

Local Interconnect Network (LIN)

Allan Carlos Figueiredo Echeverria*

Vicente Knihs Erbs*

*Universidade Federal de Santa Catarina (UFSC)

Resumo—Vários protocolos de comunicação foram desenvolvidos e são utilizados pela indústria automotiva, conectando a crescente quantidade de unidades lógicas (ECUs) presentes nos veículos atuais. Nesse cenário, o *Local Interconnect Network* (LIN) surge como uma alternativa menos custosa para o amplamente utilizado *Controller Area Network* (CAN). O presente trabalho traz uma implementação dessa interface serial, fazendo a comunicação entre dois microcontroladores. A linha por onde transitam os dados é avaliada e uma representação gráfica é construída a partir dos dados adquiridos. Por fim, os resultados permitem concluir que o protocolo, apesar da simplicidade, demonstrou robustez aos experimentos realizados e se adéqua aos casos de uso propostos para este.

Palavras-chave—LIN, Sistemas de Comunicação, Microcontroladores, Sistemas Embarcados

I. INTRODUÇÃO

A indústria automotiva vem aumentando a complexidade de seus sistemas eletrônicos ao longo dos últimos anos, com diversas unidades de controle eletrônico (ECU) gerenciando desde os sistemas e subsistemas mais simples até os mais críticos, como direção, freios, injeção, entre outros [1]. Todos estes se conectam em uma arquitetura composta por um vasto número de protocolos de comunicação utilizados pelo setor, assim que a compatibilidade entre estes e os custos associados a essa abordagem são fatores determinantes para o projeto de veículos neste momento [2].

Cada um dos protocolos desenvolvidos tem características próprias e que adéquam-no para propósitos específicos. O *Controller Area Network* (CAN) vem sendo a principal escolha desde os anos 80 dada sua confiabilidade e robustez, necessária para questões mais críticas e que comprometem a segurança dos passageiros [3]. Todavia, o seu custo mais elevado fez surgir o *Local Interconnect Network* (LIN) que, apesar de ser menos sólido que o CAN, pode ser usado como uma alternativa menos custosa para subsistemas não-críticos [4].

O presente trabalho traz a implementação de um sistema de comunicação usando o LIN. Dois microcontroladores se comunicam usando o protocolo, enquanto um terceiro microcontrolador faz a leitura dos níveis de tensão na linha e repassa a informação a um computador para gerar uma representação gráfica da comunicação em tempo real. Também, submeteu-se o sistema a ruídos externos e avaliou-se a influência desse fator sobre a linha de comunicação. Além disso, um método extra de detecção de erros foi adicionado à aplicação, trazendo maior robustez para o LIN.

Na Seção II são discutidos os aspectos relacionados ao LIN e aos métodos de detecção de erros utilizados; a Seção III expõe o experimento e sua estrutura; em seguida, os resultados desse experimento são apresentados na Seção IV; por fim, na Seção V o documento é concluído.

II. FUNDAMENTAÇÃO TEÓRICA

O experimento proposto faz uso da interface serial LIN e dos métodos de detecção de erro por bit de paridade, *checksum* e *Cyclic Redundancy Check* (CRC). Essa seção trata cada um desses tópicos em maiores detalhes sobre seu uso e funcionamento.

A. Local Internet Network (LIN)

A interface serial *Local Interconnect Network* (LIN), ISO17897, surgiu em 1999 como uma alternativa menos custosa para o *Controller Area Network* (CAN) em redes automotivas [4]. Isso se deve ao fato de o LIN ter certas características que reduzem o seu custo de implementação. As principais características da rede estão listadas a seguir [2, 4, 5]:

- Um único fio para comunicação;
- Comunicação assíncrona, implementada a partir de interfaces comuns – no caso, *Universal Asynchronous Receiver/Transmitter* (UART) ou *Serial Communications Interface* (SCI);
- Um mestre, múltiplos escravos (*Single Master / Multiple Slave*) – até 16 escravos;
- Até 19,2 Kbits/s;
- Máxima distância 40 metros;
- Auto-sincronização;
- Flexibilidade para adicionar novos *slaves*;
- Latência garantida entre mensagens;
- Possibilidade de *broadcasting* (transmitir o mesmo comando ou mensagem para todos os nós da rede).

Com isso, os custos conseguem ser reduzidos pelo uso de apenas um fio e pelo fato de a maioria absoluta dos microcontroladores atuais possuírem alguma interface UART ou SCI [2].

Em contrapartida, como um único *master* controla toda a rede, caso esse nó esteja comprometido, toda a rede é comprometida [3]. Além disso, a taxa de transmissão é muito menor comparada aos 1000 Kbits/s do protocolo CAN.

Sobre os padrões desse protocolo, a comunicação sempre acontece com a mesma estrutura. A Figura 1 extraída de [6] mostra como se dá essa comunicação: sempre o mestre envia um *header* solicitando informação a algum escravo ou enviando algum comando; então, o *slave* correspondente responde com a informação requerida e o *checksum* para proteger contra eventuais falhas na comunicação. Note que apenas o mestre pode iniciar a comunicação no LIN.

