TRABALHO DE GRADUAÇÃO

IMPLEMENTAÇÃO DO PROTOCOLO INDUSTRIAL DE CAMPO DEVICENET EM PLATAFORMA RASPBERRY PI 3 PARA AUTOMAÇÃO DE MAQUETE FERROVIÁRIA

Eduardo da Fonseca Pereira

William Jun Yamada

Brasília, Julho de 2018

ENGENHARIA
MECATRÔNICA
UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia
Curso de Graduação em Engenharia de Controle e Automação

TRABALHO DE GRADUAÇÃO

IMPLEMENTAÇÃO DO PROTOCOLO INDUSTRIAL DE CAMPO DEVICENET EM PLATAFORMA RASPBERRY PI 3 PARA AUTOMAÇÃO DE MAQUETE FERROVIÁRIA

Eduardo da Fonseca Pereira

William Jun Yamada

Relatório submetido como requisito parcial de obtenção
de grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Guilherme Caribé de Carvalho, ENM/UnB _____
Orientador

Prof. Carlos Humberto Llanos Quintero, ENM/UnB _____
Examinador interno

Prof. Jones Yudi Mori Alves da Silva, ENM/UnB _____
Examinador interno

Brasília, Julho de 2018

FICHA CATALOGRÁFICA

DA FONSECA PEREIRA, EDUARDO JUN YAMADA, WILLIAM Implementação do Protocolo Industrial de Campo DeviceNet em plataforma raspberry pi 3 para automação de maquete ferroviária.

[Distrito Federal] 2018.

vii, 101p., 297 mm (FT/UnB, Engenheiro, Controle e Automação, 2015). Trabalho de Graduação – Universidade de Brasília.Faculdade de Tecnologia.

1. Protocolo DeviceNet 2.Protocolo CAN
3. CIP

I. Mecatrônica/FT/UnB II. Título (Série)

REFERÊNCIA BIBLIOGRÁFICA

PEREIRA, EDUARDO DA FONSECA; YAMADA, WILLIAM JUN (2018). Implementação do protocolo industrial de campo DeviceNet em plataforma Raspberry Pi 3 para automação de maquete ferroviária. Trabalho de Graduação em Engenharia de Controle e Automação, Publicação FT.TG-$n^{\circ}007$, Faculdade de Tecnologia, Universidade de Brasília, Brasília, DF, 101p.

CESSÃO DE DIREITOS

AUTORES: Eduardo da Fonseca Pereira e William Jun Yamada

TÍTULO DO TRABALHO DE GRADUAÇÃO: Implementação do protocolo industrial de campo DeviceNet em plataforma Raspberry Pi 3 para automação de maquete ferroviária.

GRAU: Engenheiro ANO: 2018

Eduardo da Fonseca Pereira

William Jun Yamada

Dedicatórias

Dedico este trabalho à minha mãe, Lindasir Yokoyama Yamada, ao meu pai, Alberto Masashi Yamada, aos meus irmãos Adriano Shiroaki Yamada, Marjorie Amy Yamada e Eric Kenzo Yamada.

William Jun Yamada

Dedico à minha família que sempre me apoiou em todas as minhas escolhas.

Eduardo da Fonseca Pereira

Agradecimentos

Agradeço a todos meus professores que sempre estiveram dispostos a ajudar e a contribuir para um melhor aprendizado. Agradeço aos meus amigos que conheci durante o curso e que me apoiaram para chegar até aqui. Agradeço à minha família por me dar todas as condições necessárias para este momento.

Eduardo da Fonseca Pereira

Agradeço à minha família por todo apoio que deram. Aos meus amigos, com quem tive o prazer de compartilhar bons momentos, em especial a Eduardo Fonseca, Eduardo Moura e Luiz Gustavo. Ao professor Guilherme Caribé de Carvalho, por dar a oportunidade de realizar este projeto

William Jun Yamada

RESUMO

Este trabalho busca implementar o protocolo DeviceNet para a facilitar o gerenciamento de dados e informações de uma maquete ferroviária. Utilizando uma placa Raspberry Pi para processar as informações de entrada e saída, o protocolo CAN Bus pelo cabo fino que serve de meio físico para o transporte dos dados entre esta placa e o CLP, foi implementada a comunicação por meio do protocolo do DeviceNet. O código final foi testado por meio de uma comunicação CAN virtual e foi observado o comportamento esperado.

Palavras Chave: Protocolo CAN, CIP, Protocolo DeviceNet.

ABSTRACT

The main goal of this project is to implement the DeviceNet protocol to make the managment of input and output data from a railway model more easily doable. Using a Raspberry Pi board to process the I/O data and the CAN Bus Protocol by the thin cable as the physical medium to establish connection to the PLC using the DeviceNet module. After the code was finished it was tested using a virtual CAN and was obtained a satisfactory response, as the simulated input is replied with the expected behavior.

Keywords: CAN Protocol, CIP, DeviceNet Protocol.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Siglas

BCM	*Broadcom SOC Channel*
CAN	*Controller Area Network*
CI	Circuito Integrado
CID	*Connection ID*
CIP	*Common Industrial Protocol*
CLP	Controlador Lógico Programável
CRC	*Cyclic Redudancy Check*
CSMA/CD	*Carrier Sense Multiple Access with Collision Detection*
CPU	*Central Processing Unit*
DO	Saída Digital - *Digital Out*
DI	Entrada Digital - *Digital Input*
ECU	*Electronic Control Unit*
EDS	*Electronic Data Sheet*
GPIO	*General Purpose Input Output*
I/O	Entrada/Saída - *Input/Output*
IoT	*Internet of Things*
ISO	*International Organization for Standardization*
LAN	*Local Area Network*
MISO	*Master In Slave Out*
MOSI	*Master Out Slave In*
OSI	*Open Systems Interconnection*
ODVA	*Open DeviceNet Vendors Association*
PID	Proporcional Integral Derivativo
RTR	*Remote Time Request*
SAE	*Society of Automotive Engineers*
SLIO	*Serial Linked Input Output*
SoC	*System on Chip*
SPI	*Serial Peripheral Interface*
UCMM	*Unconnected Conection Message Manager*

Capítulo 1

Introdução

O presente capítulo apresenta sequencialmente aspectos relacionados à contextualização do projeto, definição do problema, objetivos gerais e metodologia empregada.

1.1 Contextualização

Historicamente, sabe-se que a tecnologia tem se desenvolvido muito desde a primeira revolução industrial, ocorrida no século XVIII, que foi um marco histórico pela criação das primeiras máquinas a vapor (figura 1.1), indústrias têxteis e de ferro. A mudança gerada por esta revolução acarretou em uma nova dinâmica econômica e social. Novos empregos surgiram, fábricas começaram a se expandir e as cidades cresceram em ritmos acelerados. A população, que até então era majoritariamente rural, começou a migrar para as cidades.

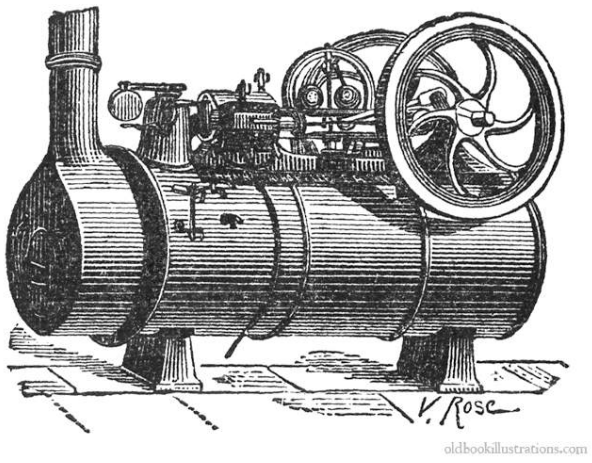

Figura 1.1: Máquina a vapor

A segunda revolução industrial, meados do século XIX, trouxe grandes inovações na manufatura. O uso da eletricidade, derivados do petróleo, aço, permitiram a produção em massa. Também foi uma era de novas invenções, como as ferrovias, lâmpadas, telégrafos, motor de combustão interna, entre outros.

A terceira revolução industrial, conhecida como revolução digital, trouxe avanços na eletrônica analógica e dispositivos mecânicos até a tecnologia digital que existe atualmente. Nesta revolução, que ainda encontra-se em andamento, já foram criados os computadores pessoais, internet, tecnologias de informações e comunicações.

Uma quarta revolução industrial está ocorrendo simultaneamente com a terceira e representa como a tecnologia pode se embarcar na sociedade e até no corpo humano. Esta revolução é marcada por grandes avanços na robótica, inteligência artificial, nanotecnologia, computação quântica, biotecnologia, internet das coisas (IoT), impressão 3D, veículos autônomos e pelo surgimento do conceito de indústria 4.0. A figura 1.2 ilustra o desenvolvimento da indústria ao decorrer das revoluções.

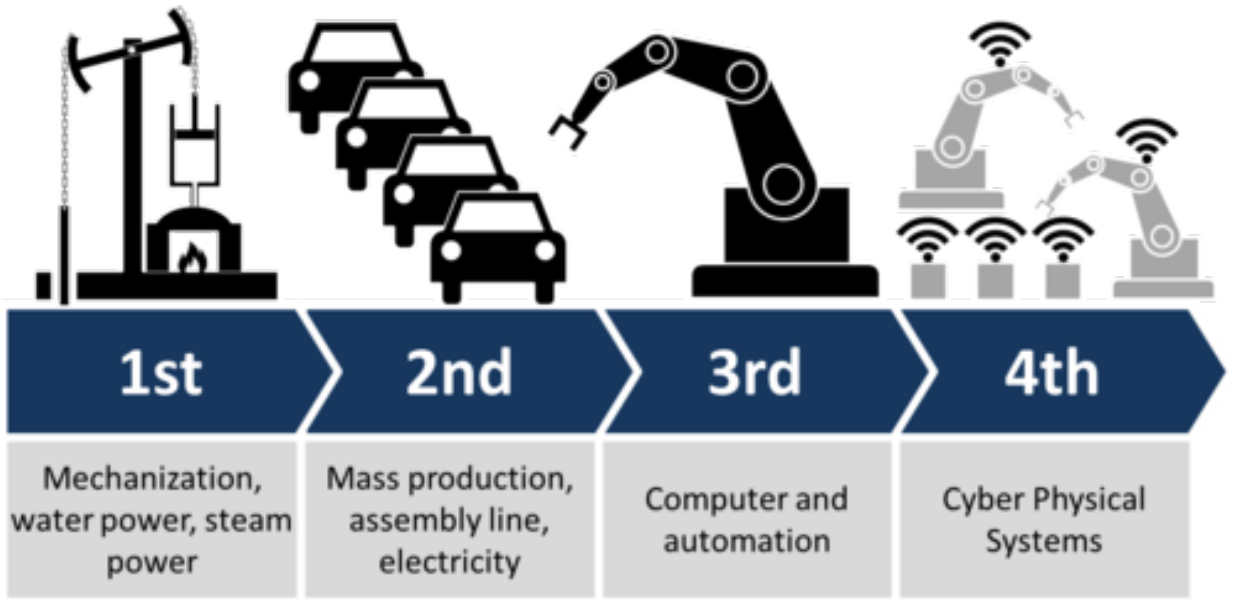

Figura 1.2: Inovações da manufatura

A indústria 4.0 está se tornando uma tendência da automação em tecnologias de manufatura. Ela facilita a visão e execução de "Fábricas Inteligentes"com as suas estruturas modulares, os sistemas ciber-físicos monitoram os processos físicos, criam uma cópia virtual do mundo físico e tomam decisões descentralizadas. Com a internet das coisas, os sistemas ciber-físicos comunicam e cooperam entre si e com os humanos em tempo real, e através da computação em nuvem, ambos os serviços internos e intra-organizacionais são oferecidos e utilizados pelos participantes da cadeia de valor.

Este conceito não fica restrito a área industrial. Ele vem se expandindo para várias outras áreas como a de redes ferroviárias, que utilizam sensores inteligentes e necessitam de redes que os suportem sem perdas de funcionalidades.

1.2 Definição do problema

A Universidade de Brasília possui, em suas dependências, um modelo de circuito de uma linha ferroviária que permite implementar sistemas de automação e controle monitorados por computador para

simular situações reais (figura 1.3).

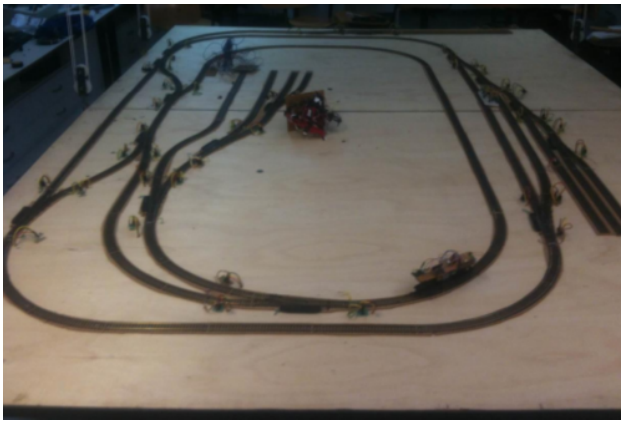

Figura 1.3: Modelo ferroviário desenvolvido no laboratório GRACO-UnB [1]

O modelo foi produzido em 2013 [1] e encontra-se em desuso, pois a forma de comunicação entre o sistema e o computador não faz uso de uma rede eficaz.

O problema original [7] dispunha de 40 I/O's, ou seja, para as entradas e saídas em uma CLP eram necessárias 40 portas digitais. A necessidade de uma grande quantidade de cabos para se conectarem as entradas do sistema ao módulo I/O do laboratório tornava inviável a manutenção e organização do modelo. O modelo também não podia ser expandido para sistemas de maior complexidade e simular contextos mais realísticos sem aumentar, consequentemente, a quantidade de cabos.

Uma solução parcial foi produzida [1] utilizando-se um circuito multiplexador de 8 bits. A necessidade de menos portas de entradas e saídas tornou o modelo ferroviário mais viável, porém, a programação exigia uma lógica complexa de decodificação dos 8 bits em linguagem *Ladder* ou *Sequential Function Chart*. Além disso, o uso de comunicação direta entre os sensores e atuadores com o CLP não permitiria o uso da inteligência dos mesmos, impedindo a adaptação para um sistema que adote o conceito de indústria 4.0.

A utilização de um protocolo de rede industrial neste projeto se tornou necessária, pois possibilitaria a redução de fios sem um aumento de complexidade de programação, permitindo entrada e saída de múltiplos dados na rede.

1.3 Objetivos do projeto

Com base no que foi visto acima ,este projeto visa tornar uma placa Raspberry Pi em um dispositivo de entrada e saída por meio de um protocolo industrial de rede de campo para realizar a interface entre o modelo de um sistema ferroviário com o controlador. Desta forma, será possível realizar a diminuição de fios sem aumento de complexidade da programação da CLP. Assim, o problema do modelo ferroviário será sanado permitindo realizar projetos mais sofisticados e estará se adequando a indústria 4.0, uma vez que a rede DeviceNet utiliza uma forma de protocolo que permite a comunicação de dispositivos inteligentes e permite a múltiplas entrada e saídas pela rede.

1.4 Apresentação do manuscrito

Este documento está subdividido em cinco capítulos, sendo o primeiro responsável por criar uma ambientação com a motivação para este trabalho, como também a definição do mesmo. O segundo capítulo explica brevemente os fundamentos básicos necessários para o entendimento dos processos que foram usados para resolver o problema proposto no capítulo um.

O terceiro capítulo explica como foi arquitetado a forma de alcançar o objetivo almejado, como foram usado os componentes e como foi gerado o código. Em seguida, o quarto capítulo expõe os resultados obtidos e o quinto, uma discussão sobre esses resultados e possíveis formas de melhoria dos mesmos.

Capítulo 2

Fundamentos

Neste capítulo serão abordados os conceitos necessários para a compreensão do projeto. Desta forma serão apresentados os conceitos de modelo OSI, FieldBus, CAN, CIP e DeviceNet.

2.1 Modelo OSI

O Modelo OSI (*Open System Interconnection*), criado em 1971 e formalizado em 1983, é um modelo de rede referência ISO dividido em camadas de funções. Seu objetivo é ser um padrão para protocolos de comunicação entre os mais diversos sistemas em uma rede, garantindo a comunicação entre dois sistemas computacionais.

Este modelo divide as redes em 7 camadas, de forma a se obter camadas de abstração. Cada protocolo implementa uma funcionalidade assinalada a uma determinada camada.

Esse modelo possui três conceitos essenciais:

1. Protocolo: este termo denota um conjunto de regras que governam na comunicação entre camadas de mesmo nível.
2. Serviço: representa qualquer serviço disponível de uma camada (provedor de serviços) para a camada adjacente (usuário de serviços).
3. Interface: especifica quais serviços estão disponíveis, como eles podem ser acessado, quais parâmetros são transferidos e resultados esperados.

2.1.1 Modelo de Camadas

O modelo OSI faz uma distinção clara entre comunicações horizontais e verticais. A figura 2.1 mostra a estrutura de camadas e como o dado é passado de um processo para outro. O sistema começa com a transmissão de dados através da camada 7 (camada de aplicação). A camada 7 prepara os dados e faz requisição de serviços à camada 6 (camada de apresentação). A camada 6 faz o mesmo: prepara os dados e requisita serviço à camada 5 (camada de sessão), e assim continua até a camada 1 (camada física). A

cada camada são adicionados cabeçalhos específicos aos dados específicos de cada protocolo. Ao chegar à camada física, o dado é, de fato, transmitido pelo meio físico.

