho, de forma que haja uma interrupção atrasada.

Exceto o último byte, os dados podem ser transmitidos da mesma forma que na interface RS-232, para que haja um melhor aproveitamento do barramento. Para isso, subtrai-se um do número de bytes antes de iniciar o envio e depois que finalizar envia-se o último byte sozinho, bloqueando no semáforo de sincronização novamente para aguardar a interrupção atrasada. Caso uma tarefa precise enviar dois bytes serão geradas duas interrupções com atraso ao invés de apenas uma. O algoritmo 5.8 mostra a implementação. Na linha 6 a função descrita acima é chamada para realizar a transmissão dos dados, exceto o último byte, aguardando a interrupção de fim de transmissão. Na linha 9 é transmitido o último byte sozinho, fazendo com que ocorra a interrupção atrasada, então na linha 10 a tarefa trava no semáforo de fim de transmissão, aguardando que a interrupção ocorra.

Algoritmo 5.7 Código para transmissão UART na interface RS-232.

```
1  void UART_Send_232(uint8_t *Buff, int Num)
2  {
3    int i;
4
5    while(Num > 0)
6    {
7      /* Transmite os bytes */
8
9      portENTER_CRITICAL(); /* Desabilita interrupções. */
10     if(Num != 17)
11     {
12       for(i = 0; i < Num && i < 16; ++i)
13         UART_TXDATA = *Buff++;
14       Num -= 16;
15     }
16     else
17     {
18       for(i = 0; i < 15; ++i)
19         UART_TXDATA = *Buff++;
20       Num = 2;
21     }
22     portEXIT_CRITICAL(); /* Habilita interrupções. */
23
24     /* Aguarda interrupção THRE */
25     xSemaphoreTake(SemaforoTX_UART, portMAX_DELAY);
26   }
27 }
```

Algoritmo 5.8 Código para transmissão UART na interface RS-485.

```
1  void UART_Send_485(uint8_t *Buff, int Num)
2  {
3    Num -= 1;
4
5    /* Envia todos os bytes exceto o último */
6    UART_Send_232(Buff, Num);
7
8    /* Transmite o último byte e aguarda interrupção THRE */
9    UART_TXDATA = Buff[Num];
10   xSemaphoreTake(SemaforoTX_UART, portMAX_DELAY);
11 }
```

5.3 IMPLEMENTAÇÃO DA INTERFACE CAN

O periférico CAN foi o segundo passo na implementação da Ponte de Interfaces, podendo utilizar as interfaces RS-232 ou RS-485 para comunicar com o computador (já funcionando como uma ponte CAN–RS-232 e CAN–RS-485).

Para a implementação do tratamento de interrupções (*handler*) foi utilizado um semá-

foro contador e um *mutex*. O semáforo contador é utilizado para sincronizar as tarefas com a liberação dos três *buffers* de transmissão disponíveis. O *mutex* é utilizado para garantir exclusão mútua das tarefas na função de transmissão (no acesso aos *buffers* de transmissão).

A função de recepção é definida pela aplicação e será descrita a seguir.

5.3.1 Configuração

Para que a interface possa ser reconfigurada em tempo de execução foi implementada uma função de configuração que recebe parâmetros genéricos, configurando os registradores do periférico a partir deles.

Para configurar a CAN a aplicação precisa informar:

- A taxa de dados;
- A função de tratamento de erros (chamada quando ocorre algum erro);
- A função de recepção de mensagens (chamada quando mensagens são recebidas); e
- Uma função para configurar o *transceiver* (chamada durante a configuração).

5.3.1.1 Configuração do *Transceiver* CAN

A configuração do *transceiver* CAN é bastante simples, necessitando manter o pino $\overline{SLEEP}$ em nível alto, para que mantenha-se no estado ativo (fora do modo de baixo consumo). Para isso basta apenas configurar a direção da porta do microcontrolador como saída e com nível alto.

5.3.2 Recepção

A recepção dos dados é realizada através da interrupção RI (*Receive Interrupt)*. Ela indica que o controlador CAN recebeu com sucesso uma mensagem através do barramento. O tratamento desta interrupção consta em ler os dados do *buffer* de recepção.

A definição da estrutura que contém a mensagem é

```
struct CAN_Msg {
  uint32_t Frame;
  uint32_t MsgID;
  uint32_t DataA;
  uint32_t DataB;
};
```

onde *Frame* indica o tipo da mensagem e o tamanho dos dados, *MsgID* indica o identificador da mensagem e *DataA* e *DataB* contém os dados da mensagem.

O protótipo da função de recepção utilizado é

```
void CAN_Receive(struct CAN_Msg *Msg, int *HiPrioTaskWoken);
```

onde o primeiro argumento é um ponteiro para a mensagem recebida e o segundo aponta para uma variável que indica se é necessário uma troca de contexto (recomendado pela documentação do FreeRTOS; deve ter seu valor modificado para 1 caso uma tarefa de maior prioridade tenha sido desbloqueada com a ação de recepção).

Assim como no caso da UART, esta função é chamada em contexto de interrupção; portanto é essencial que a função de recepção não realize processamentos longos, para reduzir ao máximo atrasos no serviço de outras interrupções. O ideal é enviar os dados recebidos para uma fila e deixar que uma tarefa faça o processamento fora do contexto de interrupção.

O algoritmo 5.9 mostra de forma simplificada a implementação da função de tratamento de interrupção da CAN, utilizando as ferramentas do sistema operacional. A variável *HiPrioTaskWoken* possui a mesma finalidade do tratamento da interrupção da UART. Na linha 6 temos o tratamento de interrupção de recepção, onde a mensagem recebida pela interface é lida (linhas 12 a 15) e chama então a função *CAN_Receive*, que envia a mensagem recebida para a fila de recepção da interface. O tratamento de fim de transmissão é realizado na linha 20, verificando qual dos três *buffers* de transmissão finalizou sua transmissão, desbloqueando o semáforo onde a tarefa de transmissão da interface aguarda, sinalizando que mais mensagens podem ser transmitidas, podendo desbloquear o semáforo mais de uma vez, caso mais de um *buffer* for liberado desde a última interrupção. Este semáforo utilizado para transmissão da CAN é um semáforo contador com limite máximo em 3, pois o *hardware* possui três *buffers* de transmissão de mensagens CAN. Veja a função de transmissão para mais detalhes sobre a implementação.