O *header* enviado pelo master é dividido em três partes: primeiro, a linha recebe um *break* que coloca a tensão em nível lógico baixo por um período maior que o requerido para enviar



Figura 1: Formato de envio das mensagens em LIN [6].

um byte; em seguida é enviado o byte de sincronização (0x55 em hexadecimal, que é uma sequência de 0's e 1's alternados), usado para sincronizar *master* e *slaves*; por último o byte de identificação é enviado. Esse último byte é essencialmente o comando que o mestre está dando para os escravos, sendo que os seis primeiros bits são a informação em si e os dois bits restantes são bits de paridade. O comando pode ser algum especificado pelo próprio protocolo – como geração de eventos, diagnósticos sobre os *slaves*, comandos definidos pelo usuário ou até códigos reservados – ou uma solicitação padrão de informação. Nesse caso, é enviado o identificador (ID) do *slave* que deverá enviar as informações que coletou.

Uma vez recebida essa solicitação, o escravo verifica se ele deverá responder ou ignorar a mensagem recebida. No caso de o ID coincidir com o seu, este envia as informações que possui. Esse dado pode ter de dois a oito bytes, sendo que essa quantidade é predefinida de acordo com o ID do *slave* (certas faixas enviam dois bytes, outras três, e assim por diante). Por fim, é enviado o *checksum* para que o mestre possa identificar eventuais erros na mensagem recebida.

Já sobre os aspectos elétricos do protocolo, a tensão de operação na linha é de 12V. Para o emissor, uma margem de 20% nos níveis de tensão é estabelecida, sendo que para o receptor essa margem é de 40% [7]. Um nível lógico alto representa uma tensão próxima de 12V, enquanto que um nível lógico baixo representa valores próximos de 0V.

A Figura 2 obtida em [7] mostra como ocorre a conexão entre mestre e escravos no LIN. Pode-se perceber que, nessa situação, o transceptor LIN é comandado pelo microcontrolador por meio de SCI, enquanto que os transceptores LIN trocam informações entre si por um único fio.

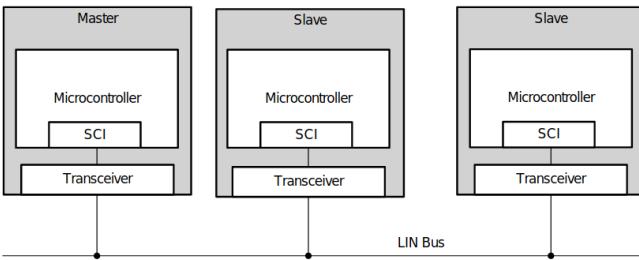


Figura 2: Estrutura de uma rede LIN [7].

Portanto, o LIN especifica várias camadas do protocolo, que vão desde a camada física até a aplicação. Com esse padrão bem definido, componentes de qualquer fabricante devem操

rar corretamente em conjunto e critérios de interoperabilidade podem ser alcançados.

B. Métodos de Detecção de Erro

Naturalmente, durante a comunicação entre dispositivos podem ocorrer interferências na linha e erros na mensagem acabam surgindo. A fim de identificar esses erros, existem os métodos de detecção que adicionam informações extras à mensagem, trazendo maior confiabilidade para a rede.

Dois métodos foram utilizados nesse projeto: o *checksum* e o *Cyclic Redundancy Check* (CRC). Cada um deles será apresentado em maiores detalhes a seguir.

1) *Bit de Paridade*: Um dos métodos mais simples de detecção de erros é o bit de paridade. A ideia é adicionar bits a uma mensagem de modo que a soma de todos os bits seja um valor par ou ímpar [8]. Assim, apenas um número ímpar de erros pode ser identificado corretamente.

Existe, portanto, uma separação entre paridade ímpar e paridade par, sendo cada variante do método previamente definida pelo protocolo ou sistema. No caso do LIN, utiliza-se a paridade par [7] construída sobre os dois bits finais do byte identificador – como é possível perceber na Figura 1.

Dadas essas características do método, esse é utilizado em situações nas quais o tempo de processamento é restrito, mas existe uma margem maior para erros na troca de dados.

2) *Checksum*: Já o *checksum* traz uma sensibilidade melhorada para a detecção de erros em comparação ao bit de paridade [8]. Para esse método, os valores são somados byte a byte, e o resultado da soma é invertido (cada bit zero se adquire o valor um e vice-versa) e então enviado. Vale ressaltar que o tamanho para esse valor enviado é fixo, sendo que comumente são adotados *checksums* de 8, 16 e 32 bits.

No decodificador, a soma é feita tanto para os bytes da mensagem quanto para o *checksum* recebido. Após a inversão do resultado, espera-se que o valor obtido seja nulo, de modo que nenhum erro tenha sido identificado na mensagem. Caso contrário, houve um erro na comunicação.

Para o protocolo em questão, o *checksum* é adotado como parte de sua estrutura, sendo enviado em toda resposta do escravo. Um byte é o tamanho da informação adicional, portanto têm-se um *checksum* de 8 bits no LIN.

3) *Cyclic Redundancy Check (CRC)*: O *Cyclic Redundancy Check* (CRC) é um algoritmo baseado na divisão polinomial em base 2 [9]. Por meio de operações XOR (ou exclusivo) acontece a divisão da mensagem a ser enviada (*dataword*)

pelo polinômio gerador. O resto dessa divisão é adicionado à mensagem, formando a *codeword*. Na decodificação, a *codeword* é dividida pelo polinômio gerador e, caso o resto (síndrome) for nulo, a mensagem foi enviada corretamente.