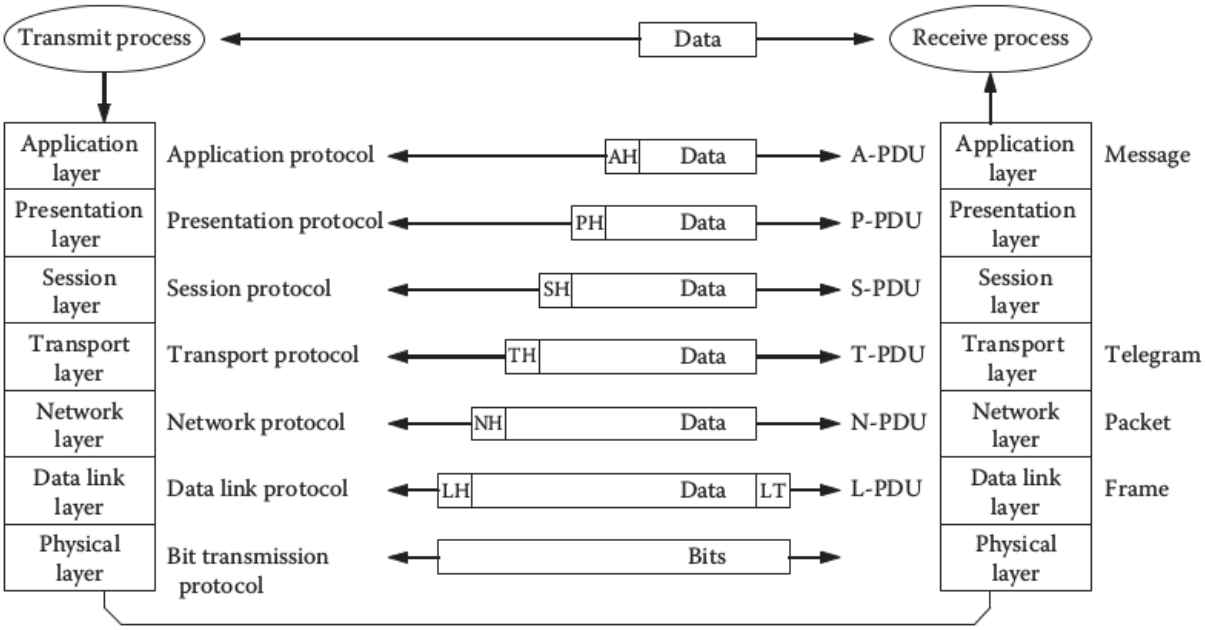

Figura 2.1: Estrutura de camadas do modelo OSI

Para melhor compreensão, uma breve descrição de cada camada:

1. **Camada física**: esta camada representa as propriedades mecânicas, elétricas, ótico, físico e lógico de um sistema de comunicação.
2. **Camada de enlace de dados**: realiza a pura conexão ponto-a-ponto entre duas redes com a tarefa de garantir a integridade dos dados. Nesta camada é produzido o data frame (estrutura dos dados) e são verificados erros.
3. **Camada de rede**: realiza roteamento de funções e também pode realizar a fragmentação e a remontagem.
4. **Camada de transporte**: responsável por receber os dados enviados pela camada de sessão e segmentá-los para que sejam enviados à camada de rede, que, por sua vez, transforma esses segmentos em pacotes. No receptor, a camada de Transporte realiza o processo inverso, ou seja, recebe os pacotes da camada de rede e junta os segmentos para enviar à camada de sessão.
5. **Camada de sessão**: responsável pela troca de dados e a comunicação entre hosts, a camada de Sessão permite que duas aplicações em computadores diferentes estabeleçam uma comunicação, definindo como será feita a transmissão de dados, colocando marcações nos dados que serão transmitidos.
6. **Camada de apresentação**: também chamada camada de Tradução, converte o formato do dado recebido pela camada de Aplicação em um formato comum a ser usado na transmissão deste dado, ou seja, um formato entendido pelo protocolo usado.

7. **Camada de aplicação**: é uma camada de inteface entre a unidade de comunicação e a aplicação real.

Por intermédio dessas definições é possível realizar comunicações complexas de forma estruturada.

2.2 FieldBus

Em meados de 1960, o sinal analógico de 4-20mA foi introduzido para realizar o controle de dispositivos industriais. Aproximadamente em 1980, os sensores inteligentes começaram a ser desenvolvidos e implementados usando um controle digital. Isso motivou a necessidade de integrar vários tipos de instrumentações digitais em campos de comunicações, visando otimizar a performance dos sistemas. Assim, tornou-se óbvio que um padrão era necessário para formalizar o controle dos dispositivos inteligentes.

Fieldbus é um termo genérico empregado para descrever tecnologias de comunicação industrial: abrange diferentes protocolos para redes industriais (como Modbus, Profibus, CAN, DeviceNet). Segundo a definição dada pela norma IEC 61158 [8]

"FieldBus é um barramento de dados digital, serial, multicomponentes, de dados para comunicação com controles industriais e dispositivos como, por exemplo, transdutores, atuadores e controladores locais".

Em uma hierarquia de redes, o FieldBus atua em nível de campo (nível de sensores e processos) como indicado na figura 2.2.

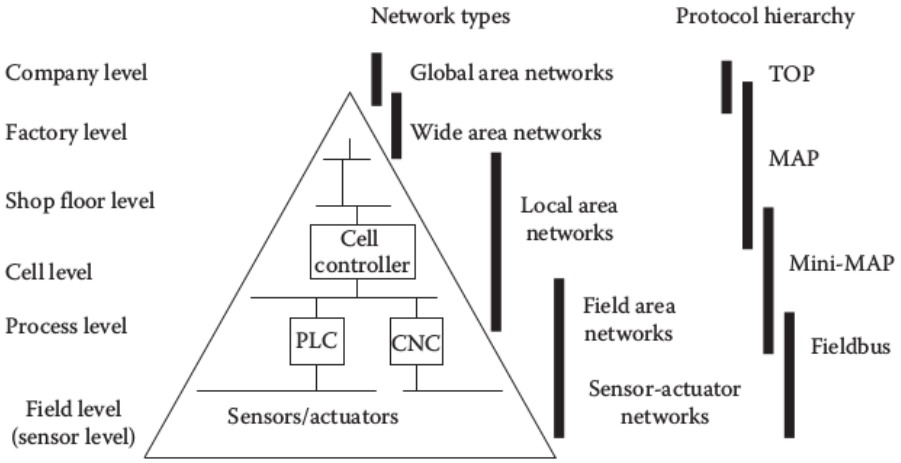

Figura 2.2: Nível hierárquico de redes de automação [2]

Apesar das LANs estarem cada vez mais presentes no cotidiano e estarem reduzindo a quantidade de níveis da hierarquia na automação. A LAN é uma rede de propósito geral, enquanto o Fieldbus é de propósito específico. Por sua alta especificidade, o protocolo de campo obtém melhores resultados em redes do mais baixo nível (que envolvem diretamente com sensores e atuadores).

Os protocolos fieldbus são modelados, em sua essência, utilizando modelo OSI. Porém, apenas as camadas 1, 2 e 7 são de fato utilizadas [9]. Desta forma, o modelo OSI fica reduzido à estrutura da figura

2.3. Algumas funcionalidades das camadas 3 a 6 ainda existem, porém, implementadas nas camadas 2 ou 7. O padrão IEC 61158 estabelece a versão de modelo de camadas do fieldbus.

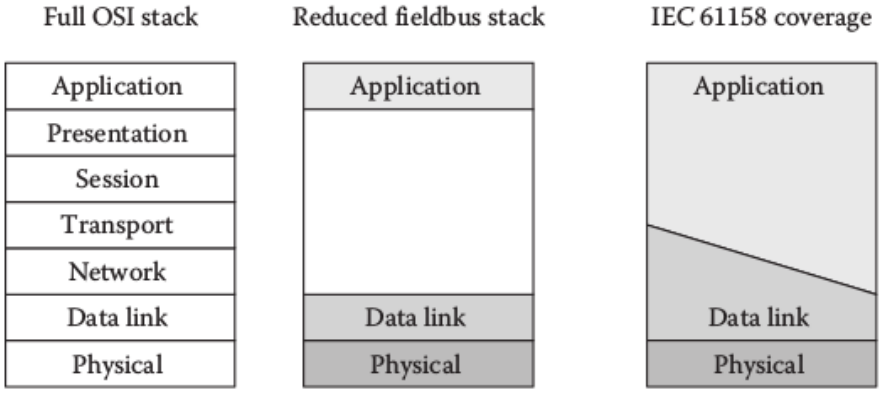

Figura 2.3: Estrutura de camadas do fieldbus definido por IEC 61158 [2]

As camadas de apresentação de um fieldbus devem ser compreensíveis. São indispensáveis para sistemas abertos e para gerar uma base grande interoperabilidade. Dessa forma, a programação de algumas abstrações de funções de certas aplicações pode ficar muito complexa. Por essa razão, alguns protocolos fieldbus não possuem a camada de apresentação. Um exemplo é o protocolo CAN, que serve como base para os protocolos CANopen, SDS e DeviceNet.

2.3 Protocolo CAN

O protocolo CAN foi desenvolvido por Robert Bosh em 1986, para aplicação na indústria autombilística, com o objetivo de simplificar os complexos sistemas de fios em veículos com sistemas de controle compostos por múltiplos microcontroladores/microcomputadores para gestão do motor, sistema ABS, controle da suspensão, etc. A sua especificação base anunciava elevada taxa de transmissão, grande imunidade a interferências eléctricas e capacidade de detectar erros.

O CAN implementa apenas as duas camadas inferiores do modelo OSI (camada física e camada de enlace de dados) e é regulamentado pela normas ISO11898, para aplicações de alta velocidade; ISO11519, para aplicações de baixa velocidade; e SAE J1939, para veículos pesados (caminhões e ônibus) [10].

O CAN é considerado um sistema de barramento série, bom para ligar em rede subsistemas inteligentes, tais como sensores e atuadores. A informação transmitida possui tamanho curto. Assim, cada mensagem CAN pode conter um máximo de 8 bytes de informação útil, sendo, no entanto, possível transmitir blocos maiores de dados recorrendo à segmentação.

2.3.1 Carrier Sense Multiple Access with Collision Detection (CSMA/CD)

O protocolo CAN é fundamentado no conceito CSMA/CD. CSMA significa *Carrier Sense Multple Access*, ou seja, cada nó da rede deve monitorar o barramento por um período de inatividade antes de

enviar uma mensagem (*Carrier Sense*). No período de inatividade, todos os nós têm mesma oportunidade para transmitir seus dados (*Multiple Access*). CD significa *Collision Detection*. Caso dois nós tentem acessar a rede ao mesmo tempo, será detectada uma colisão.

No CAN também é utilizada uma arbitragem de lógica binária não-destrutiva, o que permite que a mensagem não se corrompa, mesmo havendo colisão. Para definir essa arbitragem, define-se o estado dominante e o recessivo. O bit "0"é definido como bit dominante e "1"como recessivo. O bit dominante sempre terá prioridade sobre o barramento. Dessa forma, quando dois nós acessarem o barramento, um deles irá perder a arbitragem emitindo um bit recessivo em seu identificador de mensagem (que se encontra nos bits de arbitragem da mensagem). No momento em que perder a arbitragem, o nó com menor prioridade irá parar de transmitir os dados.

A comunicação é baseada em mensagens e é realizada em broadcast, de modo que todos os dispositivos dos nós recebem a mensagem e cabe a cada um verificar se é uma mensagem a ser processada e reconhecida ou não. Dessa forma, uma mesma mensagem também pode ser endereçada a apenas um nó ou múltiplos. Outro benefício é a possibilidade de inserir nós à rede sem precisar reprogramá-la: basta o nó ser capaz de processar as mensagens a ele direcionadas.

2.3.2 Data frame

O CAN possui quatro tipos de frame, mas o mais utilizado é o standard data frame, ilustrado na figura 2.4. Em seu primeiro bit, há um sinalizador de início de frame (*Start of Frame - SoF*) seguido pelo campo de arbitragem de 12 bits. No campo de artbitragem, encontram-se 11 bits de identificador de mensagem e um bit de RTR. O RTR é usado quando um nó necessita da informação de outro nó da rede.

Após o campo de arbitragem, existem 6 bits de controle, que definem qual o tipo de frame a ser usado e o tamanho do dado a ser transmitido na mensagem.

Em sequência, está o campo de dados, cujo tamanho é definido pelo campo de controle, podendo variar de 0 a 8 bytes.

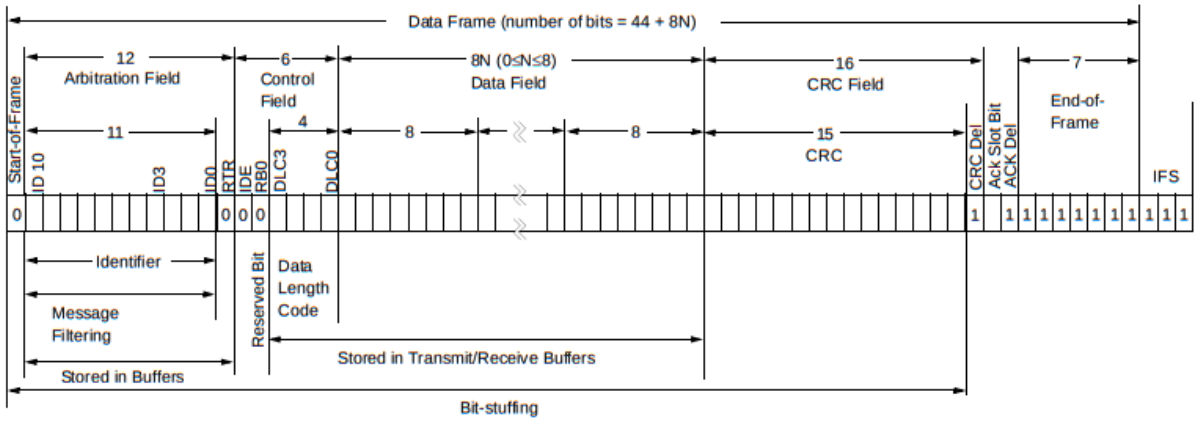

Figura 2.4: Standard CAN Data Frame [3]

Após o campo de dados, localiza-se o CRC. Este campo é utilizado para verificar a integridade da mensagem. Por fim, há 2 bits de reconhecimento e 7 bits para identificar o fim de frame. Entre uma mensagem e outra, existe um espaço entre frames preenchidos com bits recessivos.

2.3.3 Tratamento de erros

Cada nó é capaz de identificar falhas e transitar para outros estados de operação, como, por exemplo, um nó pode ir de modo normal para o modo *bus-off* (desligado da rede). São cinco condições de erros e três estados de erros no protocolo CAN. Além do mais, todos os nós com falhas não são capazes de acessar o barramento, para evitar que a rede seja derrubada.

2.3.3.1 Erros detectados

- *CRC Error* - o campo CRC no *data frame* contém um valor calculado por uma função *hash* com os dados da mensagem. Após transmitida a mensagem, todos os nós da rede recalculam o valor CRC da mensagem e verificam a integridade da mensagem. Caso o valor calculado pelos nós não coincida com o CRC da mensagem, um *error frame* é gerado. Como um nó não recebeu a sua devida mensagem por conta do erro, a mensagem é retransmitida após determinado tempo.
- *Acknowledge Error* - se os bits de reconhecimento não forem dominantes, significa que nenhum nó recebeu a mensagem. Portanto, um erro é detectado e produz-se um *error frame* e a mensagem retransmitida.
- *Form Error* - se for detectado um bit dominante no espaço entre *frames*, no *acknowledge delimiter* ou no fim do *frame*, o protocolo CAN reconhece como uma forma de violação gerando um *error frame*.
- *Bit Error* - quando um nó estiver transmitindo a mensagem e em seu próprio monitoramento do barramento identificar que enviou um bit errado, ocorre um bit *error* e a mensagem é retransmitida.
- *Stuff Error* - como o protocolo é assíncrono, é necessária uma lógica de *bit-stuffing* para sincronizar os clocks de todos os nós. Caso haja um erro de *bit-stuffing* o frame error é gerado.

2.3.3.2 Estados de erro

- *Error-Active* - é o modo de operação normal de um nó, em que a ele é permitido ler e escrever na rede.
- *Error-Passive* - ocorre se o contador de erros exceder 127. Quando um nó está nesse modo, ele deve aguardar pelo menos oito bits após o término de um frame para realizar a sua transmissão.
- *Bus-off* - ocorre se o contador de erros exceder 255. Quando um nó está nesse modo, ele não pode enviar nem receber dados da rede.

2.3.4 Controladores CAN

Existem duas versões de controladores CAN: uma com a designação de CAN básico, *Basic CAN*, e uma versão superior, possuindo um "filtro de aceitação" implementado em hardware, com a designação de CAN completo, *Full CAN*.

Na versão CAN básico, existe uma relação estreita entre o CPU e o controlador CAN, de modo que todas as mensagens existentes no barramento são verificadas individualmente pelo microcontrolador. Isso resulta na ocupação do CPU para verificar todas as mensagens, em vez de processar apenas as relevantes, o que tende a limitar o *baud rate* a 250 Kbit/s. Por outro lado, na versão CAN completo, a introdução de um filtro de aceitação permite ao controlador recusar mensagens irrelevantes, através da verificação dos identificadores, libertando o CPU para processar apenas mensagens consideradas relevantes, o que permite conseguir taxas de transmissão mais elevadas (1 Mbit/s).

Atualmente, existem CIs que implementam o controlador CAN. Há três configurações base no que diz respeito a utilização de integrados CAN: controlador CAN discreto (*Stand alone*), Microcontrolador com controlador CAN interno e *Serial Linked Input Output* (SLIO).

No primeiro caso, o controlador CAN é ligado ao microcontrolador através dos barramentos de dados e endereços. Esta configuração é útil quando se pretende atualizar sistemas já existentes. Implica-se que seja possível efetuar a interface do controlador. A interface de dados e endereços dos microcontroladores com controlador CAN interno são feitas internamente. Os integrados SLIO detêm pouca "inteligência", necessitando serem programados e calibrados por um microcontrolador.