Algoritmo 5.9 Código para tratamento de interrupções da CAN.

```
1  void CAN_Handler(void)
2  {
3    /* Inicializa */
4    int HiPrioTaskWoken = 0;
5
6    if(CAN_INTERRUPT_RI())
7    {
8      /* Dados recebidos */
9
10     struct CAN_Msg Msg;
11
12     Msg.Frame = CAN1RFS;
13     Msg.MsgID = CAN1RID;
14     Msg.DataA = CAN1RDA;
15     Msg.DataB = CAN1RDB;
16
17     CAN_Receive(&Msg, &HiPrioTaskWoken);
18   }
19
20   else if(CAN_INTERRUPT_TI())
21   {
22     /* Fim de transmissão */
23
24     /* Verifica qual buffer foi liberado e destrava o semáforo. */
25     if(CAN_FREE_BUFFER1())
26       xSemaphoreGiveFromISR(SemaforoTX_CAN, &HiPrioTaskWoken);
27     if(CAN_FREE_BUFFER2())
28       xSemaphoreGiveFromISR(SemaforoTX_CAN, &HiPrioTaskWoken);
29     if(CAN_FREE_BUFFER3())
30       xSemaphoreGiveFromISR(SemaforoTX_CAN, &HiPrioTaskWoken);
31   }
32
33   /* Realiza troca de contexto se necessário */
34   if(HiPrioTaskWoken == 1)
35     portYIELD_FROM_ISR();
36 }
```

5.3.3 Transmissão

A transmissão dos dados é realizada a partir do funcionamento conjunto da interrupção TI (*Transmit Interrupt*) e da função de transmissão.

O protótipo da da função de transmissão é

```
void CAN_Send(struct CAN_Msg *Msg);
```

cujo parâmetro é um ponteiro para a mensagem que deve ser transmitida.

A função de transmissão deve travar o semáforo de sincronização, evitando que entre na região crítica sem que haja um *buffer* de transmissão livre. Então trava o *mutex* de transmissão, evitando que múltiplas tarefas acessem os registradores de transmissão simultaneamente

(para o caso de mais de um *buffer* de transmissão estar livre). Então verifica qual *buffer* está livre e transmite a mensagem escrevendo-a nos registradores de transmissão e escrevendo no bit correspondente aquele *buffer* para iniciar a transmissão da mensagem. Depois destrava o *mutex* de transmissão para que outra tarefa possa transmitir.

Ao fim da transmissão, realizada pelo *hardware*, ocorre interrupção de transmissão, cujo tratamento consiste em verificar qual *buffer* foi liberado e destravar o semáforo de sincronização, permitindo que uma tarefa bloqueada possa entrar na região crítica.

O algoritmo 5.10 mostra o código da função de transmissão da CAN que deve ser chamada pelas tarefas. Na linha 4 a função tenta travar o semáforo de transmissão. Este é um semáforo contador com limite de contagem em 3, de forma que possa ser utilizado para gerenciar o *buffer* de transmissão triplo fornecido pelo *hardware*. Caso não houver nenhum *buffer* de transmissão livre o semáforo bloqueia a tarefa até que uma interrupção de fim de transmissão ocorra, desbloqueando o semáforo. Na linha 7 a função tenta travar o *mutex* de transmissão. A função deste *mutex* garantir que não há acessos simultâneos aos registradores de transmissão, o que poderia causar inconsistências. Este *mutex* é destravado ao fim da função, na linha 36. Nas linhas 10 a 33 a função verifica qual dos três *buffers* de transmissão está livre, escreve os dados da mensagem nos registradores correspondentes e inicia a transmissão.

Algoritmo 5.10 Código para transmissão CAN.

```
1  void CAN_Send(struct CAN_Msg *Msg)
2  {
3    /* Trava aqui se não houver buffer disponível */
4    xSemaphoreTake(SemaforoTX_CAN, portMAX_DELAY);
5
6    /* Obtém mutex */
7    xSemaphoreTake(MutexTX_CAN, portMAX_DELAY);
8
9    /* Verifica qual buffer está livre e transmite */
10   if(CAN_FREE_BUFFER1())
11   {
12     CAN_FRAME1 = Msg->Frame & 0xC00F00FF;
13     CAN_MSGID1 = Msg->MsgID;
14     CAN_DATAA1 = Msg->DataA;
15     CAN_DATAB1 = Msg->DataB;
16     CAN_TRANSMIT_REGISTER = CAN_TRANSMIT | CAN_BUFFER1;
17   }
18   else if(CAN_FREE_BUFFER2())
19   {
20     CAN_FRAME2 = Msg->Frame & 0xC00F00FF;
21     CAN_MSGID2 = Msg->MsgID;
22     CAN_DATAA2 = Msg->DataA;
23     CAN_DATAB2 = Msg->DataB;
24     CAN_TRANSMIT_REGISTER = CAN_TRANSMIT | CAN_BUFFER2;
25   }
26   else if(CAN_FREE_BUFFER3())
27   {
28     CAN_FRAME3 = Msg->Frame & 0xC00F00FF;
29     CAN_MSGID3 = Msg->MsgID;
30     CAN_DATAA3 = Msg->DataA;
31     CAN_DATAB3 = Msg->DataB;
32     CAN_TRANSMIT_REGISTER = CAN_TRANSMIT | CAN_BUFFER3;
33   }
34
35   /* Devolve mutex */
36   xSemaphoreGive(MutexTX_CAN);
37 }
```

6 ORGANIZAÇÃO DAS TAREFAS

As interfaces de comunicação possuem duas tarefas para gerenciar o recebimento e envio de mensagens no barramento. A tarefa de recepção possui prioridade mais baixa, realizando processamento dos dados recebidos e convertendo para outras interfaces. A tarefa de transmissão possui alta prioridade e apenas realiza a transmissão, garantindo que a transmissão ocorra o mais rapidamente possível.

As mensagens enviadas para as tarefas são do tipo *Message* (mostrado abaixo), possuem três campos: tipo, tamanho e dados. O campo tipo (*Type*) armazena a origem da mensagem, o campo tamanho (*Size*) armazena o comprimento da estrutura armazenada no campo dados (máximo 16 bytes) e o campo dados (*Data*) armazena os dados da mensagem.

```
struct Message {
  uint16_t Type;
  uint16_t Size;
  uint8_t Data[16];
}
```

Definimos o tipo de dados *MessageType* (mostrado abaixo) para atribuir diferentes valores numéricos a cada tipo de mensagens que trabalhamos.

```
enum MessageType {
  MessageType_CAN   = 0,
  MessageType_UART  = 1
}
```

6.1 INTERFACE CAN

O gerenciamento da CAN é realizado por quatro funções e duas filas de mensagens: inicialização, recepção, tarefa de recepção e tarefa de transmissão. Os protótipos das funções são mostrados a seguir.

```
int TaskCAN1_Init(void);
void CAN1_Receive(struct CAN_Msg *Msg, int *HiPrioTaskWoken);
void TaskCAN1(void *Arg);
void TaskCAN1_TX(void *Arg);
```

A função de inicialização *TaskCAN1_Init*, omitida, deve ser chamada antes da inicialização do sistema operacional. Ela aloca as filas de recepção e de transmissão e configura a interface e interrupções.