Existem diversos algoritmos para o CRC, variando o tamanho da mensagem a ser avaliado e o gerador. No caso em questão, como apenas um byte de informação compõe a *dataword*, o CRC-8 foi utilizado (oito bits ou um byte). Já o polinômio gerador foi o valor 0xD5 em hexadecimal, e esse valor tabelado cumpre os requisitos para que o CRC possa identificar erros de bit corretamente.

Também é possível implementar o CRC com uma abordagem baseada em uma tabela. Como a lista de valores possíveis de *dataword* para o CRC não é muito extensa, pode-se consultar qual o valor a ser adicionado à mensagem em uma tabela contendo essas relações. Ao final, o resultado obtido é o mesmo, porém com melhor desempenho em questão de tempo de execução, ao custo de um maior uso de memória.

III. MONTAGEM DO EXPERIMENTO

Foi feita uma implementação de um sistema de comunicação utilizando o protocolo LIN, a qual será apresentada em maiores detalhes a seguir. A comunicação acontece entre dois microcontroladores Tiva TM4C123GXL [10] da Texas Instruments, sendo um mestre e um escravo. Por meio de comunicação serial UART, cada microcontrolador interage com um transceptor LIN no formato de circuito integrado (CI), o MCP2003 [11] fabricado pela Microchip. Por sua vez, os transceptores se comunicam por meio do protocolo LIN, permitindo a troca de mensagens entre mestre e escravo.

Um terceiro microcontrolador foi adicionado – um Raspberry Pi Pico [12] – a fim de monitorar a tensão na linha em que acontece a comunicação por LIN. Os dados são obtidos por um conversor analógico-digital (ADC) e salvos em um arquivo no computador.

Os microcontroladores enviam mensagens ao computador no qual estão conectados; portanto, o monitoramento do que é recebido pode ser feito com qualquer software de comunicação serial. O Minicom [13] foi utilizado durante os experimentos para esse propósito, bem como para salvar os dados lidos pelo Pi Pico em um arquivo de texto.

O esquemático das conexões feitas pode ser visto na Figura 3. Nota-se que foi usado um divisor de tensão para as leituras na linha, isso porque o microcontrolador só suporta leituras até 3,3V, mas o LIN opera em 12V. Também, todos os três microcontroladores são alimentados diretamente do computador, porém os transceptores precisam de uma alimentação de, no mínimo, 12V para operar. Assim, uma alimentação externa é feita com duas baterias de 9V usadas e é acondicionada pelos diodos, o capacitor e o diodo Zener. As outras partes do circuito são especificadas pelo *datasheet* do MCP2003.

Os códigos de cada microcontrolador podem ser encontrados nos Apêndices A (mestre), B (escravo) e C (Pi Pico). O código para o responsável por medir a tensão na linha é bastante direto: um *loop* infinito em que ele faz a leitura, quantifica para um intervalo de 0 a 255 e imprime o valor. A impressão é usada para gerar o arquivo com as leituras, e

a quantificação é pelo fato de o valor precisar corresponder a um caractere. Já para o mestre e o escravo o código é um pouco mais complexo.

Basicamente, o código implementa a estrutura proposta na Figura 1, porém com algumas diferenças. O mestre possui um método para enviar o *header* solicitando a mensagem, o qual se repete em até três vezes no caso de a resposta conter algum erro – identificado pelos métodos de detecção. Esse método é executado periódica e indefinidamente.

No lado do *slave*, um caractere predefinido é enviado juntamente com o CRC do mesmo. Ao final, o *checksum* é enviado normalmente. Com dois métodos de detecção de erro, o sistema ganha uma robustez extra, que é justamente um dos contras desse protocolo. O escravo é ativado por uma interrupção, e caso o destino não seja ele a mensagem é simplesmente ignorada. Já se o ID for o seu, mas os bits de paridade estiverem incorretos, uma resposta errada é enviada para que o mestre possa repetir a solicitação.

As mensagens são enviadas no seguinte formato: primeiro, tanto uma solicitação quanto uma resposta correta devem ser enviadas; depois, o mestre deve enviar o bit de paridade incorreto e, quando então solicitado novamente pelo escravo, enviar o *header correto*; em seguida, o escravo deve enviar três respostas em que o CRC está incorreto; por fim, três respostas em que o *checksum* está incorreto são enviadas. Esse ciclo se repete indefinidamente, permitindo verificar o correto funcionamento de ambos microcontroladores e de todos os métodos de detecção de erro.

A fim de criar um gráfico a partir dos dados obtidos e salvos no arquivo de texto, utilizou-se o código disponível no Apêndice D. Com ele, uma animação dos dados surge, simulando a execução em tempo real. Sendo assim, uma visualização gráfica do que ocorre na linha é exposta.

O experimento foi construído sobre uma *protoboard*, como mostra a fig 4. Um transformador foi posicionado próximo aos componentes para gerar ruído na linha e permitir uma análise do sistema sob condições mais próximas do caso de uso real.

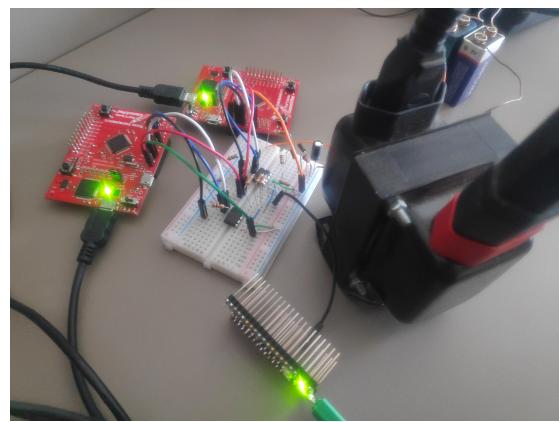


Figura 4: Imagem do experimento montado.