2.3.4.1 Controlador MCP2515

O MCP2515 é um controlador CAN *stand-alone* desenvolvido pela Microchip [4] que implementa a especificação CAN 2.0B com uma velocidade de transmissão de até 1 Mb/s. A interface com um microcontrolador é realizada por SPI. A figura 2.5 ilustra o empacotamento do controlador.

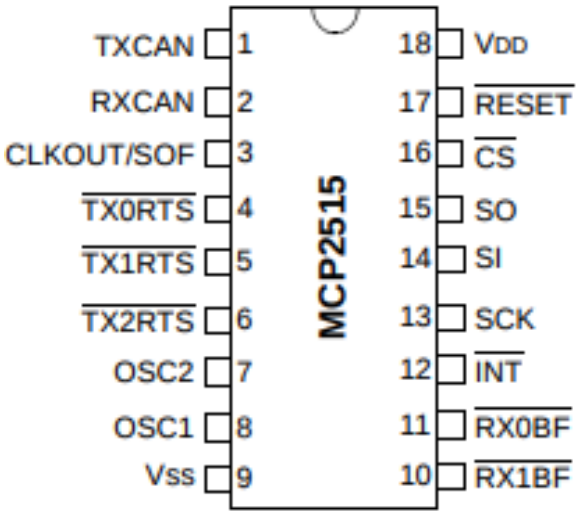

Figura 2.5: SoC do MCP2515 com empacotamento de 18 pinos[4]

Este dispositivo já realiza a padronização do CAN, além de realizar a sincronização com a rede e detecção de erros. As descrições de sua pinagem são realizadas na tabela 2.1

Name	PDIP/SOIC Pin #	TSSOP Pin #	I/O/P Type	Description	Alternate Pin Function
TXCAN	1	1	O	Transmit output pin to CAN bus	—
RXCAN	2	2	I	Receive input pin from CAN bus	—
CLKOUT	3	3	O	Clock output pin with programmable prescaler	Start-of-Frame signal
$\overline{\text{TX0RTS}}$	4	4	I	Transmit buffer TXB0 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
$\overline{\text{TX1RTS}}$	5	5	I	Transmit buffer TXB1 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
$\overline{\text{TX2RTS}}$	6	7	I	Transmit buffer TXB2 request-to-send. 100 kΩ internal pull-up to VDD	General purpose digital input. 100 kΩ internal pull-up to VDD
OSC2	7	8	O	Oscillator output	—
OSC1	8	9	I	Oscillator input	External clock input
VSS	9	10	P	Ground reference for logic and I/O pins	—
$\overline{\text{RX1BF}}$	10	11	O	Receive buffer RXB1 interrupt pin or general purpose digital output	General purpose digital output
$\overline{\text{RX0BF}}$	11	12	O	Receive buffer RXB0 interrupt pin or general purpose digital output	General purpose digital output
$\overline{\text{INT}}$	12	13	O	Interrupt output pin	—
SCK	13	14	I	Clock input pin for SPI™ interface	—
SI	14	16	I	Data input pin for SPI interface	—
SO	15	17	O	Data output pin for SPI interface	—
$\overline{\text{CS}}$	16	18	I	Chip select input pin for SPI interface	—
$\overline{\text{RESET}}$	17	19	I	Active low device reset input	—
VDD	18	20	P	Positive supply for logic and I/O pins	—
NC	—	6,15	—	No internal connection	

Note: Type Identification: I = Input; O = Output; P = Power

Tabela 2.1: Pinout do MCP2515[4]

2.3.5 Transceiver MCP2551

O MCP2551 é um dispositivo CAN de alta velocidade que serve como interface entre o controlador CAN e o barramento físico. Ele é compatível com a norma ISO-11898 e opera até 1 Mb/s. A figura 2.6 é um empacotamento do *transceiver*.

O barramento CAN possui dois estados: Dominante e Recessivo. Dominante é quando a tensão diferencial entre o CANH e o CANL é maior que um valor pré-estabelecido. Recessivo é quando a tensão diferencial é menor que outro valor pré-estabelecido. O dominante e recessivo correspondem ao nível baixo e alto no TXD, respectivamente.

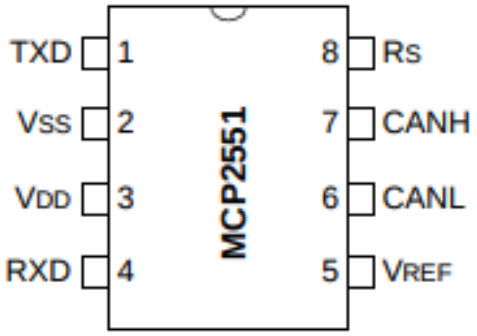

Figura 2.6: SoC do MCP2551

O dispositivo pode trabalhar em três modos de operação: *high-speed* (caso o Rs seja conectado diretamente ao *ground*), *slope control* (caso haja uma resistência entre o *ground* e o Rs, controlando o *slew-rate* do *transceiver*) e o *standby* (caso conecte o Rs ao Vdd).

As descrições de sua pinagem são realizadas na tabela 2.2

Pin Number	Pin Name	Pin Function
1	TXD	Transmit Data Input
2	Vss	Ground
3	VDD	Supply Voltage
4	RXD	Receive Data Output
5	VREF	Reference Output Voltage
6	CANL	CAN Low-Level Voltage I/O
7	CANH	CAN High-Level Voltage I/O
8	Rs	Slope-Control Input

Tabela 2.2: Pinout do MCP2551

2.4 Common Industrial Protocol - CIP

O *Common Industrial Protocol* é um protocolo de comunicação de redes industriais *peer-to-peer* que provê conexões entre dispositivos industriais (sensores, atuadores) e dispositivos de alto nível (CLP) [5]. Este protocolo possui uma grande versatibilidade, tendo sido integrado em vários outros protocolos de camada de aplicação como o EtherNet/IP™, DeviceNet™, ControlNet™, e CompoNet™. A figura 2.7 mostra a relação das principais adaptações do CIP.

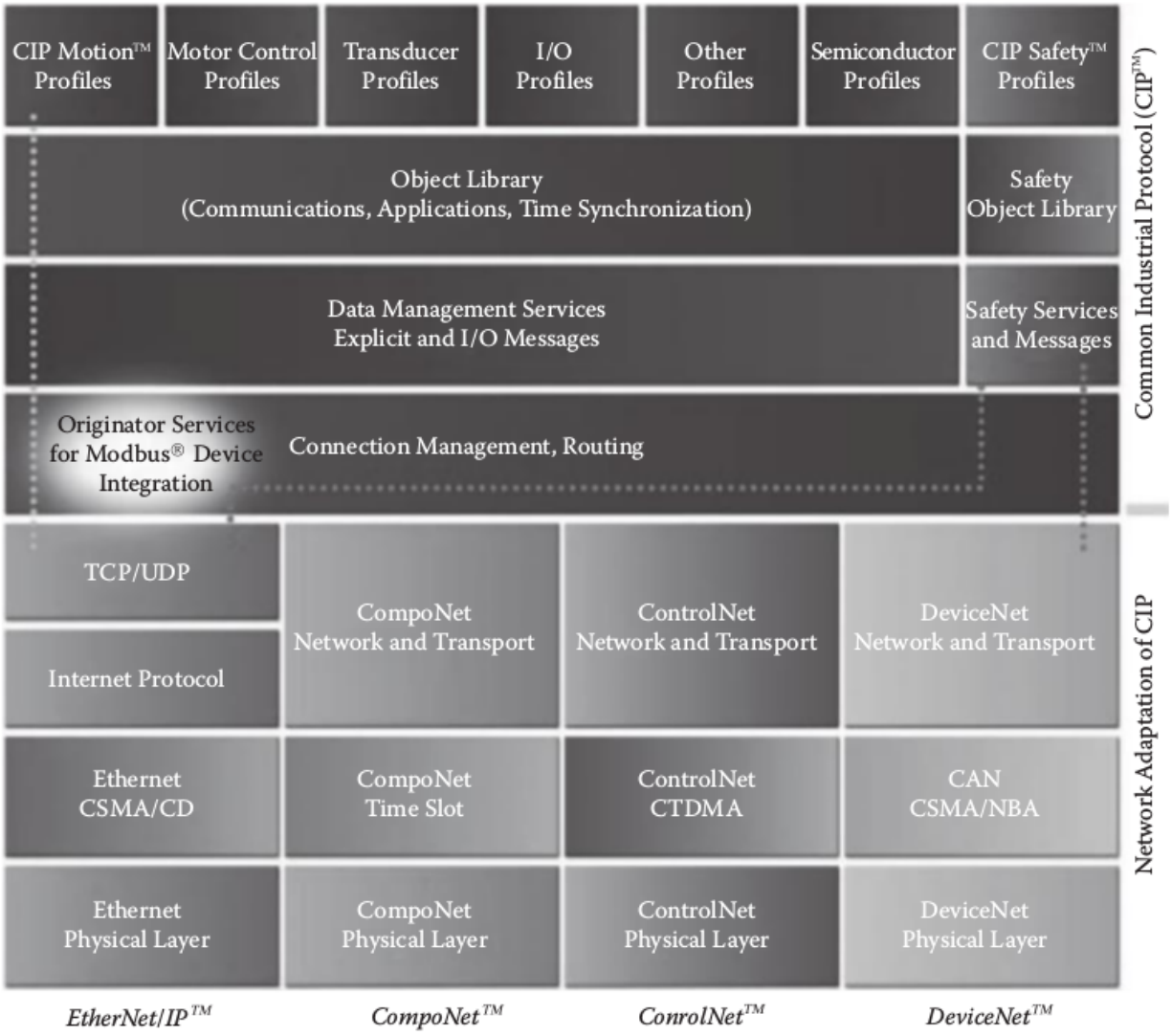

Figura 2.7: O *Common Industria Protocol* e suas principais adaptações [2]

Alguns termos importantes para a compreensão do CIP são

- *Cliente* - dentro de um modelo cliente/servidor, o cliente é o dispositivo que faz uma requisição de serviço ao servidor. O cliente espera uma resposta do servidor.
- *Servidor* - dentro de um modelo cliente/servidor, o servidor é o dispositivo que presta um serviço ao cliente. O servidor retorna uma resposta ao cliente.
- *Produtor* - dentro do modelo produtor/consumidor, o produtor é aquele que insere uma mensagem na rede para um ou mais consumidores.
- *Consumidor* - dentro do modelo produtor/consumidor, o consumidor é aquele que coleta uma mensagem da rede.
- *Modelo Produtor/Consumidor* - é um modelo *multicast*, no qual os nós definem se irão consumir a mensagem na rede dependendo do *connection ID* (CID) no pacote de dados.

- *Mensagem Explícita* - é uma mensagem que contém uma informação endereçamento e serviço que será direcionada a um dispositivo que irá realizar algum serviço oferecido por um de seus componentes.
- *Mensagem (I/O) Implícita* - é uma mensagem que os nós consumidores já sabem o que fazer com o dado baseado no CID da mensagem. Em geral, são utilizados para transportar dados de I/O.

2.4.1 Modelagem de objetos

Este protocolo tem como base a modelagem em objetos. Portanto, todo nó de uma rede é modelado como uma coleção de objetos que fará a abstração de cada componente do dispositivo. Os objetos CIP são estruturados em classe, instâncias e atributos.

Uma classe representa um conjunto de objetos que pertencem ao mesmo tipo de componente do sistema. Uma instância de um objeto é a representação de um objeto particular em uma classe. Todas instâncias de uma classe possuem o mesmo conjunto de atributos, porém, cada um tem seus particulares valores de cada atributo. A figura 2.8 ilustra as múltiplas instâncias de objeto em uma classe que residem em um nó.Cada classe também pode conter atributos que a descrevam.

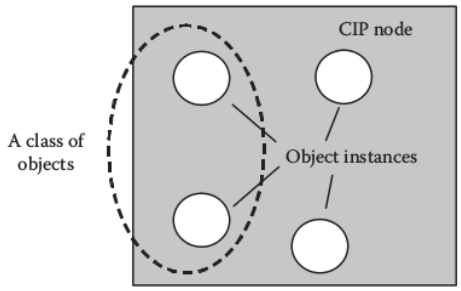

Figura 2.8: Uma classe de objetos [2]

Todos objetos e seus componentes são endereçados por um sistema de endereçamento uniforme que consiste em:

- *Media Access Control Identifier (MAC ID)* - identificação para cada nó da rede
- *Class Identifies (Class ID)* - endereço atribuído para cada classe de objeto acessível na rede
- *Instance Identifier (Instance ID)* - endereço para cada instância de objeto em uma classe
- *Attribute Identifier (Attribute ID)* - endereço dos atributos de uma classe ou instância
- *Service Code* - denota a função da instância de um objeto ou de uma classe de objeto

2.4.2 Serviços

Códigos de serviços são utilizados para definir uma ação que foi requisitada através de uma mensagem explícita. Além de serviços de leitura e escrita de dados, o CIP estabelece vários serviços de natureza

similares que podem ser utilizados por qualquer rede CIP e são úteis para vários objetos. Há código de serviço que é específico de um objeto, portanto, esse mesmo código pode ser utilizado por outro objeto para identificar um serviço distinto. Por fim, também se pode fazer um serviço específico de acordo com os requisitos de um produto a ser desenvolvido. A criação de serviços oferece uma grande flexibilidade, entretanto, o desenvolvedor necessita realizar uma documentação para futuros usos, uma vez que não é um serviço universal.

2.4.3 Protocolo de mensagens

CIP é uma rede baseada em conexão. E a toda conexão realizada são atribuídas CIDs (*Connection ID*). Se a comunicação é bidirecional, são necessárias duas CIDs. Existem dois objetos de conexão: Conexão I/O (para comunicação de propósito especial também conhecido como conexão implícita) e conexão de mensagens explícitas (de propósito múltiplo/genérico). As mensagens explícitas têm uma estrutura pré-definida pelo protocolo de bits, enquanto mensagens do tipo I/O não possuem uma estrutura pré-definida: cabe ao desenvolvedor definir a interpretação destas.

Para se estabelecer uma conexão, pode-se utilizar a função do *Unconnected Message Manager* (UCMM) que oferece o serviço de *Forward_open* que estabelece uma conexão. Existem outras formas de se estabelecer a conexão além do UCMM, um exemplo é fazer uso de comunicações baseadas em mensagens pré-estabelecidas em conjuntos Mestre-Escravo, que será descrito mais à frente.

2.4.4 Objetos de comunicação

Os objetos de comunicação gerenciam e providenciam o tempo de troca de mensagens. Cada objeto de comunicação possui uma parte de *link consumer*, ou uma de *link producer* ou ambos. As figuras 2.9 e 2.10 ilustram as conexões do tipo I/O e explícitas, respectivamente. Os atributos dos objetos de conexão estabelecem o maior tamanho de dado (em bytes) que pode ser consumido ou produzido, endereço de destino, tempo de resposta esperado, entre outros.

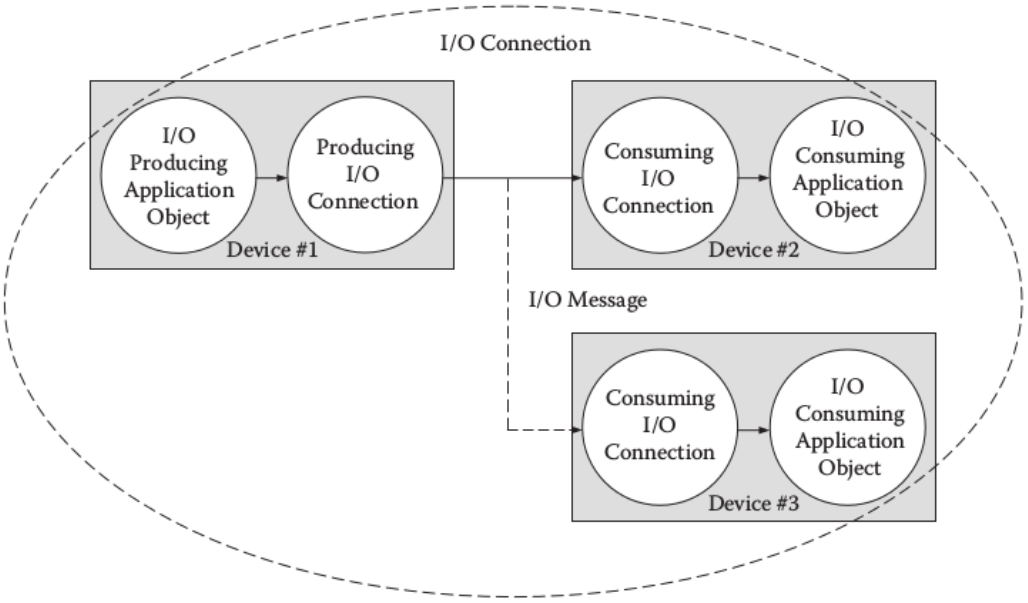

Figura 2.9: Conexão I/O [5]

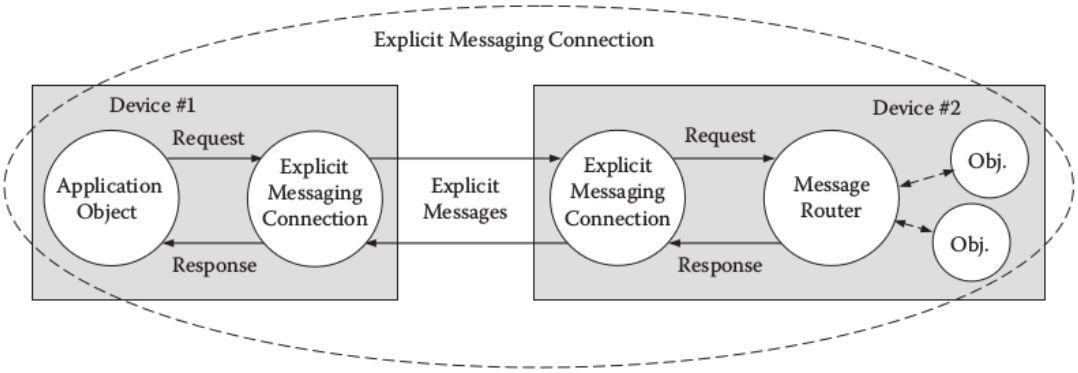

Figura 2.10: Conexão Explícita [5]

2.4.5 Biblioteca de objetos

O CIP possui uma coleção de objetos já definidos. Pode-se classificar os objetos em três grupos: de uso geral, de aplicação específica e de rede especifica.