A Interrupção de recepção de mensagens da CAN é configurada para chamar a função de recepção *CAN1_Receive*. Como esta função é chamada em contexto de interrupção, ela

apenas enfileira a mensagem recebida na fila de recepção da CAN para que ela seja processada em contexto de tarefa. O algoritmo 6.1 mostra a implementação da função de recepção. Nas linhas 6 e 7 a estrutura a ser enviada para a fila de recepção tem seu cabeçalho inicializado, indicando o tipo e o tamanho dos dados. Na linha 8 a mensagem CAN é copiada para o campo de dados e na linha 10 a estrutura é enviada, ou copiada, para a fila.

Algoritmo 6.1 Código da função recepção de mensagens CAN.

```
1  void CAN1_Receive(struct CAN_Msg *MsgCAN, int *HiPrioTaskWoken)
2  {
3    Message Msg;
4
5    /* Inicializa a estrutura e copia mensagem de MsgCAN para Msg.Data */
6    Msg.Type = MessageType_CAN;
7    Msg.Size = sizeof(struct CAN_Msg);
8    memcpy(Msg.Data, MsgCAN, sizeof(struct CAN_Msg));
9
10   xQueueSendFromISR(Queue_CAN1, &Msg, HiPrioTaskWoken);
11 }
```

A tarefa de recepção *TaskCAN1* fica em um laço infinito lendo as mensagens que chegam na fila de recepção. As mensagens são processadas, verificando se devem ser transmitidas nas outras interfaces, convertidas e enfileiradas nas filas de transmissão das interfaces configuradas. O algoritmo 6.2 mostra a implementação da tarefa de recepção. Na linha 5 temos o laço principal, um laço infinito que realiza o processamento das mensagens recebidas. Na linha 8 a tarefa aguarda a chegada de mensagens na fila de recepção. As mensagens são então filtradas (para evitar o envio de mensagens não desejadas nas outras interfaces) e enviadas para as interfaces de interesse, de acordo com a configuração.

A função *FiltroMsgCAN1*, linha 12, verifica o tipo e o conteúdo da mensagem e decide se ela deve ser enviada em outras interfaces ou se deve ser descartada. Esta função não deve ser chamada na função de recepção pois pode haver uma lista considerável de mensagens desejáveis ou indesejáveis, podendo causar um processamento longo durante a interrupção.

A função *ConverteMsgCAN_UART*, linha 20, converte mensagens CAN para o formato UART (veja a figura 3.5), de acordo com o protocolo de conversão citado na seção3.6.

Ainda sobre o algoritmo 6.2, note que, na linha 21, o terceiro argumento na chamada de *xQueueSend* é zero, forçando que a chamada retorne imediatamente caso não houver espaço na fila de destino, de forma que a mensagem não seja enviada naquela interface. Isso é importante para que a tarefa de recepção não fique bloqueada, o que impediria que as mensagens recebidas sejam enviadas nas outras interfaces.

Algoritmo 6.2 Código da tarefa de recepção de mensagens CAN.

```
1  void TaskCAN1(void *Arg)
2  {
3    Message Msg;
4
5    while(1)
6    {
7      /* Aguarda uma mensagem na fila */
8      xQueueReceive(Queue_CAN1, &Msg, portMAX_DELAY);
9
10     /* Valida e verifica se deve enviar a mensagem em outras
11        interfaces */
12     if(FiltroMsgCAN1(&Msg) == 0)
13       continue;
14
15     /* Insere a mensagem recebida na fila de transmissão da
16        interface RS-232 */
17     if (Config.ENVIAR_CAN1_RS232 == 1)
18     {
19       Message MsgUART;
20       ConverteMsgCAN_UART(&MsgUART, &Msg);
21       xQueueSend(Queue_UART0_TX, &MsgUART, 0);
22     }
23
24     /* Outras interfaces ... */
25   }
26 }
```

A tarefa de transmissão *TaskCAN1_TX* também consiste em ler uma fila em um laço infinito. As mensagens que chegam na fila de transmissão são transmitidas na interface CAN por esta tarefa. O algoritmo 6.3 mostra a implementação da tarefa de transmissão. Na linha 5 temos o laço principal, um laço infinito que realiza a transmissão das mensagens recebidas através da fila. Na linha 8 a tarefa aguarda a chegada de mensagens na fila de transmissão, enviando-as na interface com a chamada da função *CAN1_Send* na linha 11.

Algoritmo 6.3 Código da tarefa de transmissão de mensagens CAN.

```
1  void TaskCAN1_TX(void *Arg)
2  {
3    Message Msg;
4
5    while(1)
6    {
7      /* Aguarda uma mensagem na fila */
8      xQueueReceive(Queue_CAN1_TX, &Msg, portMAX_DELAY);
9
10     /* Transmite a mensagem na interface CAN */
11     CAN1_Send(Msg.Data);
12   }
13 }
```

6.2 INTERFACE RS-232

Assim como na CAN, o gerenciamento da RS-232 é realizado por quatro funções e duas filas de mensagens: inicialização, recepção, tarefa de recepção e tarefa de transmissão. Os protótipos das funções são mostrados a seguir.

```
int TaskRS232_Init(void);
void RS232_Receive(uint8_t *Buff, int Size, int *HiPrioTaskWoken);
void TaskRS232(void *Arg);
void TaskRS232_TX(void *Arg);
```

A função de inicialização *TaskRS232_Init*, omitida, deve ser chamada antes da inicialização do sistema operacional. Ela possui as mesmas funções que sua equivalente da CAN: alocar as filas de recepção e de transmissão e configurar a interface e interrupções.

A Interrupção de recepção de mensagens da RS-232 é configurada para chamar a função de recepção *RS232_Receive*. Como esta função é chamada em contexto de interrupção, ela apenas enfileira a mensagem recebida na fila de recepção da RS-232 para que ela seja processada em contexto de tarefa. O algoritmo 6.4 mostra a implementação da função de recepção. Nas linhas 5 e 6 a estrutura a ser enviada para a fila de recepção tem seu cabeçalho inicializado, indicando o tipo e o tamanho dos dados. Na linha 7 os dados recebidos da interface UART são copiados para o campo de dados e na linha 9 a estrutura é enviada, ou copiada, para a fila de recepção da interface.