Em uma implementação do sistema para um caso de uso real, isto é, em algum subsistema que componha um veículo em produção, algumas partes seriam feitas de maneiras distintas. Por exemplo, a *protoboard* deve ser substituída por uma

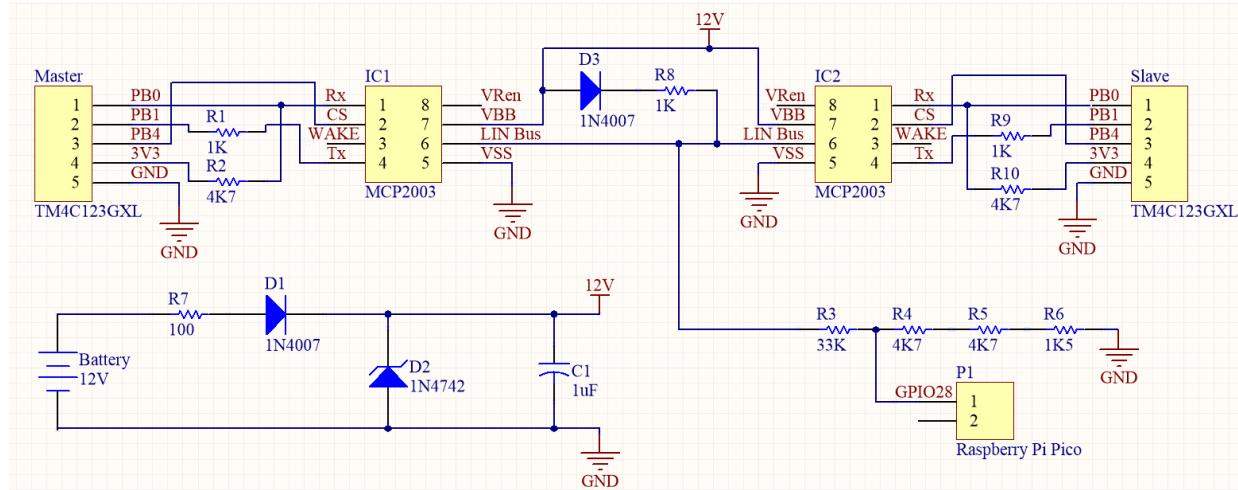


Figura 3: Esquemático das conexões do experimento.

placa de circuito impresso (PCB), os componentes eletrônicos devidamente substituídos e a fonte de alimentação deve ser mais adequada. Todavia, o MCP2003 [11] poderá ser adotado para essa situação, na posição de componente principal da rede.

Com isso, o experimento pode ser construído e testado, gerando dados que permitem avaliar o protocolo de comunicação serial *Local Interconnect Network*. Os resultados obtidos são discutidos na seção seguinte.

IV. RESULTADOS EXPERIMENTAIS

Conforme especificado até o momento, montou-se o experimento a fim de verificar o correto funcionamento da comunicação a partir do protocolo LIN. Os dados obtidos serão analisados gráfica e estatisticamente.

Uma captura do gráfico gerado a partir das leituras de tensão na linha está exposto na Figura 5, sendo que o início de uma mensagem é exibido. Várias informações podem ser obtidas apenas da análise dessa imagem.

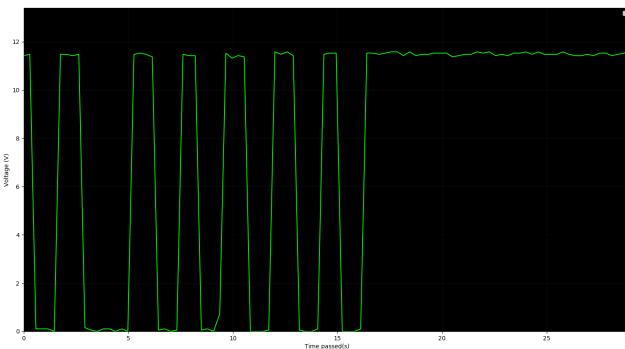


Figura 5: Captura do gráfico gerado a partir dos dados obtidos.

Primeiramente, percebe-se que os primeiros bits enviados são os de sincronização, representados pela alternância de zeros e uns lógicos (valor 0x55 em hexadecimal). Portanto, o MCP2003 [11] utiliza o sinal *break* apenas para início de seu próprio sistema, como uma interrupção, não passando essa situação adiante. Uma vez que essa sinalização dura mais do

que o tempo para envio e recebimento de um byte, é lógico pensar que essa informação não deve ser transmitida adiante, pois comprometeria a leitura pelo nó escravo.

Nota-se, também, que os níveis de tensão ficaram muito próximos de 12V e 0V, assim como prevê o protocolo. Ademais, enquanto nenhuma informação é enviada a tensão é mantida em nível alto, assim como acontece com a UART.

Após essa análise qualitativa dos resultados obtidos, utilizou-se de uma análise estatística para averiguar a conformidade dos níveis de tensão da linha. Os valores estão expostos na Tabela I. Esses números foram calculados a partir de mais de 500000 (quinhetas mil) leituras obtidas.