Os objetos de uso geral são definidos na tabela 2.3 (os valores em hexadecimal entre parênteses corresponde ao Class ID):

Assembly (0x04)	Message Router (0x02)
Acknowledge Handler (0x2B)	Originator Connection List (0x45)
Connection (0x05)	Parameter (0x0F)
Connection Configuration (0xF3)	Parameter Group (0x10)
Connection Manager (0x06)	Port (0xF4)
File (0x37)	Register (0x07)
Identity (0x01)	Selection (0x2E)

Tabela 2.3: Objetos de uso geral definidos pelo CIP

Os objetos de aplicação específica são listados na tabela 2.4

AC/DC Drive (0x2A)	Analog Group (0x22)
Analog Input Group (0x20)	Analog Input Point (0x0A)
Analog Output Group (0x21)	Analog Output Point (0x0B)
Base Energy (0x4E)	Block Sequencer (0x26)
Command Block (0x27)	Control Supervisor (0x29)
Discrete Group (0x1F)	Discrete Input Group (0x1D)
Discrete Output Group (0x1E)	Discrete Input Point (0x08)
Discrete Output Point (0x09)	Electrical Energy (0x4F)
Event Log (0x41)	Group (0x12)
Motion Device Axis (0x42)	Motor Data (0x28)
Nonelectrical Energy (0x50)	Overload (0x2C)
Position Controller (0x25)	Position Controller Supervisor (0x24)
Position Sensor (0x23)	Presence Sensing (0x0E)
S-Analog Actuator (0x32)	S-Analog Sensor (0x31)
S-Device Supervisor (0x30)	S-Gas Calibration (0x34)
S-Partial Pressure (0x38)	S-Sensor Calibration (0x40)
S-Single Stage Controller (0x33)	Safety Analog Input Group (0x4A)
Safety Analog Input Point (0x49)	Safety Discrete Input Group (0x3E)
Safety Discrete Input Point (0x3D)	Safety Discrete Output Group (0x3C)
Safety Discrete Output Point (0x3B)	Safety Dual Channel Analog Input (0x4B)
Safety Dual Channel Output (0x3F)	Safety Supervisor (0x39)
Safety Validator (0x3A)	Softstart (0x2D)
Target Connection List (0x4D)	Time Sync (0x43)
Trip Point (0x35)	

Tabela 2.4: Objetos de aplicação específica definidos pelo CIP

Por fim, os objetos de rede específica são listados na tabela 2.5

Base Switch (0x51)	CompoNet Link (0xF7)
CompoNet Repeater (0xF8)	ControlNet (0xF0)
ControlNet Keeper (0xF1)	ControlNet Scheduling (0xF2)
Device Level Ring (DLR) (0x47)	DeviceNet (0x03)
Ethernet Link (0xF6)	Modbus (0x44)
Modbus Serial Link (0x46)	Parallel Redundancy Protocol (0x56)
Power Management (0x53)	PRP Nodes Table (0x57)
SERCOS III Link (0x4C)	SNMP (0x52)
QoS (0x48)	RSTP Bridge (0x54)
RSTP Port (0x55)	TCP/IP Interface (0xF5)

Tabela 2.5: Objetos de rede definidos pelo CIP

Todo dispositivo CIP possui no mínimo um objeto de *Identity*, um objeto *Connection* ou *Connection Manager*, um *Message Router*, um objeto de rede específica. Outros objetos são adicionados para as funcionalidades desejadas para o dispositivo.

2.4.6 Configuração e EDS (*Electronic Data Sheets*)

O CIP oferece diversas formas de configurar dispositivos:

- *Data sheet* impresso
- Objeto de parâmetros
- Arquivo EDS
- Combinação de EDS com objeto de parâmetros

O *data sheet* impresso não é muito eficaz, pois necessita que o usuário adicione manualmente as informações do dispositivo. Esse método não disponibiliza o contexto, conteúdo nem o formato dos dados.

O objeto de parâmetros disponibiliza todos os possíveis dados de configuração e a configuração é realizada de forma automática pela ferramenta de configuração. Porém, para projetos de pequeno porte, detalhes de todos os parâmetros não são necessários, tornando-se um inconveniente para o desenvolvedor.

O arquivo EDS supre todas as informações necessárias do dispositivo, sem sobrecarregar de informações. Este é um arquivo que pode ser gerado em qualquer editor de texto ASCII, entretanto, há ferramentas de edição específicas para produzir EDS's para facilitar a produção do arquivo de configuração.

As regras estabelecidas pelo CIP ditam que os arquivos EDS deverão estar divididos em determinadas seções, que são identificadas por estarem entre colchetes []. As seções são:

- *File*: Descreve o conteúdo e a revisão do arquivo.

- *Device*: Usado para identificar o dispositivo.
- *Device Classification*: Descreve a quais redes ele pode ser conectado.
- *ParamClass*: Descreve as configurações além dos parâmetros das classes.
- *Params*: Identifica todas as configurações de cada dispositivo.
- *Groups*: Identifica todos os grupos de parâmetros do dispositivo e lista o nome do grupo e número dos parâmetros.
- *Assembly*: Descreve a estrutura dos dados.
- *Connection Manager*: Descreve as conexões que o dispositivo tem suporte.
- *Connection ManagerN*: O mesmo da seção [Connection Manager], porém para apenas algumas portas.
- *Port*: Descreve as várias portas de rede que o dispositivo pode se conectar.
- *Capacity*: Descreve a capacidade de comunicação das redes ControlNet e Ethernet/IP.
- *Connection Configuration*: define o *Connection Configuration Object* caso tenha sido implementado.
- *Event Enumeration*: Associa um evento ou código de status dentro de um dispositivo com uma *string*.
- *Symbolic Translation*: Traduz uma *string* simbólica para o valor real.
- *Internationalization*: Disponibilizar uma *string* em diversas linguagens.
- Modular: Descreve a estrutura modular interna do dispositivo.
- IO_Info: Descreve o método de conexão I/O e o tamanho do dado de I/O. Permitido apenas para o DeviceNet.
- *Variant_IO_Info*: Descreve múltiplos IO_Info. Permitido apenas para o DeviceNet.
- EnumPar: Enumera uma lista de parâmetros de escolha para o usuário. Permitido apenas para o DeviceNet.
- Seções *Object Class*: Descreve os detalhes de cada classe de objetos.

Uma ferramenta com uma coleção de arquivos EDS usa a seção [Device] e tenta fazer a correspondência com todos os nós encontrados na rede, permitindo a interface entre o dispositivo e a ferramenta utilizada.

2.4.7 Gerenciamento de dados

O gerenciamento de dados (descrito no apêndice C de [5]) descreve o modelo de endereçamento para entidades CIP e as estruturas de dados das próprias entidades.

O modelo de gerenciamento é realizado em segmentos, que podem ser do tipo

- *Port Segment* - usado para rotear de uma sub-rede à outra
- *Logical Segment* - informação de referência lógica (endereço de classe/instância/atributos)
- *Network Segment* - especifica parâmetros de rede necessário para transmitir para outras redes.
- *Symbolic Segment* - nomes simbólicos
- *Data Segment* - Dado embarcado (como, por exemplo, dados de configuração)

Entre esses, os mais importantes são o *Logical Segment* e o *Data Segment*, os quais serão melhor detalhados.

2.4.7.1 Logical Segment

O *Logical Segment*, que possuí em seu primeiro byte valores entre 0x20 e 0x3F, pode ser utilizado para endereçar objetos e seus atributos em um dispositivo. A estrutura típica é colocada na sequência [Class ID] [Instance ID] [Attribute ID] (segundo [2]).

Esse tipo de endereçamento é utilizado tipicamente para endereçar objetos *Assembly*, *Parameter*, entre outros endereçáveis. É utilizado de forma extensa em arquivos EDS e mensagens explícitas.

2.4.7.2 Data Segment

O *Data Segment* providencia um mecanismo para entregar dados para uma aplicação. Seu primeiro byte tem valores entre 0x80 e 0x9F. Os dados podem ser estruturados ou elementares e cada um tem um tipo de codificação que segue os requerimentos da norma IEC 61131-3 [8]. Para o contexto deste projeto os dados elementares são os mais relevantes, pois são estes que são utilizados para especificar parâmetros do arquivo EDS. Os tipos mais relevantes são

- 1 bit
  - Boolean, BOOL, Código 0xC1
- 1 byte
  - Bit string, 8 bits, BYTE, Código 0xD1
  - Unsigned 8-bit integer, USINT, Código 0xC6
  - Signed 8-bit integer, SINT, Código 0xC2

- 2 bytes
  - Bit string, 16 bits, WORD, Código 0xD2
  - Unsigned 16-bit integer, UINT, Código 0xC7
  - Signed 16-bit integer, INT, Código 0xC3
- 4 bytes
  - Bit string, 32 bits, DWORD, Código 0xD3
  - Unsigned 32-bit integer, UDINT, Código 0xC8
  - Signed 32-bit integer, DINT, Código 0xC4

2.5 DeviceNet

DeviceNet é uma rede de campo (*fieldbus*) nível que estabelece a conexão entre dispositivos industriais simples (sensores, atuadores) e dispositivos de alto nível (controladores). DeviceNet é baseado no *Common Industrial Protocol* com algumas adaptações e também utiliza o protocolo CAN. A figura 2.11 ilustra um exemplo de rede DeviceNet.

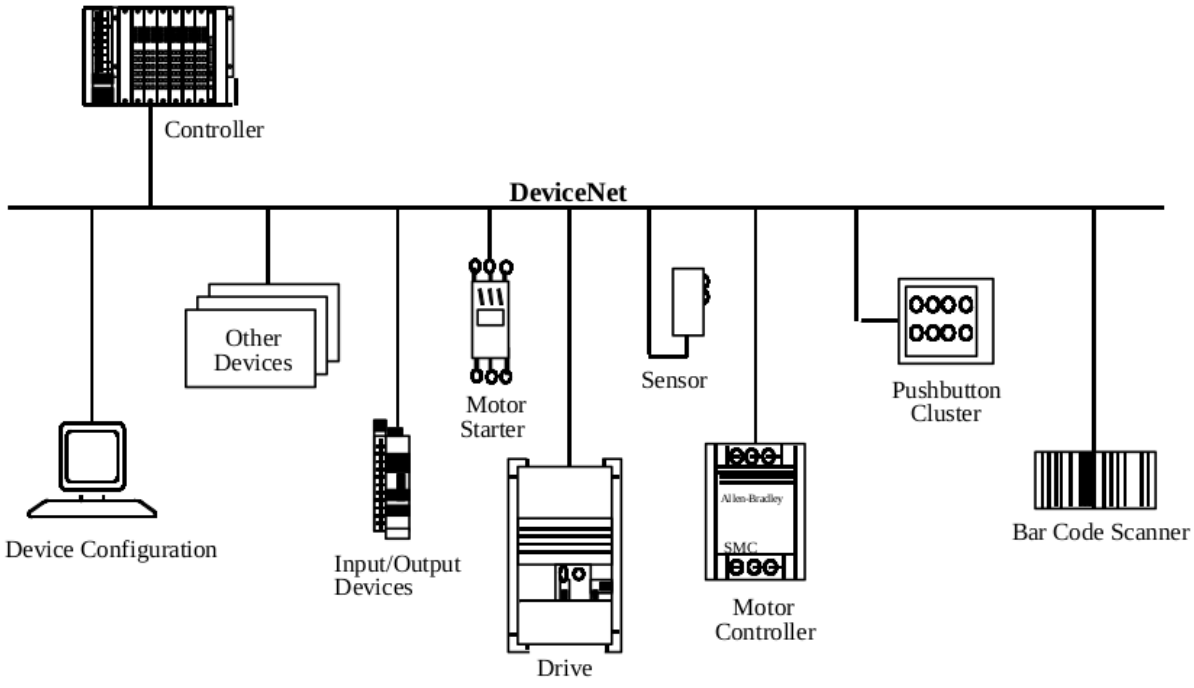

Figura 2.11: Exemplo de rede DeviceNet [3]

DeviceNet providencia uma solução de baixo custo de transporte de informações orientadas para controle a dispositivos de rede de baixo nível, acesso à inteligência presente nos dispositivos e capabilidade de interações mestre/escravo ou *peer-to-peer*.

2.5.0.1 Características do DeviceNet

- Configuração *Trunkline-dropline* (Tronco-derivação)
- Suporte até 64 nós
- Remoção de nós sem prejudicar a rede
- Suporte para sensores e atuadores
- Proteção contra erros na fiação
- Seleção da taxa de dados (125k baud, 250k baud, e 500k baud), vide tabela 2.6

Data Rate	Trunk Distance	Drop Length	
		Maximum	Cumulative
125k baud	500 meters (1640 ft.)	6 meters (20 ft.)	156 meters (512 ft.)
250k baud	250 meters (820 ft.)		78 meters (256 ft.)
500k baud	100 meters (328 ft.)		39 meters (128 ft.)

Tabela 2.6: Tabela de taxa de transmissão por queda de tensão [3]

- Alimentação configurável
- Suporta altas correntes (até 16 Ampères)
- Proteção de sobrecarga embutida
- Uso do *Controller Area Network* (CAN) para controle de acesso de mídia e meio físico de transporte de sinais
- Modelo baseado em conexões
- Fragmentação para mover grandes informações
- Detecção de MAC ID duplicado

2.5.0.2 Modelo de Objeto do DeviceNet

A figura 2.12 ilustra um modelo abstrato de um objeto de um produto DeviceNet.

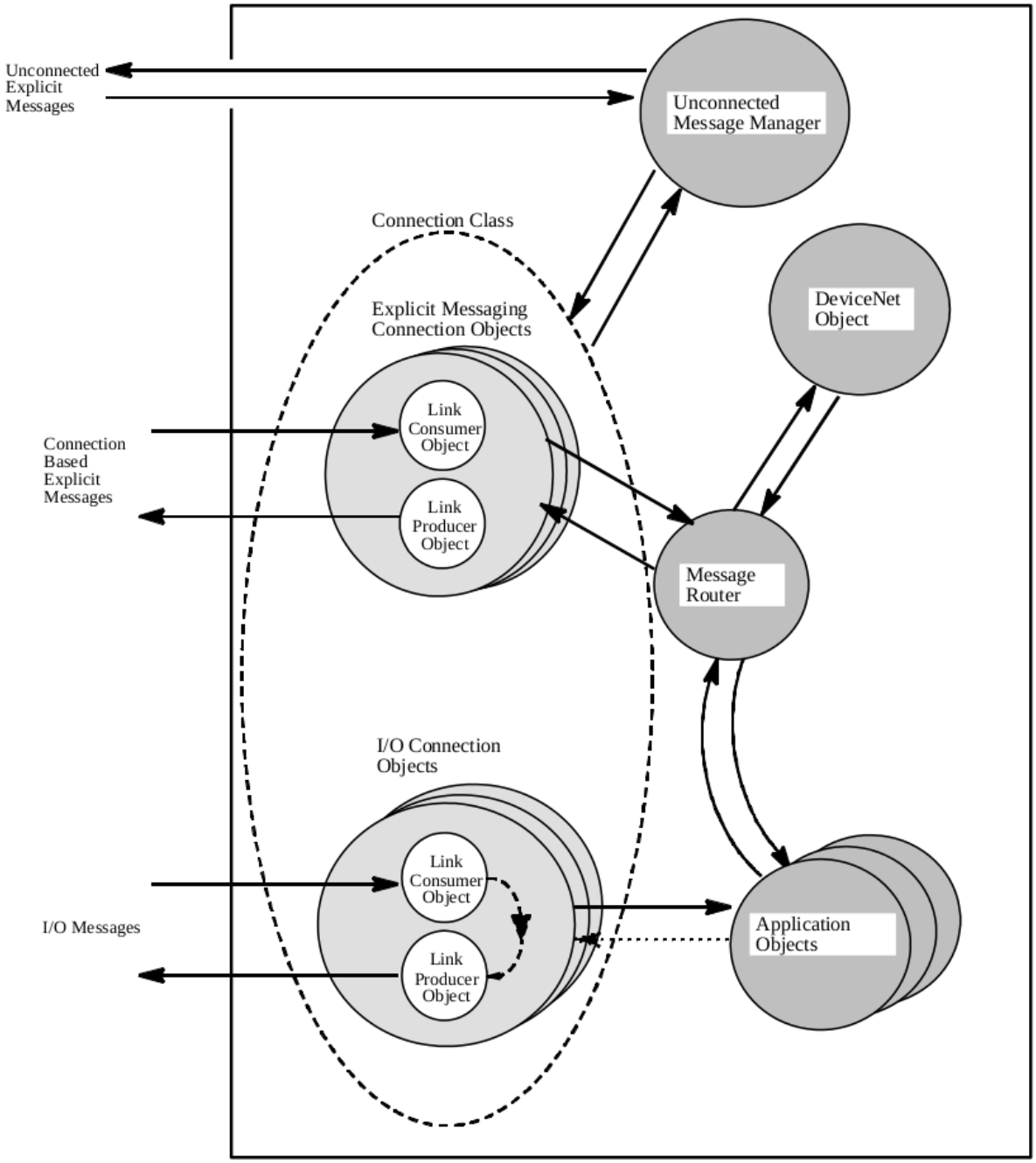

Figura 2.12: Modelo Abstrato de um objeto DeviceNet [3]

Os seguintes componentes são incluídos:

- *Unconnected Message Manager* (UCMM) - Processa conexões explícitas não conectadas
- *Connection Class* - Aloca e gerencia recursos internos
- *Connection Object* - Gerencia uma conexão em específico
- *DeviceNet Object* - Providencia a configuração e status da rede física do DeviceNet

- *Link Producer Object* - Usado por um *Connection Object* para transmitir dados
- *Link Receiver Object* - Usado por um *Connection Object* para receber dados
- *Message Router* - Distribui a requisição explícita de mensagens para o objeto apropriado
- *Applications Objects* - Implementa o propósito do produto

2.5.1 Protocolo de Mensagens do DeviceNet

Uma conexão DeviceNet provê um caminho entre múltiplas aplicações. Quando uma conexão é estabelecida, as transmissões acerca desta conexão são atribuídas a um *Connection ID* (CID). Se a conexão envolve uma troca bidirecional, então, dois CIDs são atribuídos, como se observa no exemplo da figura 2.13.