Algoritmo 6.4 Código da função recepção de mensagens RS-232.

```
1  void RS232_Receive(uint8_t *Buff, int Size, int *HiPrioTaskWoken)
2  {
3    Message Msg;
4
5    Msg.Type = MessageType_UART;
6    Msg.Size = Size;
7    memcpy(Msg.Data, Buff, Size);
8
9    xQueueSendFromISR(Queue_RS232, &Msg, HiPrioTaskWoken);
10 }
```

A tarefa de recepção *TaskRS232* fica em um laço infinito lendo as mensagens que chegam na fila de recepção. As mensagens são processadas, verificando se devem ser transmitidas nas outras interfaces, convertidas e enfileiradas nas filas de transmissão das interfaces configuradas. O algoritmo 6.5 mostra a implementação da tarefa de recepção.

Algoritmo 6.5 Código da tarefa de recepção de mensagens RS-232.

```
1  void TaskRS232(void *Arg)
2  {
3    Message Msg;
4
5    /* Buffer de dados para verificação e conversão dos dados
6       recebidos em mensagens CAN */
7    static uint8_t Buff[32];
8    static int Size = 0;
9    Message MsgCAN;
10
11   while(1)
12   {
13     /* Aguarda por dados na fila e copia no buffer */
14     xQueueReceive(Queue_RS232, &Msg, portMAX_DELAY);
15
16     memcpy(&Buff[Size], Msg.Data, Msg.Size);
17     Size += Msg.Size;
18
19     if(ConverteMsgUART_CAN(&MsgCAN, Buff, &Size))
20     {
21       if (Config.ENVIAR_RS232_CAN1 == 1)
22       {
23         /* Verifica se deve enviar a mensagem na
24            interface CAN */
25         if(FiltroMsgRS232_CAN1(&Msg) == 1)
26         {
27           /* Insere a mensagem recebida na fila de
28              transmissão da interface CAN */
29           xQueueSend(Queue_CAN1_TX, &MsgCAN, 0);
30         }
31       }
32
33       if (Config.ENVIAR_RS232_RS485_MENSAGEM == 1)
34       {
35         /* Envia mensagem completa para RS-485 transmitir */
36         Message MsgUART;
37         ConverteMsgCAN_UART(&MsgUART, &MsgCAN);
38         xQueueSend(Queue_RS485_TX, &MsgUART, 0);
39       }
40     }
41
42     if (Config.ENVIAR_RS232_RS485_BYTEABYTE == 1)
43     {
44       /* Envia dados brutos recebidos para RS-485 transmitir */
45       xQueueSend(Queue_RS485_TX, &Msg, 0);
46     }
47
48     /* Outras interfaces ... */
49   }
50 }
```

Nas linhas 7 e 8 são declaradas as variáveis *Buff* e *Size*, que armazenam os dados recebidos pela interface e precisam ser mantidos estáticos para que possam ser verificados novamente no futuro. O laço principal inicia na linha 11, aguarda a recepção de dados na fila

de recepção (linha 14) e os adiciona no *buffer* de dados local (linhas 16 e 17). Em seguida, na linha 19, é verificado se os dados do *buffer* local consistem em uma mensagem válida, de acordo com o protocolo discutido na seção 3.6 (veja a figura 3.5) convertendo a mensagem para o formato CAN. Essa função também limpa os dados caso receba uma mensagem completa ou não atendam ao protocolo (ex: tamanho ou CRC inválido). As mensagens válidas são filtradas com a chamada da função *FiltroMsgRS232_CAN1* na linha 25, que verifica o tipo e o conteúdo da mensagem e decide se ela deve ser enviada em outras interfaces ou se deve ser descartada. A mensagem convertida é então enviada para a fila de transmissão das interfaces configuradas (linha 29 para CAN e linha 38 para RS-485). A transmissão de dados da RS-232 para RS-485 pode ocorrer de duas maneiras: por mensagem ou por byte. Quando configurada por mensagem, a transmissão de uma interface para outra só ocorre quando a mensagem segue o protocolo, enquanto que no modo byte a byte todos os dados são transmitidos assim que recebidos. A segunda forma é útil para tarefas de ambientes de desenvolvimento, como detectar problemas de transmissão em outros módulos, enquanto a primeira é mais adequada para ambientes de produção.

A tarefa de transmissão *TaskRS232_TX* também consiste em ler uma fila em um laço infinito. As mensagens que chegam na fila de transmissão são transmitidas na interface RS-232 por esta tarefa. O algoritmo 6.6 mostra a implementação da tarefa de transmissão. Na linha 5 temos o laço principal, um laço infinito que realiza a transmissão dos dados recebidos através da fila. Na linha 8 a tarefa aguarda a chegada de dados na fila de transmissão, enviando-as na interface com a chamada da função *RS232_Send* na linha 11.

Algoritmo 6.6 Código da tarefa de transmissão de mensagens RS-232.

```
1  void TaskRS232_TX(void *Arg)
2  {
3    Message Msg;
4
5    while(1)
6    {
7      /* Aguarda uma mensagem na fila */
8      xQueueReceive(Queue_RS232_TX, &Msg, portMAX_DELAY);
9
10     /* Transmite a mensagem na interface RS-232 */
11     RS232_Send(&Msg.Data, Msg.Size);
12   }
13 }
```

6.3 INTERFACE RS-485

O gerenciamento da RS-485 é semelhante a RS-232, sendo realizado por quatro funções e duas filas de mensagens: inicialização, recepção, tarefa de recepção e tarefa de transmissão. Ainda há o adicional de duas variáveis para o controle de tempo de espera da transmissão.

Os protótipos das funções e as declarações das variáveis são mostrados a seguir.

```
int TaskRS485_Init(void);
void RS485_Receive(uint8_t *Buff, int Size, int *HiPrioTaskWoken);
void TaskRS485(void *Arg);
void TaskRS485_TX(void *Arg);

/* Tempos em ticks do sistema operacional */
const uint32_t TempoEsperaTX = 5;
uint32_t TempoUltimoRX = 0;
```

Exemplos e descrição da função de inicialização *TaskRS485_Init* e da tarefa de recepção *TaskRS485* são idênticas à interface RS-232 e foram omitidas.