Tabela I: Resultados estatísticos dos níveis de tensão.

Nível Lógico	Estatística	Quantizado	Tensão [V]
1	Média	227	11,831
	Desvio-Padrão	2,414	0,127
	Mínimo	148	7,714
	Máximo	236	12,301
0	Média	6	0,313
	Desvio-Padrão	4,513	0,235
	Mínimo	0	0
	Máximo	136	7,088

Como é possível perceber, a tensão ficou em 11,831V na média para o nível lógico alto, com um desvio-padrão de 0,127V. Para o nível lógico baixo o valor ficou em 0,313V, com um desvio-padrão de 0,235V. Ambos os resultados demonstram que a tensão ficou bem estável ao longo desses níveis, além de estarem muito próximas aos valores nominais – completamente aceitável dentro das faixas de operação.

O valor máximo para o nível zero e o valor mínimo para o nível alto, porém, ficam muito foras dos limites de operação e seriam interpretados como erros. O número 138 (quantizado) foi utilizado como separador por representar 40% do máximo, que seria o limite para sair da faixa aceitável para o nível alto. Nesse caso, avaliou-se a incidência de valores fora das faixas aceitáveis de operação, como exibido na Tabela II. Os limites 92 e 138 representam as faixas aceitáveis de operação, por isso foram escolhidos. Os outros valores – 20 e 210 – foram utilizados para levar a uma situação mais extrema e verificar

a quantidade de valores existentes próximos do limite.

Tabela II: Ocorrência de valores fora da margem de aceitação.

Faixa	Ocorrências	Percentual
Valores entre 92 e 138 (inclusive)	10	0,0019%
Valores entre 20 e 210 (inclusive)	36	0,0068%

Percebe-se que a ocorrência de valores fora da faixa de operação é ínfima, com percentual de menos de 0,01%. Possivelmente, esses dados são oriundos de medições feitas exatamente no momento em que a tensão estava passando de um nível alto para baixo ou vice-versa.

Desse modo, pôde-se avaliar o desempenho e particularidades do protocolo LIN, exibindo ótimos resultados apesar do baixo custo em comparação com outras alternativas para o mercado automobilístico.

V. CONCLUSÃO

Logo, foi feita uma implementação do protocolo *Local Interconnect Network* (LIN) e avaliou-se o seu desempenho por métodos gráficos e estatísticos. O experimento consistiu na comunicação entre dois microcontroladores, um executando a função de mestre e outro de escravo. Um terceiro microcontrolador foi adicionado para medir os níveis de tensão na linha, e um gráfico animado, simulando a execução em tempo real, foi gerado a partir dos dados obtidos. Uma fonte de ruído externo foi adicionada, fornecendo uma análise mais completa do protocolo. Além disso, um método de detecção de erros extra foi adicionada ao sistema, o CRC, complementando os já utilizados *checksum* e bits de paridade.

Apesar de ser tido como uma alternativa mais barata e, contudo, menos confiável de protocolo de comunicação serial para redes automotivas, os experimentos mostraram um ótimo desempenho do LIN. A ocorrência de valores fora dos limites aceitáveis é desprezível, com uma estabilidade visível nos níveis de tensão para ambos níveis lógicos. Entretanto, o protótipo foi feito com apenas um escravo e com uma linha extremamente curta. Também, o ruído eletromagnético em veículos pode ser maior em situações práticas do que aquele adicionado. Trabalhos futuros podem reconsiderar esses aspectos durante a análise estatística dos dados, estudando seus efeitos.

Portanto, o LIN é um protocolo de comunicação feito para redes automotivas e que expõe ótimos resultados experimentais, devendo ser considerados como uma alternativa viável para muitos dos subsistemas que compõem as funcionalidades veiculares.

Electronic Materials Production, 2003. IEEE, 2003, pp. 150–153.

- [3] J. M. Ernst and A. J. Michaels, “Lin bus security analysis,” in *IECON 2018-44th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2018, pp. 2085–2090.
- [4] Y. Xu, J. Wang, W. Chen, J. Tao, and Q. Liu, “Application of lin bus in vehicle network,” in *2006 IEEE International Conference on Vehicular Electronics and Safety*. IEEE, 2006, pp. 119–123.
- [5] M. Popa, V. Groza, and A. Botas, “Lin bus testing software,” in *2006 Canadian Conference on Electrical and Computer Engineering*. IEEE, 2006, pp. 1287–1290.
- [6] “LIN bus explained - a simple intro (2021).” [Online]. Available: <https://www.csselectronics.com/screen/page/lin-bus-protocol-intro-basics/language/en>
- [7] E. Hackett, “LIN protocol and physical layer requirements,” 2018.
- [8] M. Rahmani, W. Hintermaier, B. Muller-Rathgeber, and E. Steinbach, “Error detection capabilities of automotive network technologies and ethernet-a comparative study,” in *2007 IEEE Intelligent Vehicles Symposium*. IEEE, 2007, pp. 674–679.
- [9] C. Partridge, J. Hughes, and J. Stone, “Performance of checksums and crcs over real data,” *ACM SIGCOMM Computer Communication Review*, vol. 25, no. 4, pp. 68–76, 1995.
- [10] *Datasheet TM4C123GH6PM Microcontroller*, Texas Instruments. [Online]. Available: <https://www.ti.com/lit/ds/spsm376e/spsm376e.pdf>
- [11] *Datasheet MCP2003/4 Microcontroller LIN Transceiver*, Microchip. [Online]. Available: <http://ww1.microchip.com/downloads/en/devicedoc/22230a.pdf>
- [12] *Raspberry Pi Pico Datasheet – an RP2040-based microcontroller board*, Raspberry Pi. [Online]. Available: <https://datasheets.raspberrypi.org/pico/pico-datasheet.pdf>
- [13] “Man page of minicom.” [Online]. Available: <http://fmatrm.if.usp.br/cgi-bin/man/man2html?1+minicom>