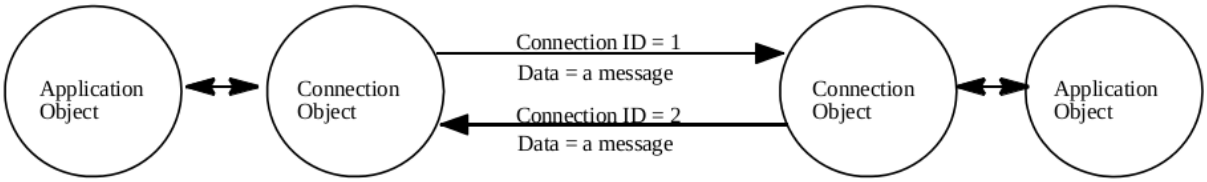

Figura 2.13: Conexões e CIDs [3]

2.5.1.1 Campo de Identificação do CAN

A estrutura padrão do CAN que é utilizada pelo protocolo DeviceNet é ilustrado na figura 2.4

O DeviceNet possui 11 bits de identificação CAN que subdivide-se em quatro grupos, conforme a tabela a seguir

IDENTIFIER BITS											HEX RANGE	IDENTITY USAGE
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
0	Group 1 Message ID				Source MAC ID						000 – 3ff	Message Group 1
1	0	MAC ID						Group 2 Message ID			400 – 5ff	Message Group 2
1	1	Group 3 Message ID			Source MAC ID						600 – 7bf	Message Group 3
1	1	1	1	1	Group 4 Message ID (0 – 2f)						7c0 – 7ef	Message Group 4
1	1	1	1	1	1	1	X	X	X	X	7f0 – 7ff	Invalid CAN Identifiers
10	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**		

Tabela 2.7: Campo de identificação do CAN [3]

- *Message ID* - Identifica a mensagem dentro do Grupo de Mensagens. O *Message ID* em conjunto com o MAC ID gera o *Connection ID*.

- *Source MAC ID* - MAC ID atribuído ao nó transmissor. Os grupos 1 e 3 precisam deste ID.
- *Destination MAC ID* - MAC ID atribuído ao dispositivo receptor da mensagem. O grupo 2 pode utilizar tanto o *Source* quanto o *Destination ID*.
- Grupos de Mensagens: Os grupos 1 e 3 providenciam uma solução a prioridade de acesso ao barramento, não sendo apenas baseado no MAC ID do nó, e distribuído assim que possível. O grupo 2 providencia uma solução que leva em consideração o limite de filtração de múltiplos chips CAN. O grupo 4 é o conjunto de conexões Offline. A ordem de prioridades é: primeiro o grupo 1, depois o grupo 2 e por fim o grupo 3 (o grupo 4 não necessita).

É importante ressaltar que as *Message ID's* 6 e 7 do grupo 2 são reservadas pelo DeviceNet para o gerenciamento de conexões Mestre/Escravo predefinidas e a checagem de *MAC ID's* duplicadas respectivamente. Enquanto as *Message ID's* 5 e 6 do grupo 3 também são reservadas para mensagens de resposta e requisição explicitas ainda não conectadas, enquanto a *Message ID* 7 para este grupo é invalida e não utilizada.

2.5.1.2 Máquina de Estados de Acesso à Rede

A figura 2.14 ilustra a máquina de estados do acesso de rede.

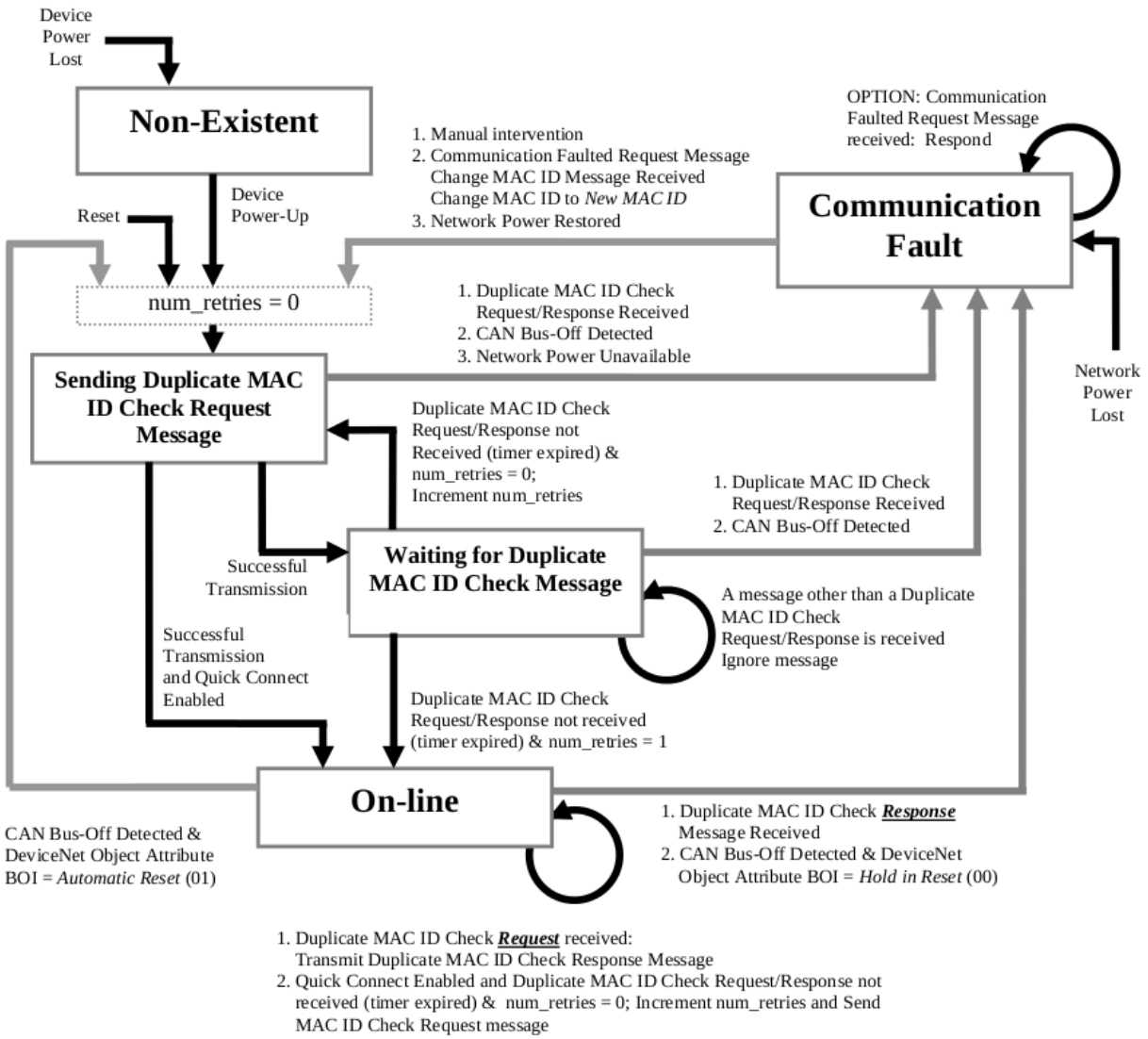

Figura 2.14: Diagrama da Máquina de estados da rede de acesso [3]

Em uma rede DeviceNet não pode haver dois dispositivos com um mesmo MAC ID. Dessa forma, sempre que um dispositivo irá acessar a rede, ele envia uma mensagem *Duplicate MAC ID Check Request*. Se um dispositivo conectado à rede responder com *Duplicate MAC ID Check Request/Response Receive*, significa que já existe um dispositivo com dado MAC ID, gerando um erro de comunicação. Caso não receba a resposta em um segundo, o dispositivo a ser conectado emite novamente a mensagem, e se continuar sem resposta, não há outro dispositivo que já esteja conectado à rede com mesmo MAC ID, tornando possível a conexão. Caso a rede esteja no modo *Quick Connect*, só é necessário a emissão de uma mensagem *Duplicate MAC ID Check Request* e não haver resposta para conectar.

2.5.2 Estabelecimento de Conexão

Todo dispositivo que deseja se comunicar pelo DeviceNet deve passar por um algoritmo para acessar a rede como descrito da figura 2.14. Diferentes tipos de conexões são possíveis.

2.5.2.1 Conexões de Mensagens Explícitas

Conexões de mensagens explicitas são gerenciadas por Mensagens explícitas não-conectadas. Mensagens de requisição não-conectadas são definidas por transmitir uma mensagem do grupo 3 com *Message ID* 6, e os únicos serviços que podem ser requisitados são para pedir para abrir uma conexão de mensagens explicitas e fechar o requerimento de conexão.

Analogamente, estas mensagens de requisição não conectadas são respondidas por mensagens com *Message ID* 5 do mesmo grupo 3, as quais podem fornecer os seguintes serviços:

- Abrir uma resposta de conexão de mensagens explicita;
- Fechar uma resposta de conexão;
- Resposta de Erro;
- Mensagem de *Heartbeat* do dispositivo;
- Mensagem de desligamento do dispositivo.

2.5.2.2 Conexões I/O

Para estabelecer uma conexão I/O é necessário seguir os seguintes passos:

1. Estabelecer uma conexão de mensagens explícitas;
2. Criar um objeto de conexão I/O por enviar uma requisição para a classe de conexão do DeviceNet;
3. Configurar todas as instancias da conexão;
4. Aplicar as configurações para o objeto de conexão I/O;
5. Repetir este processo para o outro lado da conexão.

Não é necessário estabelecer uma conexão I/O de forma dinâmica.

2.5.3 Protocolo das Mensagens

Nesta seção será descrito qual o protocolo utilizado para traduzir os dados recebidos pelo protocolo CAN para mensagens explícitas e mensagens I/O.

2.5.3.1 Mensagens Explicitas

Uma mensagem explícita utiliza a parte *Data Field* de um *frame* do CAN para transmitir a informação definida DeviceNet, como visto abaixo.

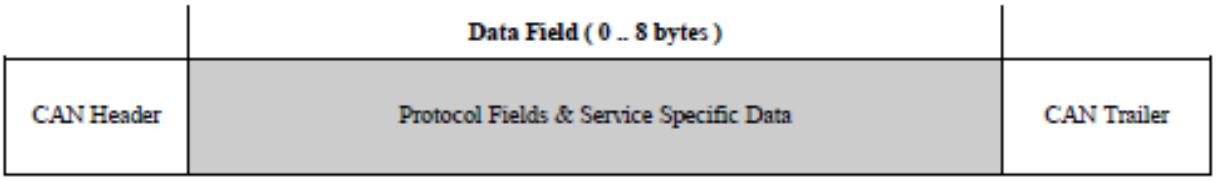

Figura 2.15: Campo do CAN *Data Field* usado pelas mensagens explícitas [3]

A informação contida nas mensagens explicitas obedecem o seguinte formato.

Byte Offset	Contents: 7	6	5	4	3	2	1	0
0	Message header							
1	Fragmentation Protocol							
2 ... 7	Message Body							

Figura 2.16: Formato do *Data Field* para mensagens explícitas [3]

O *Message Header* contém três campos:

- Frag: Contém um bit. Caso este seja "0"significa que a mensagem não é fragmentada; logo, o campo de *Fragmentation Protocol* não existirá no *Data Field*. Caso seja "1", a mensagem será fragmentada.
- XID: Este campo também possui um bit, mas seu valor é desprezado, O servidor não utiliza essa informação, sendo apenas utilizado por uma aplicação para comparar uma resposta de um pedido associado.
- MAC ID: Possui 6 bits e contém o MAC ID do emissor ou do destinatário. São feitas duas checagens neste campo: se o MAC ID do destinatário está especificado no ID da conexão, então o MAC ID do emissor deve estar definido no *Header*. Se o MAC ID do emissor for especificado no ID da conexão, então o MAC ID do destinatário deve ser especificado no *Header*. Caso essas duas checagens falhem, a mensagem é descartada.

O *Message Body* contém o campo de serviço e os argumentos específicos de serviço.

O Campo de Serviço contém o código de serviço que possui os sete bits menos significativos, os quais indicam o tipo de serviço que está sendo transmitido. Também possui o R/R no bit mais significativo, o qual determina se a mensagem é uma requisição ou resposta, dependendo se este bit for "0"ou "1", respectivamente.

2.5.3.2 Fragmentação e Remontagem

A lógica para a decisão de usar uma fragmentação é diferente entre mensagens explícitas e mensagens I/O. Para mensagens explícitas, é checado o tamanho de cada mensagem, e, caso forem maiores que 8 bits, é necessário usar o protocolo de fragmentação; porém, para mensagens I/O, é conferido o atributo de tamanho do objeto de conexão, e, caso este seja maior que 8 bits, suas mensagens I/O serão fragmentadas. Os 8 bits reservados para o protocolo de fragmentação são divididos entre o tipo de fragmento e a contagem de fragmento, como visto abaixo.

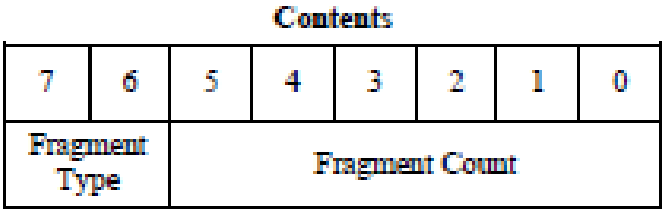

Figura 2.17: Organização dos bits no protocolo de fragmentação

O campo de tipo de fragmento pode variar de 0 a 3, conforme na tabela abaixo:

0	Primeiro fragmento
1	Fragmento do meio
2	Último fragmento
3	Confirmação do fragmento, usado pelo lado que recebe os fragmentos

Tabela 2.8: Tabela com as especificações do tipo de fragmento

O campo de contagem de fragmento serve para marcar cada fragmento separadamente, para que o lado que os receber possa identificar, caso algum seja perdido.

2.5.3.3 Mensagens I/O

Fora o protocolo de fragmentação que pode ser usado em mensagens I/O, o DeviceNet não define nenhum protocolo para o *Data Field* deste tipo de mensagem.

2.5.4 Conexões Mestre-Escravo

As mensagens explícitas têm como um de seus propósitos configurar um objeto de conexão, que pode ser usado para criar uma relação mestre-escravo.

O dispositivo mestre é aquele que reúne e distribui o *I/O data* para controlador de processos, enquanto os escravos são os quais recebem o *I/O data* distribuído pelo mestre.

O dispositivo mestre checa a lista de *scan* para determinar com qual dispositivo escravo vai se comunicar e enviar os comandos para os dispositivos corretos. Um dispositivo escravo não é capaz de iniciar comunicação sem antes receber a ordem do dispositivo mestre para a tal.

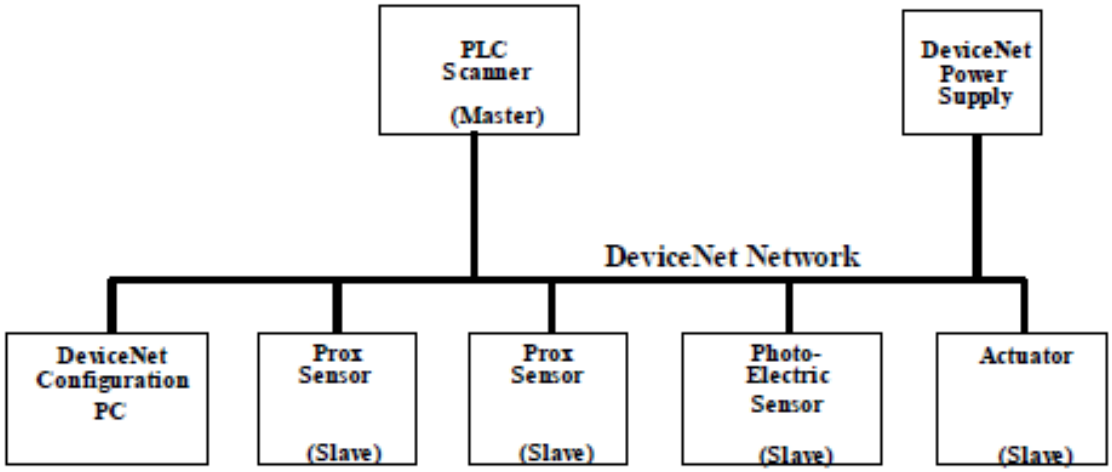

Figura 2.18: Exemplo de uma implementação mestre-escravo

O conjunto de conexão mestre-escravo possui *Connection ID's* predefinidos entre mensagens do grupo 1 e 2.