A Interrupção de recepção de mensagens da RS-485 é configurada para chamar a função de recepção *RS485_Receive*. Como esta função é chamada em contexto de interrupção, ela apenas enfileira a mensagem recebida na fila de recepção da RS-485 para que ela seja processada em contexto de tarefa. Esta função ainda atualiza a variável *TempoUltimoRX*, que identifica o último instante em que um dado foi recebido, permitindo que seja utilizado pela transmissão. O algoritmo 6.7 mostra a implementação da função de recepção, que é idêntico à *RS232_Receive*, exceto por salvar o horário de recepção do último dado na interface, na linha 12. Salvar o horário permite que a tarefa de transmissão aguarde um tempo pré-determinado após o último dado recebido, de forma que não ocorram colisões na transmissão.

Algoritmo 6.7 Código da função recepção de mensagens RS-485.

```
1  void RS485_Receive(uint8_t *Buff, int Size, int *HiPrioTaskWoken)
2  {
3    Message Msg;
4
5    Msg.Type = MessageType_UART;
6    Msg.Size = Size;
7    memcpy(Msg.Data, Buff, Size);
8
9    xQueueSendFromISR(Queue_RS485, &Msg, HiPrioTaskWoken);
10
11   /* Salva o horário em que recebeu a mensagem */
12   TempoUltimoRX = xTaskGetTickCount();
13 }
```

A tarefa de transmissão *TaskRS485_TX* também consiste em ler uma fila em um laço infinito. As mensagens que chegam na fila de transmissão são transmitidas na interface RS-485 por esta tarefa. O algoritmo 6.8 mostra a implementação da tarefa de transmissão. A implementação é idêntica a tarefa de transmissão da RS-232, porém temos o adicional das linhas 11 e 12, que mantém a tarefa de transmissão da RS-485 suspensa pelo período *TempoEsperaTX* após o último dado ser recebido pela interface. Após o tempo de espera é realizada a transmissão dos dados recebidos através da fila de transmissão.

Algoritmo 6.8 Código da tarefa de transmissão de mensagens RS-485.

```
1  void TaskRS485_TX(void *Arg)
2  {
3    Message Msg;
4
5    while(1)
6    {
7      /* Aguarda uma mensagem na fila */
8      xQueueReceive(Queue_RS485_TX, &Msg, portMAX_DELAY);
9
10     /* Verifica o tempo desde o último dado recebido */
11     while((xTaskGetTickCount() - TempoUltimoRX) < TempoEsperaTX)
12       vTaskDelay(TempoEsperaTX);
13
14     /* Transmite a mensagem na interface RS-485 */
15     RS485_Send(&Msg.Data, Msg.Size);
16   }
17 }
```

7 RESULTADOS

Para validar a implementação da ponte foram realizados testes de conversão utilizando o *hardware* citado na seção 1.3. O microcontrolador utilizado no *hardware*, foi o LPC2366, operando em sua frequência máxima 72 MHz.

Os *transceivers* CAN, RS-485 e RS-232 utilizados foram MCP2551, SP485 e MAX3221, respectivamente. As taxas de transmissão da CAN, RS-485 e RS-232 foram respectivamente 250 kbps, 115,2 kbps e 115,2 kbps, sendo as duas últimas ambas configuradas sem paridade e com um bit de parada (*stop bit*).

O sistema operacional de tempo real utilizado foi o FreeRTOS v8.2.1. Foi utilizado o *kernel* preemptivo, configurado para gerar interrupções de *timer* a cada 1 milissegundo (mil *ticks* dos sistema operacional por segundo). Devido ao uso na implementação foram habilitados *mutex* e semáforos contadores.

7.1 CONVERSÃO RS-232 PARA RS-485

A figura 7.1 mostra uma captura de tela de osciloscópio da conversão da interface RS-232 para RS-485, sendo elas produtora e consumidora, respectivamente. Ambas as interfaces estavam configuradas a 115,2 kbps, sem paridade e com um bit de parada. É mostrado a conversão de dois bytes (0x00 e 0x7B) recebidos pela RS-232, convertidos e transmitidos pela RS-485. Em amarelo (acima na figura) vemos a tensão na linha RX da RS-232 e em azul (embaixo na figura) vemos a linha A da RS-485.

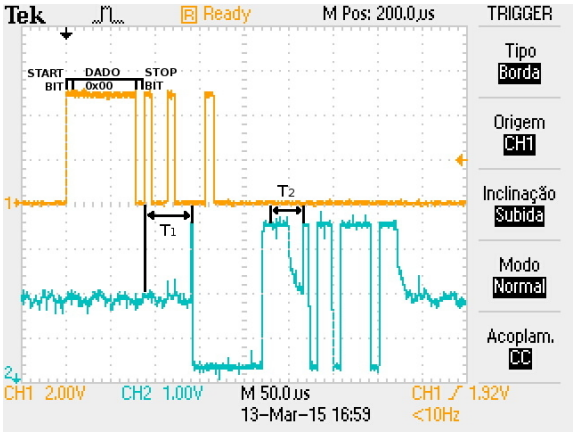

Figura 7.1: Conversão de dois bytes da interface RS-232 para RS-485.
Fonte: Autoria própria.

Pode-se notar duas coisas na figura. Primeiro que há um atraso T_1 de aproximadamente 50 μs entre o final do primeiro byte recebido na RS-232 e o início da transmissão na

RS-485, o que é normal e era esperado, devido as reações do *hardware* para a recepção e a transmissão não serem instantâneas e ao fato de ser realizada uma cópia dos dados de uma tarefa (recepção RS-232) para outra (transmissão RS-485). Nas análises consideramos o tempo de processamento como sendo instantâneo.

Segundo ponto é que há um tempo extra T_2 de aproximadamente 35 μs na transmissão da RS-485 entre o bit de parada do primeiro byte e o bit de início do segundo byte. Isso ocorre devido a uma limitação do *hardware* do microcontrolador, que gera interrupções atrasadas quando apenas um byte é enviado sozinho. Na interface RS-485 essa interrupção atrasada é gerada de propósito no fim de um bloco transmitido para que o *driver* do *transceiver* seja desabilitado somente após o fim da transmissão do último byte. Porém se o bloco possui apenas um byte o atraso é gerado a cada transmissão.

De acordo com o manual do usuário do microcontrolador este atraso T_2 é de cerca de 25 μs após o bit de parada, independente da taxa de dados utilizada. Assim, usando a mesma taxa de dados para ambas, a interface RS-232 será mais rápida que a RS-485 devido a necessidade desta interrupção atrasada. Comparando os tempos de transmissão para mensagens (blocos) de 16 bytes a 115.2 kbps obtemos T_{RS-232} = 1390 μs e T_{RS-485} = 1415 μs, com uma razão de aproximadamente 1,02. A razão aumenta para cerca de 1,29 se os blocos forem de apenas um byte, como é o caso mostrado na figura 7.1.