REFERÊNCIAS

- [1] I. Studnia, V. Nicomette, E. Alata, Y. Deswarthe, M. Kaâniche, and Y. Laarouchi, “Survey on security threats and protection mechanisms in embedded automotive networks,” in *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*. IEEE, 2013, pp. 1–12.
- [2] C. Gabriel and H. Horia, “Integrating sensor devices in a lin bus network,” in *26th International Spring Seminar on Electronics Technology: Integrated Management of*

APÊNDICE A
MASTER.C

```

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_gpio.h"
#include "inc/hw_uart.h"

#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"
#include "driverlib/uart.h"

#define ESC_REG(x) (*((volatile uint32_t *)(x)))
#define UART_BASE UART1_BASE

unsigned long message_count = 0;

void configure_output_pin(uint32_t port, uint8_t pin)
{
    GPIOPinTypeGPIOOutput(port, pin);
    GPIOPadConfigSet(port, pin, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
}

uint8_t checksum(uint8_t val[])
{
    int soma = val[0] + val[1] + val[2];
    int res = (soma & 0xFF) + ((soma >> 8) & 0xFF);
    res = (~res) & 0xFF;

    if (res == 0) return 1;
    return 0;
}

uint8_t CRCDecode(uint8_t val[])
{
    unsigned int data = ((val[0] << 8) & 0xFF00) + (val[1] & 0xFF);
    unsigned int res = (data >> 8) & 0xFF;
    uint8_t i;
    for (i = 0 ; i < 8 ; i++) {
        res = (res << 1) & 0x1FF;
        res += ((data >> (7 - i)) & 0x1) & 0x1FF;
        res ^= 0xD5 * (res / 256);
    }

    res = res & 0xFF;

    if (res == 0) return 1;
    return 0;
}

```

```

int sendHeader(uint8_t slaveID) {

    if (slaveID > 59 || slaveID < 2) {
        UARTCharPut(UART0_BASE, 'E');
        UARTCharPut(UART0_BASE, ':');
        UARTCharPut(UART0_BASE, ' ');
        UARTCharPut(UART0_BASE, 'I');
        UARTCharPut(UART0_BASE, 'D');
        UARTCharPut(UART0_BASE, '\n');
        UARTCharPut(UART0_BASE, '\r');
        return -2;
    }

    uint8_t attempts = 0; // Checksum/CRC: if wrong, retry 3 times, return -1 if fail
    uint8_t success = 0;

    uint8_t i = 0;

    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3, 0xFF); // Cyan
    SysCtlDelay(SysCtlClockGet() / 30);

    while (success == 0 && attempts < 3) {

        message_count += 1;

        UARTCharPutNonBlocking(UART0_BASE, '0' + slaveID);

        uint8_t parity = 0;

        if ((message_count - 2) \% 9 == 0) { // Send wrong parity once in a while
            parity = 0x03;
        } else {
            for (i = 0 ; i < 6 ; i++)
                parity = parity + ((slaveID >> i) & 0x01);

            parity = parity % 2;
            parity = (parity << 1) & 0x03;
        }

        uint8_t id = (((slaveID & 0x3F) << 2) + parity) & 0xFF;

        UARTBreakCtl(UART_BASE, 1); // send BREAK
        UARTCharPutNonBlocking(UART0_BASE, '>');
        UARTCharPutNonBlocking(UART0_BASE, ' ');
        UARTBreakCtl(UART_BASE, 0); // stop BREAK

        UARTCharPut(UART_BASE, 0x55); // send SYNC (0x55)
        UARTCharPut(UART_BASE, id); // send slaveID + parity

        uint8_t received[7];
        i = 0;
        while (!UARTCharsAvail(UART_BASE));
        while (UARTCharsAvail(UART_BASE) || i < 5) {
            char input = UARTCharGet(UART_BASE);
            received[i++] = input;
        }

        // MCP2003 repeats the message sent, so it removes the first two bytes
    }
}

```

```

received[0] = received[2];
received[1] = received[3];
received[2] = received[4];

attempts = attempts + 1;
success = 1;

// Checks checksum
if (checksum(received) == 0) {
    UARTCharPutNonBlocking(UART0_BASE, 'C');
    success = 0;
}

// Checks CRC
if (CRCDecode(received) == 0){
    UARTCharPutNonBlocking(UART0_BASE, ',');
    UARTCharPutNonBlocking(UART0_BASE, 'C');
    UARTCharPutNonBlocking(UART0_BASE, 'R');
    UARTCharPutNonBlocking(UART0_BASE, 'C');
    success = 0;
}

if (success == 0) {
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00)
        ; // Off
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0xFF); // Blue
    SysCtlDelay(SysCtlClockGet() / 30);
} else {
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00)
        ; // Off
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0xFF); // Green
    SysCtlDelay(SysCtlClockGet() / 30);
    UARTCharPutNonBlocking(UART0_BASE, received[0]); // Print message
}
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); // Off
SysCtlDelay(SysCtlClockGet() / 30);

UARTCharPutNonBlocking(UART0_BASE, '\n');
UARTCharPutNonBlocking(UART0_BASE, '\r');
}

GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); // Off

if (success == 0) {
    return -1;
}

return 1;
}

// MAIN -----
int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
        SYSCTL_XTAL_16MHZ);
}