Connection ID = CAN Identifier (bits 10:0)											
10	9	8	7	6	5	4	3	2	1	0	Used For
0	Group 1 Message ID					Source MAC ID					Group 1 Messages
0	1	1	0	0		Source MAC ID					Slave's I/O Multicast Poll Response
0	1	1	0	1		Source MAC ID					Slave's I/O Change of State or Cyclic Message
0	1	1	1	0		Source MAC ID					Slave's I/O Bit-Strobe Response Message
0	1	1	1	1		Source MAC ID					Slave's I/O Poll Response or COS/Cyclic Ack Message
1	0			MAC ID				Group 2 Message ID			Group 2 Messages
1	0			Source MAC ID				0	0	0	Master's I/O Bit-Strobe Command Message
1	0			Source MAC ID				0	0	1	Master's I/O Multicast Poll Group ID
1	0			Destination MAC ID				0	1	0	Master's Change of State or Cyclic Ack Message
1	0			Source MAC ID				0	1	1	Slave's Explicit/Unconnected Response Messages
1	0			Destination MAC ID				1	0	0	Master's Explicit Request Messages
1	0			Destination MAC ID				1	0	1	Master's I/O Poll Command/COS/Cyclic Message
1	0			Destination MAC ID				1	1	0	Group 2 Only Unconnected Explicit Request Messages

Tabela 2.9: *Connection ID's* predefinidos para conexões mestre-escravo

2.5.5 Tipos de Cabo

A rede DeviceNet aceita três tipos de cabo, descritos na tabela 2.10 e ilustrados na figura 2.19

Tipo do Cabo	Função do Cabo	Taxa de Transmissão		
		125Kbis/s	250 Kbits/s	500 Kbits/s
Cabo Grosso	Tronco	500m	250m	100m
Cabo Fino	Tronco	100m	100m	100m
Cabo flat	Tronco	380m	200m	750m
Cabo Fino	Derivação	6m	6m	6m
Cabo Fino	Σ Derivação	156m	78m	39m

Tabela 2.10: Tabela com as especificações dos tipos de cabos

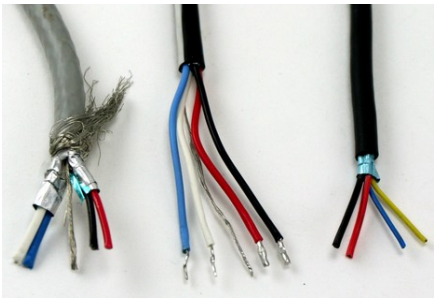

Figura 2.19: Cabo grosso, fino e flat, respectivamente

2.6 Raspberry Pi

O Raspberry Pi modelo 3 B é um computador que é baseado em um *system of chip* (SoC) Broadcom BCM2835, que inclui um processador 1.2GHz 64-bit *quad-core* ARMv8 CPU, 1 GB de RAM, Bluetooth 4.1, GPU VideoCore IV, e 512 MB de memória RAM em sua última revisão. O dispositivo não inclui uma memória não-volátil - como um disco rígido - mas possui uma entrada de cartão SD para armazenamento de dados.

As especificações do dispositivo se encontra na Tabela 2.11

Especificações do Raspberry Pi
Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
1GB RAM
BCM43438 wireless LAN e Bluetooth Low Energy (BLE) a bordo
GPIO de 40 pinos
4 portas USB
Saída HDMI
Porta de câmera CSI
Porta para display touch DSI
Entrada para Micro SD

Tabela 2.11: Especificações do Raspberry Pi [6]

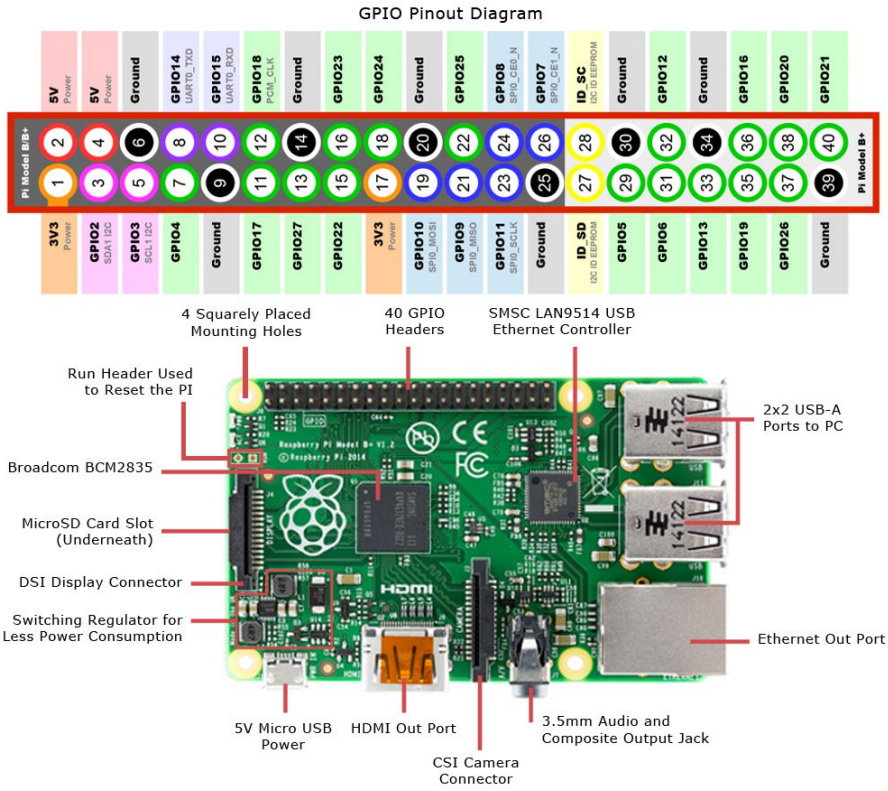

Figura 2.20: Mapeamento dos pinos do Raspberry Pi

Dos 40 pinos (ilustrados na figura 2.20), a placa possui 28 pinos de GPIO, onde existem pinos com funcionalidades específicas, que são necessários para produzir uma comunicação via SPI. Dessa forma, o GPIO16 (MOSI), GPIO9 (MISO), GPIO11 (SCLK), GPIO8 (CE0) já estarão reservados, além do GPIO25, que foi escolhido para a funcionalidade de interrupt (INT), escolhido pelos autores. Assim, sobram 23 GPIOs para as funcionalidades de entrada e saída.

Com o uso de multiplexadores, é possível usar nove GPIOs para obter 64 entradas (oito seriam usados como chave de seleção e um, para leitura da saída dos multiplexadores). Para as saídas, usam-se mais oito (seis para uso de velocidade de referência e dois para selecionar o dispositivo alvo).

A existência de pinos de entrada/saída com possibilidade do uso da comunicação SPI, a grande comunidade do Raspberry Pi e a simplicidade de uso foram os principais fatores para se adotar a placa.

2.6.0.1 Wiring Pi

Para se realizar o acesso aos pinos de GPIO, utiliza-se a biblioteca Wiring Pi [11], desenvolvida para a linguagem C/C++ e Python. Esta biblioteca disponibiliza as funções *wiringPiSetup(void)*, *pinMode (int pin, int mode)*, *digitalWrite (int pin, int value)* e *digitalRead (int pin)*.

- A função *wiringPiSetup(void)* realiza a inicialização do wiringPi. Vale ressaltar que o wiringPi utiliza uma enumeração de pinos diferente, como se pode observarr na figura 2.21

```
pi@raspberrypi:~ $ gpio readall
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 | ALT0 | 1 |  3 || 4  |   |      | 5V      |     |     |
 |   3 |   9 |   SCL.1 | ALT0 | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 0 |  7 || 8  | 1 | IN   | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | IN   | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI | ALT0 | 1 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO | ALT0 | 1 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 1 | OUT  | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | OUT  | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 0 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 0 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 1 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
```

Figura 2.21: Mapeamento dos pinos do Raspberry Pi (coluna BCM) pelo WiringPi (coluna wPi)

- A função *pinMode (int pin, int mode)* define o pino *pin* para um dos modos possíveis: INPUT, OUTPUT, PWM_OUTPUT ou GPIO_CLOCK.

- As funções *digitalWrite (int pin, int value)* e *digitalRead (int pin)* permitem ler/escrever nos pinos GPIO.

2.6.1 SocketCAN

SocketCAN é um conjunto de drivers CAN, contribuído pela *Volkswagen Research*, para o kernel do Linux. Ele abrange o conceito de *Berkley sockets*, criando uma nova família de protocolos que providencia estruturas que permitem diferentes protocolos no barramento.

A aplicação primeiramente configura e acessa a interface CAN, inicializando um socket (de forma similar ao TCP/IP), e, então, conecta o socket à interface. Uma vez conectada, o socket pode ser utilizado da mesma forma que o UDP (*User Datagram Protocol*) via funções `read`, `write`, entre outros.

2.7 Controlador Lógico Programável

O CLP é um tipo de computador muito utilizado na indústria, principalmente por ser capaz de suportar condições como poeira, vibrações e temperaturas extremas, além de ser flexível para aceitar inserção de vários módulos de entrada e saída.

Este pode ser dividido entre as seguintes partes:

- Rack com a fonte de alimentação;
- CPU;
- Entradas e Saídas;
- Software.

2.7.1 Rack e fonte de alimentação

É responsável pela comunicação e alimentação elétrica entre os módulos da CLP. A fonte do rack geralmente fornece 24V de alimentação em corrente contínua e possui entrada AC de 110V e/ou 220V.

2.7.2 CPU

O CPU da CLP deve controlar a lógica e monitorar a comunicação, podendo variar o seu modo de operação entre:

- Programação - a CLP aceita uma lógica desenvolvida por um usuário por meio de uma conexão com um PC;
- Run - modo em que o CLP está operando, usando o programa inserido;
- Stop - todas as saídas estão desligadas e não é possível rodar fazer o download de um programa;

- Reset - quando acionado, restaura o CLP para suas condições iniciais de operação.

2.7.3 Entradas e Saídas

Podem ser do tipo digital ou analógico, sendo que as entradas são aquelas que fornecem algum dado para a CLP e influenciam na execução do programa. Durante a execução do programa, as informações processadas vão para um cartão de saída analógico ou digital que, por sua vez, faz acionamentos ou desligamentos de dispositivos conectados.

2.7.4 Software

Para criar um programa é necessário um computador com o software específico da CLP usada. A maioria dos CLPs utilizam a linguagem *Ladder* para programação. Outras linguagens existentes são o *Sequential Function Chart*, o *Structured Text*, o *Instruction List* e o Diagrama de Blocos.

2.7.4.1 Ciclo de Scan

O ciclo de *scan* do Scanner de uma CLP consiste na seguinte sequência: checar a CPU; checar os módulos de entrada e saída; checar as entradas; executar o programa e atualizar as saídas. A figura 2.22 ilustra o ciclo. O tempo levado para executar um ciclo é o tempo de *scan* e é afetado principalmente pelo número de linhas do programa a ser executado.

Figura 2.22: Ciclo Scan

Capítulo 3

Metodologia

Neste capítulo será abordado o projeto e a arquitetura de objetos do protocolo DeviceNet a ser implementado no Raspberry Pi. Também são especificados quais objetos serão utilizados no projeto.

3.1 Interface entre o meio físico e o Raspberry Pi

Foi utilizado o controlador MCP2515 e *transceiver* do MCP2551 como uma solução de interface física para se realizar a comunicação do cabo CAN e o Raspberry Pi. O circuito projetado é ilustrado na figura 3.1.

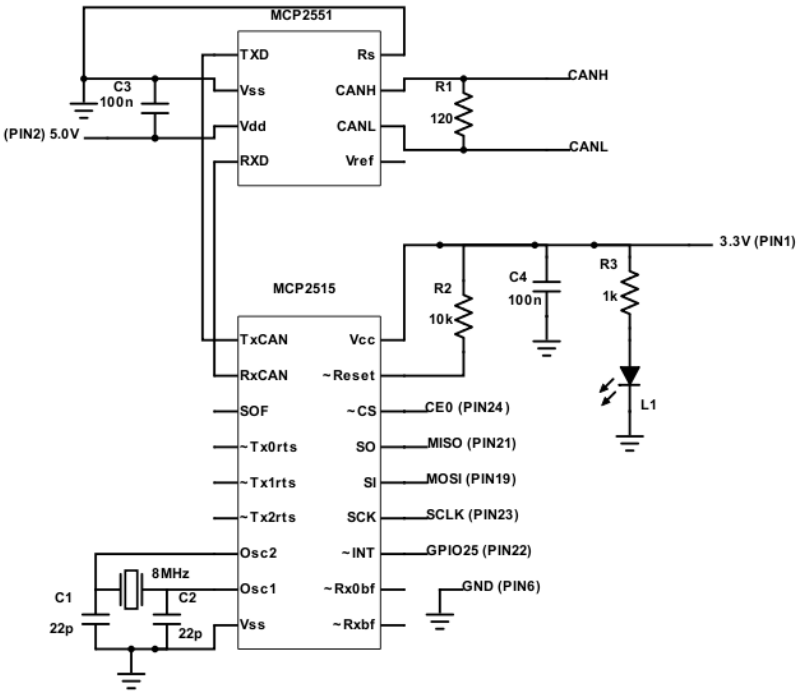

Figura 3.1: Circuito para realizar a interface entre o cabo CAN e o Raspberry Pi

Observa-se que o circuito conecta o Rs do *transceiver* diretamente ao *ground*, de modo que a operação do MCP2551 fica em *high-speed* e, entre os terminais CANH e CANL, é necessário inserir uma resistência de 120 Ω, a fim de evitar reflexão de sinal sobre o barramento.

3.2 Arquitetura do programa

O protocolo a ser implementado tem como base o CIP, que faz a modelagem do sistema a partir de objetos. Desta forma, é necessário realizar a seleção de objetos que serão implementados.

Todo dispositivo conta com pelo menos um *Connection Manager* ou *Connection Manager Object*, um *Identity Object*, um *Message Router Object* e um objeto de rede (para este projeto, é o *DeviceNet Object*).

Além desses objetos, também são necessários os de aplicação, aqueles que implementarão os propósitos do dispositivo. Através dos projetos produzidos no laboratório GRACO/UnB [1, 7], verifica-se a necessidade de implementar pelo menos 45 sensores de presença e saídas que servirão de velocidade de referência às locomotivas.

As entradas (sensores *reed-switch* utilizados para detecção de presença das locomotivas) podem ser representadas por *Presence Sensing Object* ou por *Discrete Input Point Object*. Ambos os objetos possuem os mesmos parâmetros e serviços, por isso, optou-se pela seleção do segundo objeto, por apresentar um nome mais genérico.

As saídas (velocidades de referência das locomotivas) podem ser modeladas por meio do *Discrete Output Point Object* ou pelo conjunto de objetos *AC/DC Drive*, *Control Supervisor* e *Motor Data*. Apesar do conjunto apresentar-se mais completo, a complexidade de implementação destes objetos é muito maior que a do primeiro, e a maioria dos serviços implementados não seriam interessantes para o projeto (por exemplo, o *Control Supervisor Object* possui um serviço para realizar um controle PID, porém, o controle já era realizado de forma mais eficiente em um circuito produzido e integrado na própria locomotiva [1], tornando essa *feature* do objeto desnecessária). A lógica utilizada nos projetos anteriores também levavam em conta o uso de saídas discretas [7, 1], razão da opção pela implementação do *Discrete Output Point Object*.

Como existem muitos objetos de entrada e saída, é conveniente criar *Assembly Objects* para coletarem todos os dados de entrada em uma instância, e de saída, em outra, o que possibilitará simplificar o roteamento de dados I/O para a rede.

A arquitetura de objetos está ilustrada na figura 3.2, na qual a interface entre os *Connection Objects* e a rede DeviceNet é realizada pelo protocolo CAN. É possível observar que a escolha de objetos tornará o Raspberry Pi um dispositivo de entrada e saída. Desta forma, os futuros usuários terão a liberdade de implementar os significados de cada bit de entrada e de saída, podendo portar o Raspberry para outros projetos de outras aplicações.

A linguagem para se desenvolver o protocolo foi o C++, uma vez que este é uma linguagem que tem suporte para orientação a objetos.

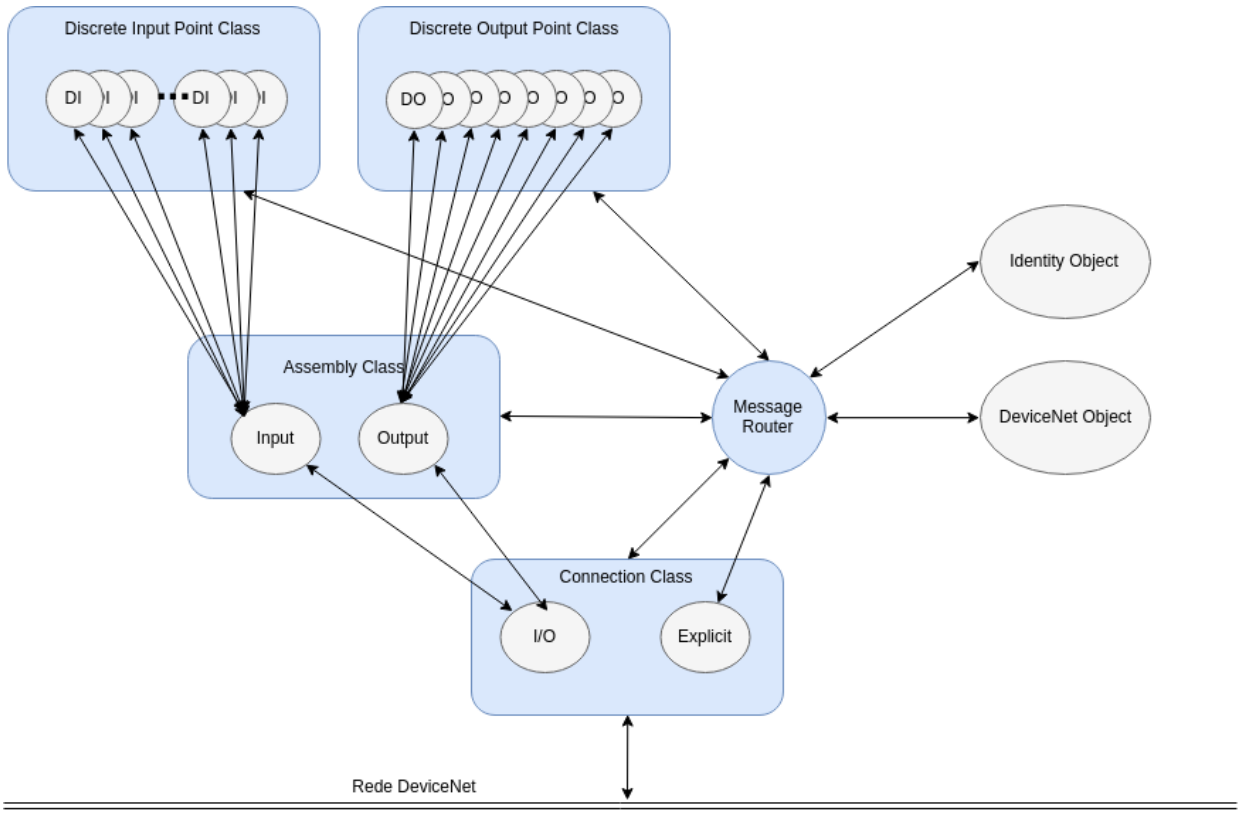

Figura 3.2: Arquitetura dos Objetos CIP

Cada classe de objeto a ser implementado possui um conjunto de serviços e atributos que são especificados pela ODVA [5, 3]. Os parâmetros a serem implementados serão definidos a seguir.