7.2 CONVERSÃO CAN PARA RS-232

A figura 7.2 mostra a captura de tela da conversão de 40 mensagens CAN para a interface RS-232. A interface CAN é a produtora, recebendo mensagens do barramento, e a RS-232 é consumidora, transmitindo as mensagens convertidas, estando a CAN configurada em 250 kbps e a interface RS-232 configurada em 115,2 kbps, sem paridade e um bit de parada. Em amarelo (em cima na figura), vemos a linha CANH da CAN e em azul (abaixo na figura) vemos a linha TX da RS-232.

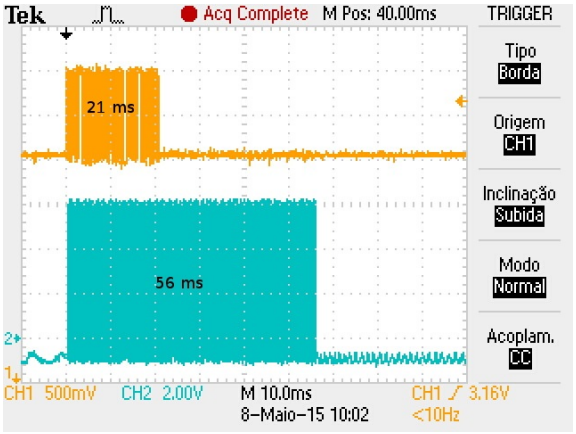

Figura 7.2: Conversão de 40 mensagens da interface CAN para RS-232.
Fonte: Autoria própria.

As mensagens enviadas neste teste são mostradas na seção 7.6. Vemos toda a conversão e praticamente não se percebe o tempo de atraso entre o início da recepção na CAN e o início da transmissão na RS-232. Note que a razão entre os tempos de recepção e transmissão é aproximadamente 2,6 (56/21), que é próximo do valor esperado de $T_{UART}/T_{CAN} = 2{,}76$.

Na figura 7.3, por sua vez, vemos a captura de tela da mesma sequência de mensagens sendo transmitida, porém aproximando 40 vezes na escala do tempo. Podemos perceber o atraso T_1 de aproximadamente 3,25 ms entre o início da recepção CAN e o início da transmissão na interface RS-232. Também observemos o atraso T_2 de aproximadamente 0,625 ms entre o fim da recepção da primeira mensagem CAN e o início da transmissão na RS-232. Esse intervalo T_2 é significativamente maior que na conversão entre RS-232 e RS-485 devido ao processamento necessário para converter a mensagem CAN para o protocolo UART.

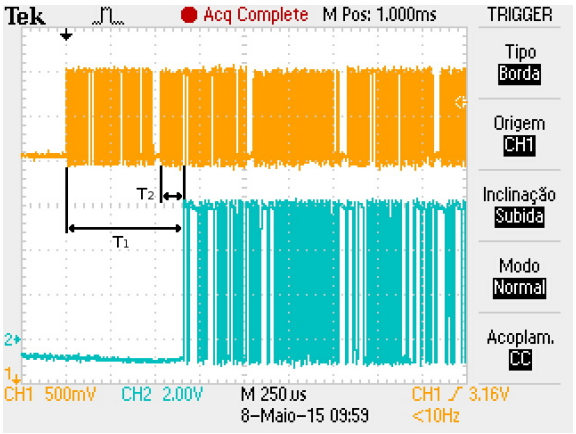

Figura 7.3: Conversão CAN – RS-232 aproximando no início da transmissão.
Fonte: Autoria própria.

Note que mesmo na captura aproximada não é possível identificar atraso algum entre duas mensagens consecutivas na RS-232, pois nesta interface o *driver* do *transceiver* está sempre habilitado, não havendo necessidade de forçar uma interrupção atrasada no fim do bloco

transmitido, como seria o caso da interface RS-485.

Na CAN, com a captura aproximada, é possível ver vários intervalos que aparentam ser atrasos na transmissão. Estes são sete bits com polaridade 1 que fazem parte do quadro CAN, marcando o fim da mensagem, não sendo um atraso nesse caso.

7.3 CONVERSÃO CAN PARA RS-485

A figura 7.4 mostra a captura de tela da conversão de 28 mensagens CAN para a interface RS-485. A interface CAN é a produtora, recebendo mensagens do seu barramento, e a RS-485 é consumidora, transmitindo as mensagens convertidas. A CAN estava configurada a 250 kbps e a RS-485 estava configurada em 115,2 kbps, sem paridade e um bit de parada. Acima, em amarelo, vemos a tensão na linha CANH da CAN e abaixo, em azul, vemos a linha A da RS-485.

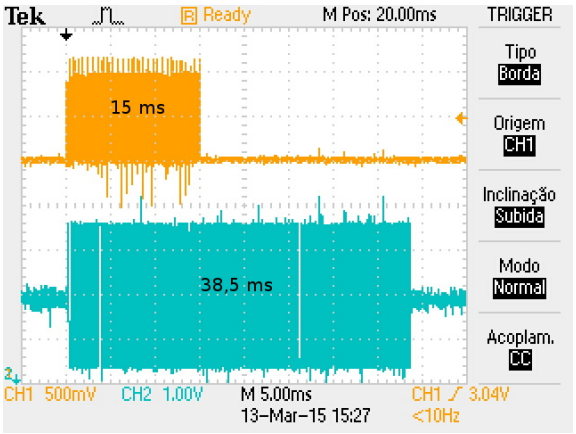

Figura 7.4: Conversão de 28 mensagens da interface CAN para RS-485.
Fonte: Autoria própria.

Devido a diferença na ordem de grandeza entre o tempo de transmissão e os atrasos, não é possível perceber o tempo entre os inícios das curvas, devido à transmissão da primeira mensagem na RS-485 ocorrer após o fim de sua recepção na CAN, nem o tempo causado pelas interrupções atrasadas no fim dos blocos.

Apesar desses atrasos existirem a razão entre os tempos de recepção e de transmissão é de aproximadamente 2,6 (38,5/15), próximo ao valor esperado de 2,76.

7.4 TESTE DE RAJADAS

Foi realizado um teste de rajadas para verificar a validade da equação 4.5. O teste realizado possui um grau de liberdade, onde o número de mensagens por rajada pode variar.

O número de mensagens por rajada variou de 40 a 72. Foram enviadas 900 rajadas com intervalo de 1 segundo entre cada rajada para cada valor do número de mensagens, totalizando 28.800 rajadas enviadas (1.663.200 mensagens). A taxa de bits na CAN foi 250 kbps e na RS-232 foi de 115,2 kbps. O tamanho do *buffer* de transmissão da RS-232 foi de 40 mensagens.