```

```

SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); // Necessary for UART0
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); // LED
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); // Output to PC
SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);

while (! SysCtlPeripheralReady(SYSCTL_PERIPH_UART0) || ! SysCtlPeripheralReady(
    SYSCTL_PERIPH_GPIOF)) {}

//UART 1 CONFIG
GPIOPinConfigure(GPIO_PB0_U1RX);
GPIOPinConfigure(GPIO_PB1_U1TX);
GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
UARTConfigSetExpClk(UART_BASE, SysCtlClockGet(), 300, (UART_CONFIG_WLEN_8 |
    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

//UART 0 CONFIG
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200, (UART_CONFIG_WLEN_8 |
    UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); // Off

configure_output_pin(GPIO_PORTB_BASE, GPIO_PIN_4);
GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);

IntMasterEnable();

while(1) {
    sendHeader(2);
    SysCtlDelay(SysCtlClockGet() * 1.7);
}

return 0;
}

```

APÊNDICE B SLAVE.C

```

#include <stdint.h>
#include <stdbool.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "inc/hw_ints.h"
#include "inc/hw_memmap.h"
#include "inc/hw_types.h"
#include "inc/hw_sysctl.h"
#include "inc/hw_gpio.h"
#include "inc/hw_uart.h"

#include "driverlib/sysctl.h"
#include "driverlib/gpio.h"
#include "driverlib/interrupt.h"
#include "driverlib/pin_map.h"

```

```

#include "driverlib/uart.h"

#define ESC_REG(x)          (*((volatile uint32_t *)(x)))
#define SLAVE_ID             2
#define MESSAGE              0x63

#define UART_BASE            UART1_BASE

typedef struct {
    unsigned int count;
    uint8_t msg;
    uint8_t buf[3];
} data_t;

data_t Data;

void configure_output_pin(uint32_t port, uint8_t pin)
{
    GPIOPinTypeGPIOOutput(port, pin);
    GPIOPadConfigSet(port, pin, GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD_WPU);
}

uint8_t checksum(data_t d) {
    int soma = d.buf[0] + d.buf[1];
    int res = (soma & 0xFF) + ((soma >> 8) & 0xFF);
    return (~res) & 0xFF;
}

uint8_t CRC(data_t d){
    unsigned int res = d.msg;
    uint8_t u;
    for (u = 0 ; u < 8 ; u++) {
        res = (res << 1) & 0x1FF;
        res ^= 0xD5 * (res / 256);
    }
    res = (res & 0xFF);

    return res;
}

/* LIN message sequence:
 *
 * 1: Right message
 * 2: sendError() (wrong parity from master)
 * 3: Right message
 * 4-6: Wrong Checksum
 * 7-9: Wrong CRC
 *
 */
void sendData() {

    Data.buf[0] = Data.msg;

    // Calculate CRC and Checksum
    Data.buf[1] = CRC(Data);
    Data.buf[2] = checksum(Data);

    if (Data.count % 9 > 5) {

```

```

// Send wrong CRC
Data.buf[1] = Data.buf[1] + 1;
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_3, 0xFF); // Yellow
SysCtlDelay(SysCtlClockGet() / 30);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); //
Off

} else if (Data.count % 9 > 2) {
    // Send wrong Checksum
    Data.buf[2] = Data.buf[2] + 1;
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2, 0xFF); // Magenta
    SysCtlDelay(SysCtlClockGet() / 30);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); //
    Off

} else {
    // Send data right
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_3, 0xFF); // Green
    SysCtlDelay(SysCtlClockGet() / 30);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); //
    Off
}

// Send msg + CRC + checksum
UARTCharPut(UART_BASE, Data.buf[0]);
UARTCharPut(UART_BASE, Data.buf[1]);
UARTCharPut(UART_BASE, Data.buf[2]);

// Uncomment lines above to print message sent
// UARTCharPutNonBlocking(UART0_BASE, Data.buf[0]);
// UARTCharPutNonBlocking(UART0_BASE, Data.buf[1]);
// UARTCharPutNonBlocking(UART0_BASE, Data.buf[2]);
// UARTCharPutNonBlocking(UART0_BASE, '\n');
// UARTCharPutNonBlocking(UART0_BASE, '\r');

Data.count = Data.count + 1;
Data.buf[0] = Data.buf[1] = Data.buf[2] = 0;
}

void sendError() {
    Data.count = Data.count + 1;
    Data.buf[0] = Data.buf[1] = Data.buf[2] = 0x55;

    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0xFF); // Red
    SysCtlDelay(SysCtlClockGet() / 30);
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); // Off

    UARTCharPutNonBlocking(UART_BASE, Data.buf[0]);
    UARTCharPutNonBlocking(UART_BASE, Data.buf[1]);
    UARTCharPutNonBlocking(UART_BASE, Data.buf[2]);

    UARTCharPutNonBlocking(UART0_BASE, Data.buf[0]);
    UARTCharPutNonBlocking(UART0_BASE, Data.buf[1]);
    UARTCharPutNonBlocking(UART0_BASE, Data.buf[2]);
    UARTCharPutNonBlocking(UART0_BASE, '\n');
    UARTCharPutNonBlocking(UART0_BASE, '\r');

    Data.buf[0] = Data.buf[1] = Data.buf[2] = 0;
}