3.2.1 Objetos de Comunicação

3.2.1.1 Link Producer

O *Link Producer* é o componente responsável pela transmissão de dados em baixo nível. Este objeto suporta dois tipos de serviços: *Create* (inicializa um objeto do tipo *Link Producer*) e *Delete* (deleta um objeto do tipo *Link Producer*). Esta classe possui dois atributos: *state* (dado do tipo USINT que indica se o *Link producer* ainda não foi inicializado ou se está sendo executado) e o *connectio_id*.

As instâncias do *link producer* possuem três serviços: *Get_Attribute* (lê um atributo do objeto), *Set_Attribute* (modifica um atributo do objeto) e *Send* (avisa o objeto para transmitir os dados para a rede). O comportamento da classe segue de acordo com a máquina de estados presente na figura 3.3

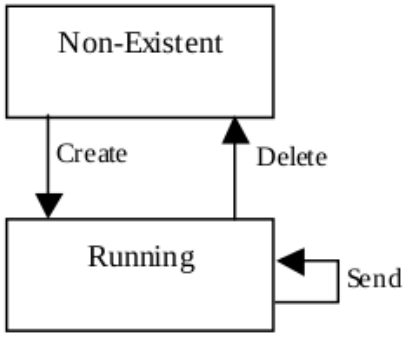

Figura 3.3: Máquina de estados do *Link Producer*[5]

3.2.1.2 Link Consumer

O *Link Consumer* é o componente responsável pela recepção das mensagens em baixo nível. Os serviços fornecidos, os atributos e a máquina de estados são iguais ao do *Link producer*, apenas suas funcionalidades variam. No *Link consumer* não existe o serviço *Send*.

3.2.1.3 Connection Object Class

O *Connection Object Class* aloca e gerencia os recursos internos das mensagens implícitas e explícitas. Um CIP *Connection Object* utiliza serviços do *Link Producer* e do *Link Consumer* de acordo com a figura 3.4. Este objeto modela a comunicação entre duas aplicações.

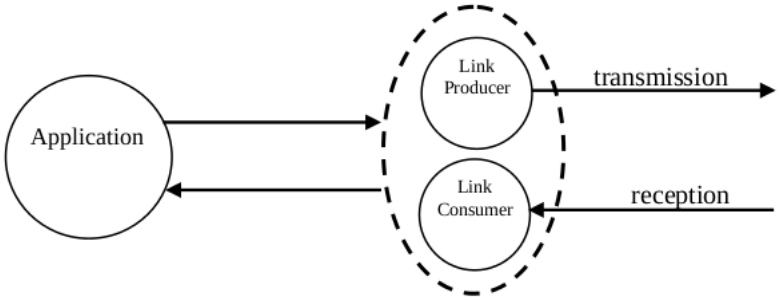

Figura 3.4: Relação entre o *Connection Object* (Tracejado) e o *Link Producer/Consumer*[5]

Os atributos da classe que foram implementados se encontram na tabela 3.1 e os serviços implementados do objeto de conexão estão listados na tabela 3.2

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.1: Atributos da classe *Connection*[5]

Service Code	Necessidade	Nome	Descrição
0x0E	Condicional	Get_Attribute_ Single	Usado para ler um valor de um atributo
0x10	Condicional	Set_Attribute_ Single	Usado para escrever um valor de um atributo

Tabela 3.2: Serviços da *Connection Class*[5]

Cada instância desta classe possui os atributos da tabela 3.3. As instâncias não oferecem serviço algum, exceto se fosse implementado o serviço de classe *Create*.

ID	Implementação	Nome	Tipo de dado	Descrição
1	Necessário	State	USINT	Status do objeto
2	Necessário	Instance_type	USINT	Indica se é conexão I/O ou explícita
3	Necessário	TransportClass_Trigger	BYTE	Define o comportamento da conexão
4	Condicional	DeviceNet_produced_ connection_id	UINT	Identificador CAN que transmite dado para a sub-rede
5	Condicional	DeviceNet_consumed_ connection_id	UINT	Identifica o grupo se mensagens associados a uma conexão no DeviceNet
6	Condicional	DeviceNet_initial_comm_ characteristics	BYTE	Identificador CAN que recebe dado para a sub-rede
7	Necessário	Produced_connection_ size	UINT	Número de bytes máximo transmitidos por esta conexão
8	Necessário	Consumed_connection_ size	UINT	Número de bytes máximo recebidos por esta conexão
9	Necessário	Expected_packet_rate	UINT	Define o temporizador desta conexão
10	Condicional	CIP_produced_ connection_id	UDINT	Identificador da mensagem enviada
11	Condicional	CIP_consumed_ connection_id	UDINT	Identificador da mensagem recebida
12	Necessário	Watchdog_timeout_action	USINT	Define o comportamento em caso de timeout
13	Necessário	Produced_connection_ path_length	UINT	Número de bytes em produced_connection_path
14	Necessário	Produced_ connection_path	UINT	Especifica o objeto para onde é produzido o dado dessa conexão
15	Necessário	Consumed_connection_ path_length	UINT	Número de bytes em consumed_connection_path
16	Necessário	Consumed_ connection_path	UINT	Especifica o objeto de onde é consumido o dado dessa conexão

Tabela 3.3: Atributos das instâncias *Connection Class*[5]

3.2.1.4 Message Router Object

O *Message Router Object* (Código de Classe 0x02) providencia um ponto de conexão de mensagem, onde um cliente pode endereçar um serviço para qualquer objeto ou instância que reside no dispositivo físico.

Os atributos da classe são padrões e estão listadas na tabela 3.4 ,e os serviços de classe e instâncias, na tabela 3.5.

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.4: Atributos da classe *Message Router*[5]

Código	Implementação	Nome	Descrição
0x0E	Necessário	Get_attribute_single	Retorna o conteúdo de um atributo específico

Tabela 3.5: Serviços de *Message Router Class*[5]

Não existem atributos nem serviços de instâncias estabelecidos pela ODVA [5].

3.2.2 Objetos de Aplicação Específica

3.2.2.1 Assembly Object

O *Assembly Object* (Código de Classe 0x04) liga atributos de múltiplos objetos, o que permite que os dados de cada objeto sejam enviados ou recebidos por uma única comunicação. Existem dois tipos de instâncias de um *Assembly Object*.

O CIP distingue o *Assembly* em entrada e saída pela perspectiva do elemento controlador (CLP). O *Assembly* de entrada é aquele que coleta informação de uma aplicação (por exemplo: sensores *reed switch*). *Assembly* de saída seria aquele que consome o dado do elemento controlador e distribui para as aplicações de saída (por exemplo: velocidade de um trem). O mapeamento é bem flexível, possibilitando mapear até bits.

Os atributos de classe estão contidos na tabela 3.6. O único atributo para instâncias de um *Assembly* estático é dado pela tabela 3.7

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.6: Atributos da classe *Assembly*[5]

ID	Implementação	Nome	Tipo de dado	Descrição
3	Necessário	Data	ARRAY de BYTE	Contém os dados assimilados

Tabela 3.7: Atributos das instâncias *Assembly Class*[5]

O único serviço que o objeto possui consta na tabela 3.8.

Código	Implementação	Nome	Descrição
0x0E	Necessário	Get_attribute_single	Retorna o conteúdo de um atributo específico

Tabela 3.8: Serviços de *Assembly Class*[5]

3.2.2.2 Identity Object

O *Identity Object* (Código de Classe 0x01) providencia a identificação e informações gerais sobre o dispositivo. Este objeto deve estar presente em todos os produtos CIP. Em sua semântica deve conter o *Vendor ID*, tipo do dispositivo, código do produto, revisão, *status*, código serial, nome do produto.

O atributo da classe encontra-se na tabela 3.9, enquanto os de instância de implementação necessária estão descritos na tabela 3.10

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.9: Atributos da classe *Identity*[5]

ID	Implementação	Nome	Tipo de dado	Descrição
1	Necessário	Vendor ID	UINT	Identificador do fornecedor
2	Necessário	Device Type	UINT	Indica o tipo de dispositivo do produto
3	Necessário	Product Code	UINT	Identifica o produto do fornecedor
4	Necessário	Revision	USINT	Representa a revisão do produto
5	Necessário	Status	WORD	Resume a condição do dispositivo
6	Necessário	Serial Number	UDINT	Serial Number do produto
7	Necessário	Product Name	SHORT INT	Nome do dispositivo

Tabela 3.10: Atributos necessários das instâncias da *Identity Class*[5]

O serviço oferecido pela classe *Identity* entrontra-se na tabela 3.11

Service Code	Necessidade	Nome	Descrição
0x0E	Condicional	Get_Attribute_ Single	Usado para ler um valor de um atributo

Tabela 3.11: Serviços da *Identity Class*[5]

3.2.2.3 Discrete Input Point Object

O *Discrete Input Point Object* (Código de Classe 0x08) modela entradas discretas de um produto. O conceito de entrada é com referência à rede, portanto, uma entrada produz dados para a rede.

O atributo da classe encontra-se na tabela 3.12 e o atributo da instância, na tabela 3.13.

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.12: Atributos da classe *Discrete Input Point*[5]

ID	Implementação	Nome	Tipo de dado	Descrição
3	Necessário	Value	BOOL	Indica o valor da entrada discreta

Tabela 3.13: Atributos necessários das instâncias da *Discrete Input Point Class*[5]

O único serviço da classe implementado encontra-se na tabela 3.14

Service Code	Necessidade	Nome	Descrição
0x0E	Condicional	Get_Attribute_ Single	Usado para ler um valor de um atributo

Tabela 3.14: Serviços da *Discrete Input Point Class*[5]

3.2.2.4 Discrete Output Point Object

O *Discrete Output Point Object* (Código de Classe 0x09) modela saídas discretas de um produto. O conceito de saída é com referência à rede, portanto, uma saída consome dados da rede.

O atributo da classe encontra-se na tabela 3.15 e o atributo da instância, na tabela 3.16.

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.15: Atributos da classe *Discrete Output Point*[5]

ID	Implementação	Nome	Tipo de dado	Descrição
3	Necessário	Value	BOOL	Indica o valor da entrada discreta

Tabela 3.16: Atributos necessários das instâncias da *Discrete Output Point Class*[5]

Os serviços da classe implementados encontram-se na tabela 3.17

Service Code	Necessidade	Nome	Descrição
0x0E	Condicional	Get_Attribute_ Single	Usado para ler um valor de um atributo
0x10	Condicional	Set_Attribute_ Single	Usado para escrever um valor de um atributo

Tabela 3.17: Serviços da *Discrete Output Point Class*[5]

3.2.3 Objeto DeviceNet

Providencia configuração e *status* da rede física do *DeviceNet*. O produto deve suportar apenas um objeto *DeviceNet* para cada rede física, como ilustrado na figura 3.5

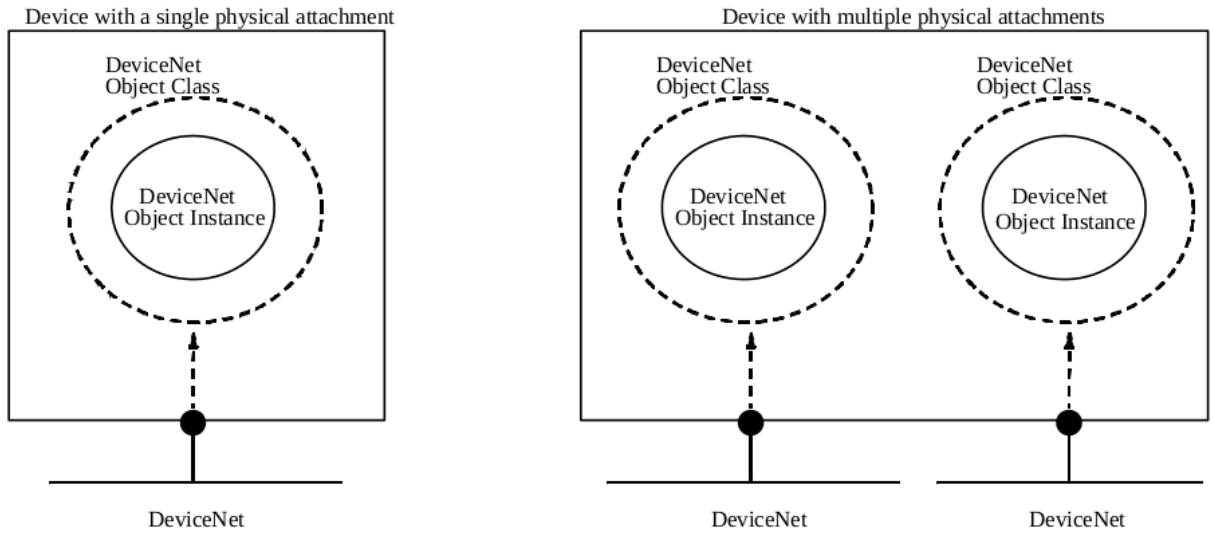

Figura 3.5: Mapeamento lógico de um objeto DeviceNet

O único atributo de classe que o *DeviceNet* possui está na tabela 3.18

ID	Necessidade	Regra de Acesso	Nome	Tipo de Dado	Descrição
1	Necessário	Get	Revision	UINT	Revisão do objeto

Tabela 3.18: Atributos da classe *DeviceNet*[3]

Os serviços são listados na tabela 3.19.

Service Code	Necessidade	Nome	Descrição
0x05	Condicional	Reset	Usado para resetar uma instância DeviceNet
0x0E	Opcional	Get_Attribute_ Single	Usado para ler um valor de um atributo
0x10	Opcional	Set_Attribute_ Single	Usado para escrever um valor de um atributo
0x4B	Opcional	Allocate_Master/Slave_ Connection_Set	Resquisita uma conexão Master/Slave
0x4C	Condicional	Release_Master/Slave_ Connection_Set	Liberação de uma conexão Master/Slave

Tabela 3.19: Serviços *DeviceNet*[3]

Os atributos implementados da instância são listadas na tabela 3.20.

ID	Implementação	Nome	Tipo de dado	Descrição
1	Opcional	MAC ID	USINT	Endereço do nó
2	Opcional	Baud Rate	USINT	Taxa de transmissão de dados
3	Opcional	BOI	BOOL	Bus-off interrupt
5	Opcional	Allocation Information	Struct de:	
		Allocation Choice Byte	BYTE	Usado para identificar o tipo de conexão Master/Slave
		Master's MAC ID	USINT	Usado para identificar o MAC ID do Master de uma conexão Master/Slave

Tabela 3.20: Atributos das instâncias *DeviceNet Class*[3]

Capítulo 4

Resultados

Neste capítulo serão apresentados os resultados obtidos através da implementação da metodologia descrita no capítulo anterior.

4.1 Implementação de Software

Através da arquitetura em objetos, implementou-se o protocolo DeviceNet em C++ (código presente em anexo). O *SocketCAN* permite criar uma rede CAN virtual, possibilitando simular se o protocolo *DeviceNet* se comunicará de forma correta com outro dispositivo, mesmo que a implementação física (protocolo CAN) não funcione devidamente.

O código foi propriamente adaptado para realizar a interface com a rede CAN, de acordo com CIA-CAN [12]. Desta forma, foram rigorosamente testadas todas as funcionalidades que seriam necessárias no protocolo.

4.1.1 Implementação do protocolo em rede virtual

Algumas simulações foram realizadas e, entre as mais importantes, verificou-se as mensagens de criação de conexões explícitas ou implícitas, uso de serviços por meio de mensagens explícitas e mensagens I/O para atualizar as entradas e saídas. A mensagem de erro mais relevante para o protocolo é o de MAC ID duplicado. Os passos da simulação aqui estabelecidos seguem a ordem a que o *Scanner DeviceNet* realiza.

Primeiramente, o *Scanner* estabelece as conexões explícitas. Depois estabelece a I/O. Após estabelecer a I/O, ele modifica o atributo da instância de conexão *expected packet rate*, por meio de mensagem explícita. Por fim, ele entra no ciclo *scan*, atualizando periodicamente as entradas e saídas através de mensagens I/O ou mensagens explícitas (caso o dispositivo não tenha suporte a mensagens I/O).