A transmissão inicial das mensagens foi realizada na CAN por um módulo externo, com a Ponte recebendo estas mensagens, convertendo e transmitindo na RS-232 e RS-485. A interface RS-232 estava sendo monitorada por um computador a partir de um adaptador USB-Serial, salvando os dados para análise. Desconsideramos qualquer possível influência do conversor USB-Serial nos resultados. As capturas de tela mostradas na figura 7.2 foram realizadas durante este teste.

As mensagens enviadas possuem tamanho de dados igual a 8 bytes, com 2 bytes realizando contagem das mensagens enviadas. O contador foi reiniciado sempre que as 900 rajadas eram enviadas. Para evitar o uso do *bit-stuffing* na transmissão da CAN, foram utilizados identificador e os outros 6 bytes de dados com bits alternantes; o identificador utilizado foi 0x0AAAAAAA e os dados eram igual a 0x55, exceto os dois bytes que realizam a contagem das mensagens transmitidas. A figura 7.5 mostra como os dados foram transmitidos pela ponte e recebidos pelo computador. Os campos destacados em negrito contém os dados não fixados, como o valor do contado de 16 bits e o CRC.

Início	**Identif.**				**Tam.**	**Dados (8 Bytes)**								**CRC**	
										ContL	ContH				
02	0A	AA	AA	AA	08	55	55	55	55	**00**	**00**	55	55	**3F**	**30**

Figura 7.5: Mensagem UART recebida.
Fonte: Autoria própria.

Como exemplo, as três primeiras e as três últimas mensagens da primeira rajada que recebidas pela ponte e transmitidas na UART são mostradas a seguir.

```
00:  02  0A AA AA AA  08  55 55 55 55  00 00 55 55  3F 30
01:  02  0A AA AA AA  08  55 55 55 55  03 00 55 55  31 1A
02:  02  0A AA AA AA  08  55 55 55 55  04 00 55 55  C2 3C
     ...
37:  02  0A AA AA AA  08  55 55 55 55  27 00 55 55  2C 6D
38:  02  0A AA AA AA  08  55 55 55 55  01 00 55 55  CB 03
39:  02  0A AA AA AA  08  55 55 55 55  02 00 55 55  C5 29
```

Note que as mensagens não foram recebidas na sequência esperada. Levantou-se a hipótese de que isso ocorreu devido ao módulo que gerou as mensagens ter transmitido elas na sequência mostrada. O código de geração das mensagens foi modificado, para validar a hipótese, e confirmou-se o funcionamento correto da ponte de interfaces.

O módulo gerador de mensagens (idêntico ao utilizado como ponte) utiliza três *buffers* de transmissão da CAN, nos quais ocorre arbitração das mensagens enfileiradas. Porém como

os identificadores das mensagens eram todos iguais, portanto não haveria arbitração entre eles pois têm a mesma prioridade, o *hardware* do controlador CAN deu preferência por enviar as mensagens presentes na primeira posição do *buffer*. Dessa forma, a primeira mensagem (com contador zero) foi enfileirada na primeira posição, iniciando sua transmissão; a segunda e a terceira mensagens foram enfileiradas na segunda e terceira posição enquanto a primeira era enviada; então após o fim da transmissão da primeira mensagem a quarta (com contador três) foi enfileirada no primeiro *buffer*, sendo escolhida pelo *hardware* para envio; e assim sucessivamente até que todas a mensagem número 40 foi enfileirada na primeira posição e teve seu envio finalizado. Nesse ponto o *hardware* do controlador CAN escolheu para envio a mensagem na segunda posição e em seguida a da terceira.

Isso não aconteceria caso o contador de mensagens estivesse utilizando o identificador, pois ocorreria arbitragem das mensagens a partir dos seus identificadores, sendo enviada a mensagem de maior prioridade primeiro, ou seja, menor identificador primeiro. Porém os resultados não foram influenciados devido as mensagens estarem fora de ordem.

A figura 7.6 mostra um gráfico obtido à partir da análise das mensagens convertidas pela Ponte e recebidas pelo computador com a interface RS-232. Para cada rajada transmitida registrou-se o ponto (x, y), onde x e y identificam o número de mensagens transmitidas e recebidas, respectivamente. A curva superior (em azul) mostra as maiores rajadas recebidas e a curva inferior (em verde) mostra as menores. Ambas as curvas possuem inclinação unitária para $N \leq 64$. A curva inferior possui inclinação média de 0,375 $\approx$ 1/2,67 para $N > 64$ (intervalo de $64 < x \leq 72$).

Porém a curva superior possui uma inclinação de 0,5 = 1/2 para $N > 64$ (intervalo de $64 < x \leq 72$). A transmissão de um maior número de mensagens na RS-232 é causada pela variação no tempo de transmissão (ou *jitter* de transmissão) que ocorre na CAN devido ao *bit-stuffing*, utilizado para garantir a sincronização entre transmissor e receptor.

No pior caso, ou seja quando o *bit-stuffing* é mínimo, cada 2,67 mensagens transmitidas na CAN a interface RS-232 é capaz de transmitir uma mensagem após o preenchimento do *buffer*.

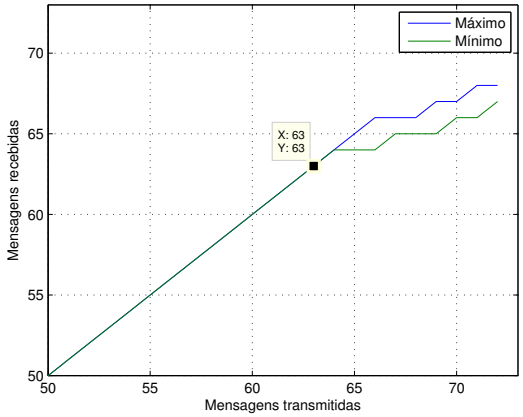

Figura 7.6: Resultado do teste de rajadas.
Fonte: Autoria própria.