```

```

}

void uartHandler1(void) {
    uint32_t status;
    status = UARTIntStatus(UART_BASE, true);
    UARTIntClear(UART_BASE, status);

    IntDisable(INT_UART1);

    uint8_t i = 0;

    while (UARTCharsAvail(UART_BASE)) {
        char input = UARTCharGetNonBlocking(UART_BASE);

        if (i == 1) {
            if (((input >> 2) & 0x3F) == SLAVE_ID) {
                uint8_t parity = 0;
                uint8_t j;
                for (j = 0 ; j < 6 ; j++) {
                    parity = parity + ((SLAVE_ID >> j) & 0x01);
                }
                parity = parity % 2;
                parity = (parity << 1) & 0x03;

                if (parity == (input & 0x03)) {
                    sendData();
                } else {
                    sendError();
                }
            }
            i = i + 1;
        }
        IntEnable(INT_UART1);
    }

// MAIN -----
int main(void) {
    SysCtlClockSet(SYSCTL_SYSDIV_2_5 | SYSCTL_USE_PLL | SYSCTL_OSC_MAIN |
                  SYSCTL_XTAL_16MHZ);

    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA); // Necessary for UART0
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
    SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF); // LED
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0); // Output to PC
    SysCtlPeripheralEnable(SYSCTL_PERIPH_UART1);

    while (!SysCtlPeripheralReady(SYSCTL_PERIPH_UART0) || !SysCtlPeripheralReady(
                  SYSCTL_PERIPH_GPIOF)) {}

    //UART 1 CONFIG
    GPIOPinConfigure(GPIO_PB0_U1RX);
    GPIOPinConfigure(GPIO_PB1_U1TX);
    GPIOPinTypeUART(GPIO_PORTB_BASE, GPIO_PIN_0 | GPIO_PIN_1);
    UARTConfigSetExpClk(UART_BASE, SysCtlClockGet(), 300, (UART_CONFIG_WLEN_8 |

```

```

UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE) );
UARTIntEnable(UART_BASE, UART_INT_RX | UART_INT_RT);
IntEnable(INT_UART1);

//UART 0 CONFIG
GPIOPinConfigure(GPIO_PA0_U0RX);
GPIOPinConfigure(GPIO_PA1_U0TX);
GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
UARTConfigSetExpClk(UART0_BASE, SysCtlClockGet(), 115200, (UART_CONFIG_WLEN_8 |
UART_CONFIG_STOP_ONE | UART_CONFIG_PAR_NONE));

GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);
GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3, 0x00); // Off

Data.count = 0;
Data.msg = MESSAGE;
Data.buf[0] = 0;
Data.buf[1] = 0;
Data.buf[2] = 0;

configure_output_pin(GPIO_PORTB_BASE, GPIO_PIN_4);
GPIOPinWrite(GPIO_PORTB_BASE, GPIO_PIN_4, 0xFF);

IntMasterEnable();

while(1) {
}

return 0;
}

```

APÊNDICE C

LIN-ANALOG.PY

```

from machine import Pin, ADC
from utime import sleep

adc = ADC(28)
led = Pin(25, Pin.OUT)

def ADC_to_char():
    # Returns value in a range from 0 to 255
    # 255 represents [3.3 * (33 + 4.7 + 4.7 + 1.5) / (4.7 + 4.7 + 1.5)] Volts
    # 255 -> 13.291 V
    # 1 represents 13.291 / 255
    # 1 -> 0.0521 V
    return round(adc.read_u16() * 255 / 65536)

def main():
    factor = 3.3 * (33 + 4.7 + 4.7 + 1.5) / (4.7 + 4.7 + 1.5)
    led.on()

    while True:
        print(ADC_to_char())

if __name__ == "__main__":
    main()

```

APÊNDICE D
PLOT-DATA.PY

```

#!/usr/bin/env python3
import random
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

#Open file with data to be displayed
f = open("data/data_plot.txt", 'r')
data = f.readlines()

#Convert raw data to voltage
def treat_data():
    return (int(data[n])*13.291/256)

#Lists that hold the values to be displayed
vals = [0] * 100
x_vals = np.linspace(29,0,100)

#Graphic settings
fig, ax = plt.subplots()
line, = ax.plot(x_vals, vals, '#00FF00')
ax.legend()
plt.axis([0,29,0,13.4])
plt.xlabel('Time passed(s)')
plt.ylabel('Voltage (V)')
ax.set_facecolor('#000000')
plt.grid(True, color='0.05')

n = 0
lim = len(data)
def animate(i):
    global n
    vals.append(treat_data())
    vals.pop(0)
    line.set_data(x_vals, vals)
    if n >= lim:
        n = 0
    else:
        n = n + 1
    return line

def main():
    ani = FuncAnimation(fig, animate, interval = 1)
    plt.show()

if __name__ == "__main__":
    main()

```