De acordo com a equação 4.5, repetida abaixo, para um *buffer* de tamanho 40 é possível enviar uma rajada de até 63 mensagens na CAN de forma que todas sejam enviadas com sucesso. Devido a primeira mensagem recebida ser consumida assim que chega ao *buffer* foi possível transmitir 64 mensagens com sucesso.

$$
\begin{aligned}
N &< \frac{N_{BUFF}}{1 - T_{PROD}/T_{CONS}} \\
&= N_{BUFF} / \left(1 - \frac{N_{bits\,msg\,CAN}/f_{CAN}}{N_{bits\,msg\,UART}/f_{UART}}\right) \\
&= 40 / \left(1 - \frac{128/250000}{160/115200}\right) \\
&= 63,3
\end{aligned}
$$

Acima, $N_{bits\,msg\,CAN}$ denota o número de bits em uma mensagem CAN (foi considerada uma mensagem com 8 bytes de dados) e $N_{bits\,msg\,UART}$ denota o número de bits em uma mensagem UART, incluindo bits de início e de parada, f_{CAN}e f_{UART} são as taxas de transferência das interfaces CAN e UART em bps (bits por segundo).

8 CONCLUSÃO

No decorrer do desenvolvimento deste trabalho identificamos algumas situações onde a utilização de uma ponte de interfaces é capaz de aumentar a eficiência e flexibilidade dos canais de comunicação utilizados por dispositivos de um sistema de tempo real.

As aplicações de maior destaque de uma ponte de interfaces são: extensão de barramento, conversor de taxa de transmissão, conversor de interfaces e segmentador de barramentos (filtro seletivo de mensagens).

É descrita a implementação da conversão de protocolos CAN-UART, a recepção e transmissão de mensagens nas interfaces CAN e UART e a implementação de uma ponte de três interfaces: CAN, RS-232 e RS-485. Também realizamos uma análise de interfaces com tempos de transmissão diferentes, identificando as limitações de transmissão que devem ocorrer para que a transmissão de todas as mensagens seja garantida.

Utilizando os conceitos e métodos descritos neste documento é possível implementar as aplicações citadas acima de maneira isolada ou composta, sendo possível atender as necessidades de um grande conjunto sistemas.

Para a implementação de uma ponte de interfaces é muito importante que haja determinismo das operações que ocorrem durante as interrupções. Dessa forma será possível verificar se todas as interrupções geradas serão atendidas.

Durante execução do código de aplicação o determinismo é desejável, porém não é essencial. É possível determinar se o processamento não-determinístico afetará o funcionamento da ponte considerando o tempo de processamento como constante e igual ao o pior caso. Garantindo que é possível realizar o processamento contínuo do pior caso podemos considerar o comportamento não-determinístico no contexto de aplicação, sendo assim possível realizar o processamento de todas as mensagens, independente da capacidade utilizada nos barramentos.

Por utilizar um sistema operacional de tempo real (determinístico) foi possível atender a necessidade de determinismo nas interrupções e aplicação. A disponibilidade de filas, semáforos e *mutex* pelo sistema operacional foi de grande utilidade, permitindo focar na implementação da ponte e não no desenvolvimento de estruturas determinísticas de sincronização e troca de dados.

A ponte de interfaces implementada funciona conforme esperado, com tempos de transmissão medidos compatíveis com os valores calculados. No teste de rajadas de mensagens, os número limite de mensagens transmitidas com sucesso foi idêntico ao valor calculado a partir do tamanho do *buffer* disponível. As condições de teste foram CAN a 250 kbps, com apenas dois dispositivos em comunicação e RS-232 e RS-485 a 115,2 kbps, também com apenas dois dispositivos comunicando. A análise de utilização das filas de mensagens foi realizada considerando que as transmissões poderiam ser realizadas instantaneamente nos barramentos.

Como sugestão para trabalhos futuros é possível realizar a análise para transmissões em

barramentos já em uso por outros dispositivos, ou seja, analisar em barramentos com capacidade utilizada diferente de zero. Também é possível realizar uma análise mais profunda da influência do *bit-stuffing* e erros de transmissão, que ocorrem nas interfaces CAN, no desempenho da conversão CAN-UART realizada pela ponte de interfaces.

REFERÊNCIAS BIBLIOGRÁFICAS

ALANI, Mohammed M. **Guide to OSI and TCP/IP Models**. New York: Springer, 2014.

AXELSON, Jan. **Serial Port Complete:** COM Ports, USB Virtual Ports and Ports for Embedded Systems. Second edition. Madison: LLC, 2007.

CORRIGAN, Steve. **Introduction to the Controller Area Network (CAN)**. Dallas: Texas Instruments, 2008. Disponível em: <http://www.ti.com/lit/an/sloa101a/sloa101a.pdf>. Acesso em maio/2015.

EDWARDS, James; BRAMANTE, Richard. **Networking Self-Teaching Guide**. Indianapolis: Wiley, 2009.

FARINES, Jean-Marie; FRAGA, Joni da Silva; OLIVEIRA, Rômulo Silva de. **Sistemas de Tempo Real**. Terceira edição. Florianópolis: Departamento de Automação e Sistemas, 2000.

KOPETZ, Hermann. **Real-Time Systems:** Design Principles for Distributed Embedded Applications. Second edition. New York: Springer, 2011.

NXP SEMICONDUCTORS. **UM10211 LPC23XX User manual**. Rev. 4.1. NXP, 2012. Disponível em: <http://www.nxp.com/documents/user_manual/UM10211.pdf>. Acesso em maio/2015.

PRASAD, P. Rajendra; RAO, S.P Venu Madhava. **Data Transactions from UART to SPI Slave Devices through UART-SPI Controller from an SOC**. Hyderabad: Research India Publications, 2013. Disponível em: <http://www.ripublication.com/aeee/53_pp%20%20%20413-420.pdf>. Acesso em maio/2015.

REAL TIME ENGINEERS LTD. **FreeRTOS:** Real Time Operating System for Embedded Systems. Real Time Engineers Ltd., 2015. URL: <http://www.freertos.org/>. Acesso em maio/2015.

ROBERT BOSCH GMBH. **CAN Specification**. Version 2.0. Stuttgart: Bosh, 1991. Disponível em: <http://esd.cs.ucr.edu/webres/can20.pdf>. Acesso em maio/2015.

SOJKA, Michal; PÍŠA, Pavel; ŠPINKA, Ondřej; HARTKOPP, Oliver; HANZÁLEK, Zdeněk. **Timing Analysis of a Linux-Based CAN-to-CAN Gateway**. Schramberg: Thirteenth Real-Time Linux Workshop, 2011. Disponível em: <https://lwn.net/images/conf/rtlws-2011/proc/Sojka.pdf>. Acesso em maio/2015.

TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Terceira edição. São Paulo: Pearson, 2009.

WATTERSON, Conal. **Controller Area Network (CAN) Implementation Guide**. Norwood: Analog Devices, 2012. Disponível em: <http://www.analog.com/media/en/technical-documentation/application-notes/AN-1123.pdf>. Acesso em maio/2015.