4.1.1.1 Estabelecimento de conexões explícitas

Estabeleceu-se o MAC ID do dispositivo como 60 (para o *DeviceNet* pode ser atribuido de 0 a 62). Desta forma, segundo o padrão mestre/escravo, a mensagem de solicitar uma conexão de um mestre de

MAC ID 10 (0x0A) seria em hexadecimal:

```
5E6#0A4B0301010A
```

Note-se que o campo de identificação CAN se encontra no campo antes da cerquilha (`5E6`), e o corpo da mensagem se encontra após (`0A4B0301010A`).

- 5E6 - Grupo = 2, dest MAC ID = 60, Message ID = 6 (*Unconnected Explicit Request Messages*)
- 0A - Frag = 0, XID = 0, Source MAC ID = 10 (Mensagem não fragmentada)
- 4B - R/R = 0, Service Code = 0x4B (R/R = Resquest, Service = Allocate Connection)
- 03 - Class ID = 3 (DeviceNet Object)
- 01 - Instance ID = 1
- 01 - Allocation Choice = 1 (Explicit)
- 0A - Allocator MAC ID = 10

A resposta obtida foi dada pela figura 4.1

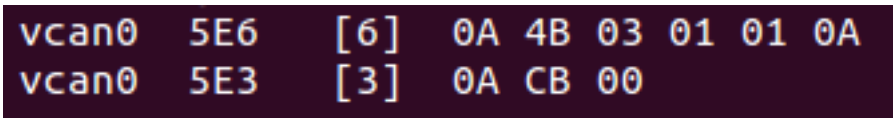

Figura 4.1: Resposta para solicitação alocação de uma conexão explícita

Pode-se analisar o campo de resposta (na segunda linha):

- 5E6 - Grupo = 2, src MAC ID = 60, Message ID = 3 (*Unconnected Explicit Response Messages*)
- 0A - Frag = 0, XID = 0, dest MAC ID = 10 (Mensagem não fragmentada)
- CB - R/R = 1, Service Code = 0x4B (R/R= Response, Service = Allocate Connection)
- 00 - Body format = DeviceNet (8/8)

Portanto, verificou-se uma resposta de uma alocação de conexão explícita estabelecida.

4.1.2 Estabelecimento de uma conexão I/O

Utilizando os mesmos parâmetros de endereços utilizados na alocação explícita, montou-se a mensagem de requisição de alocação I/O.

```
5E6#0A4B0301020A
```

- 5E6 - Grupo = 2, dest MAC ID = 60, Message ID = 6 (*Unconnected Explicit Request Messages*)
- 0A - Frag = 0, XID = 0, Source MAC ID = 10 (Mensagem não fragmentada)
- 4B - R/R = 0, Service Code = 0x4B (R/R = Resquest, Service = Allocate Connection)
- 03 - Class ID = 3 (DeviceNet Object)
- 01 - Instance ID = 1
- 02 - Allocation Choice = 2 (I/O Poll)
- 0A - Allocator MAC ID = 10

Caso a mensagem de alocação I/O seja enviada antes de se ter estabelecido uma conexão explícita, a resposta será dada pela figura 4.2

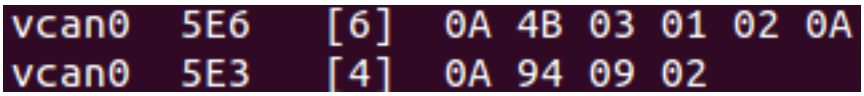

Figura 4.2: Resposta para solicitação alocação de uma conexão I/O antes da explícita

A resposta representa um erro:

- 5E3 - Grupo = 2, src MAC ID = 60, Message ID = 3 (*Unconnected Explicit Response Messages*)
- 0A - Frag = 0, XID = 0, dest MAC ID = 10 (Mensagem não fragmentada)
- 94 - R/R = 1, Service Code = 0x14 (Response, Service Code = Error)
- 09 - General error = Invalid Attribute
- 02 - Additional code = 2 (definido pelo DeviceNet)

Caso a mensagem de alocação I/O seja enviada após ter-se estabelecido uma conexão explícita, a resposta será dada pela figura 4.3:

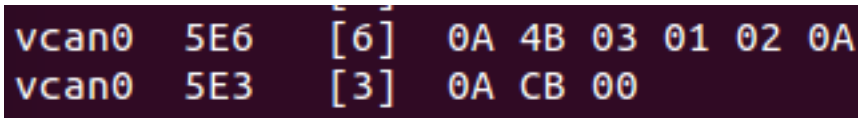

Figura 4.3: Resposta para solicitação alocação de uma conexão I/O após da explícita

Observa-se que é a mesma resposta de quando foi alocada a mensagem explícita na figura 4.1, o que significa uma alocação bem sucedida.

4.1.3 Configuração do Expected Packet Rate

A mensagem que o mestre deve enviar ao escravo para se configurar o valor do *expected packet rate* será:

```
5E4#0A100502090010
```

- 5E4 - Grupo = 2, dest MAC ID = 60, Message ID = 4 (*Master's Explicit Request Messages*)
- 0A - Frag = 0, XID = 0, source MAC ID = 10 (Mensagem não fragmentada)
- 10 - R/R = 0, Service Code = 0x10 (R/R = Resquest, Service = Set single attribute)
- 05 - Class ID = 5 (Connection Object)
- 02 - Instance ID = 2 (IO Poll)
- 09 - Attribute = 9 (Expected Packet Rate)
- 0010 - Expected Packet Rate Value

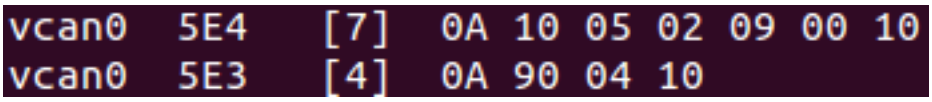

Figura 4.4: Resposta para mudança do valor EPR

A análise da resposta da figura 4.4 é:

- 5E3 - Grupo = 2, dest MAC ID = 60, Message ID = 4 (*Slave's Explicit Response*)
- 0A - Frag = 0, XID = 0, source MAC ID = 10 (Mensagem não fragmentada)
- 90 - R/R = 1, Service Code = 0x10 (R/R = Response, Service = Set single attribute)
- 0410 - Expected Packet Rate Value (existe um algorítimo de cálculo de EPR)

4.1.4 Leitura das entradas

A leitura das entradas pode ser realizada por meio de mensagens explícitas ou I/O, caso exista a leitura por meio de mensagens I/O.

O programa realizado faz o processamento da mensagem I/O através do tamanho de seus dados recebidos pela mensagem; se não houver dados, é uma requisição de leitura das entradas, ao qual a resposta gerada são oito bytes (64 bits) contendo os valores de todos os sensores.

Desta forma, a leitura pode ser realizada através do comando:

```
5E5#
```

Onde o header significa:

- 5E5 - Grupo = 2, dest MAC ID = 60, Message ID = 5 (*Master's I/O Poll Command*)

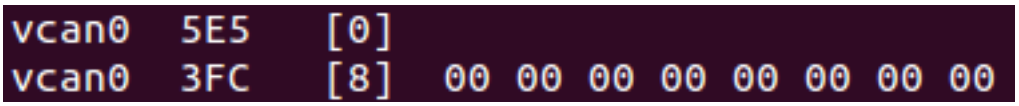

Figura 4.5: Resposta para mensagem de leitura I/O

Verifica-se que todas entradas estão nulas.

4.1.5 Escrita de saídas

A escrita das saídas podem ser realizadas por mensagens explícitas e I/O. A escrita por I/O tem prioridade por consumir menos mensagens no barramento.

A mensagem I/O para escrita de saída deve ter um dado de 1 byte (8 bits). Cada bit corresponde a uma saída. Um exemplo de mensagem do tipo de escrita de saída seria:

```
5E5#16
```

- 5E5 - Grupo = 2, dest MAC ID = 60, Message ID = 5 (*Master's I/O Poll Command*)
- 16 - 00010110 = instâncias 2, 3 e 5 em nível alto

Para se ler o valor de uma instância, basta utilizar a mensagem explícita:

```
5E4#0A0E090103
```

- 5E4 - Grupo = 2, dest MAC ID = 60, Message ID = 4 (*Master's Explicit Request Messages*)
- 0A - Frag = 0, XID = 0, source MAC ID = 10 (Mensagem não fragmentada)
- 0E - R/R = 0, Service Code = 0x0E (R/R = Resquest, Service = Get single attribute)
- 09 - Class ID = 9 (Discrete Output Point Object)
- 01 - Instance ID = 1
- 03 - Attribute ID = 3 (Value)

Utilizando as mensagens acima, obteve-se os resultados da figura 4.6:

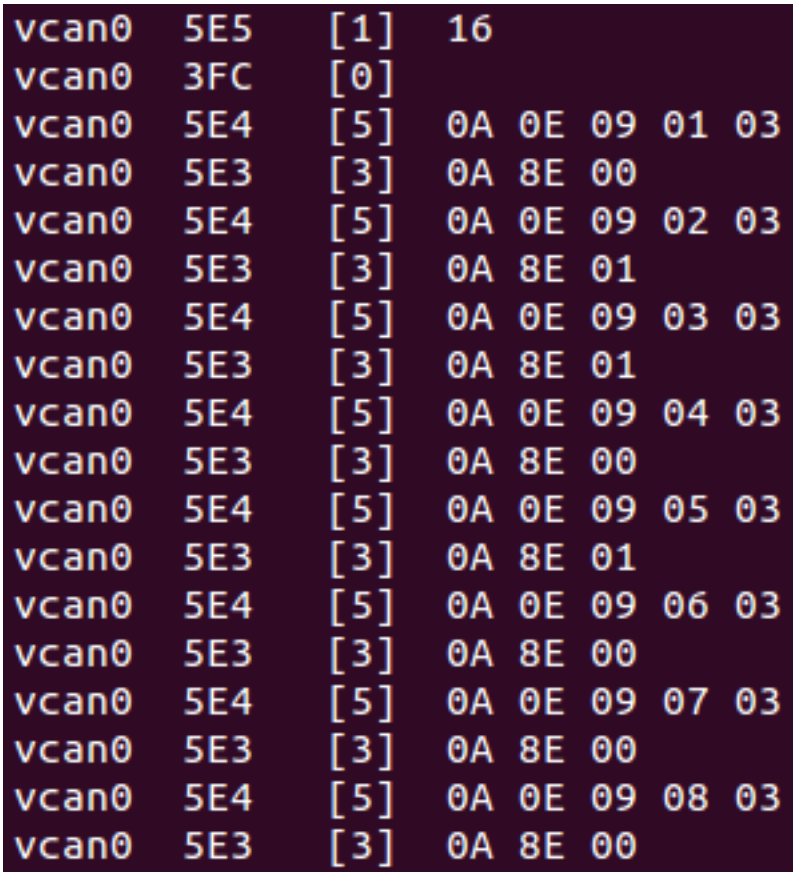

Figura 4.6: Resposta para mensagem de escrita I/O

Verifica-se que para a resposta da mensagem I/O não houve dado retornado, o que era o esperado. Para cada mensagem explicita, o 3 byte de dado corresponde ao valor atual da instância. Desta forma, o roteamento da mensagem de escrita I/O ocorreu como esperado.

4.1.6 MAC ID Duplicado

Caso seja enviada uma mensagem com o cabeçalho:

```
5E7#
```

- 5E7 - Grupo = 2, dest MAC ID = 60, Message ID = 7 (*Duplicate Duplicate MAC ID Check*)

O dispositivo responde com uma mensagem de erro e entra em estado de erro, no qual não realiza qualquer outra operação a não ser que seja resetado, como ilustra a figura 4.7.

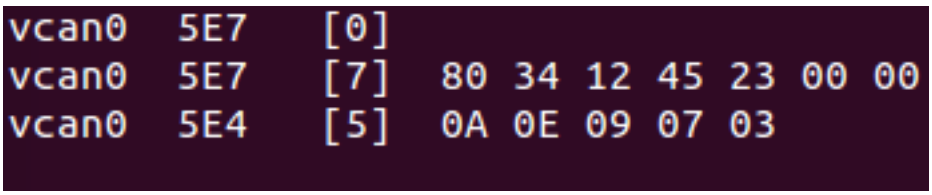

Figura 4.7: Mensagem de MAC ID Duplicado recebida

4.1.7 Avaliação do protocolo DeviceNet implementado

Através de testes severos em uma gama de variedades como os ilustrados nos tópicos anteriores, foi possível verificar que o protocolo foi corretamente implementado, observando-se se o programa produzia todas as mensagens de erro nos devidos contextos, e se todos os serviços tomavam a ação esperada.

Portanto, o protocolo DeviceNet pode ser corretamente implementado no Raspberry Pi.

4.2 Implementação na rede física

Utilizando o circuito projetado, foi realizada uma interface entre o Raspberry Pi e o cabo CAN fino. Porém, ao se conectar a rede, a interface permitiu apenas a leitura da rede. Quando se tentava realizar a escrita, o controlador CAN se desconectava da rede.

Pela análise da rede CAN, foi descoberto que o controlador CAN (MCP2515) passava do estado *Error-Active* (modo de funcionamento normal) para o *Bus-off* (modo offline). Segundo o protocolo CAN e o manual do MCP2515 [12, 4], esse modo é alcançado apenas se o contador de erros do controlador CAN superar 255. Isso significa que a mensagem transmitida apresentou erro 256 vezes.

Como o monitoramento do cabo CAN não indicou qualquer leitura dos sinais transmitidos, conclui-se que o erro gerado não tem associação com os erros CAN (*Ack error, Bit error, Form error, CRC error, Stuff error*); portanto, existe algum erro de comunicação entre MCP2515 e o driver CAN do Linux.

Não houve tempo necessário para investigar a falha, pois, inicialmente, o protocolo CAN realizava a interface corretamente com o meio físico e o Raspberry Pi, possibilitando a leitura e escrita de dados. No momento em que foi realizada a interface, o protocolo *DeviceNet* não estava completo. O erro ocorreu de forma inesperada e não foi encontrada uma solução para o problema. Assim, não foi possível realizar o empilhamento da camada de aplicação, fornecida pelo *DeviceNet*, com as camadas de enlace de dados e física, fornecidas pelo protocolo CAN.

Entretanto, foram encontradas algumas alternativas no fórum do Raspberry Pi, onde várias pessoas relataram o mesmo problema. A primeira alternativa seria a troca do conjunto MCP2515 e MCP2551 por um controlador CAN produzido especificamente para realizar interface entre o CAN e o Raspberry, conhecido como PiCAN. A segunda possibilidade seria utilizar uma placa que já tenha suporte para realizar a interface com o CAN (por exemplo o BeagleBone Black). Por último, que seria uma solução mais drástica, seria implementar o protocolo CAN em um SoC (*System on Chip*) e fornecer o próprio driver para o Raspberry Pi.

Capítulo 5

Conclusões

O principal objetivo, criar uma comunicação por meio do protocolo DeviceNet entre uma placa Raspberry Pi e uma CLP, foi alcançado; porém, ainda não foi possível testar a comunicação utilizando uma conexão CAN real.

O protocolo DeviceNet foi implementado corretamente no Raspberry Pi, o que pode ser comprovado pelas simulações em redes CAN virtuais. Através do código produzido é possível realizar a comunicação de dispositivos de entrada e saída com a CLP, caso seja realizada a devida interface entre o DeviceNet com o CAN. As funcionalidades implementadas permitem a leitura dos sensores e escrita nos atuadores via portas lógicas.

O protocolo CAN, implementado através do controlador MCP2515, *transceiver* MCP2551 e o *driver* SocketCAN passou a apresentar erros ao final do projeto, impossibilitando avaliar o DeviceNet implementado no Raspberry em uma rede real. A compreensão da falha desses dispositivos não pôde ser realizada por completa. Algumas possíveis formas de se contornar este problema foram encontradas, porém, não houve tempo/recurso financeiro para implementá-las.

5.1 Perspectivas Futuras

Uma das soluções seria a mudança para o controlador PiCAN (cerca de 50 dólares), o que não iria acarretar em mudança no código produzido neste projeto, pois este módulo foi produzido e testado pela comunidade de usuários do Raspberry Pi, especificamente para realizar a comunicação Raspberry/CAN utilizando os *drivers* do SocketCAN.

Outra possibilidade seria o uso do Beaglebone Black (cerca de 70 dólares), que já possui entradas e saídas específicas para uso do CAN, necessitando apenas de um *transceiver* para realizar a interface com a rede CAN. Dessa forma, seria necessário modificar o código para utilizar os pinos GPIO desta plataforma, uma vez que também utiliza os *drivers* oferecidos pelo SocketCAN.

Uma última solução seria utilizar um SoC para se implementar o protocolo CAN e produzir o próprio *driver* para o Linux. O custo financeiro é baixo para se realizar essa implementação, porém, há uma maior

complexidade para se produzir esse dispositivo, o que demandaria mais tempo para ser desenvolvido.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] ANDRADE, L. H. do E. S.; VEIGA, I. V. A. *PROJETO E IMPLEMENTAÇÃO DE UMA MAQUETE PARA ESTUDO DE PROBLEMAS RELACIONADOS À AUTOMAÇÃO DE SISTEMAS DE TRANSPORTE FERROVIÁRIO*. [S.l.]: Universidade de Brasília, 2013.

[2] ZURAWSKI, R. *The Industrial Communication Technology Handbook (Industrial Information Technology)*. 1. ed. [S.l.]: CRC Press, 2005. ISBN 0849330777,9780849330773.

[3] ODVA. *THE CIP NETWORKS LIBRARY, Volume 3, DeviceNet Adaption of CIP*. [S.l.]: ODVA and ControlNet International Ltd., 2007.

[4] DATASHEET MCP2515. [S.l.]: Microchip. `http://ww1.microchip.com/downloads/en/DeviceDoc/21801d.pdf`.

[5] ODVA. *THE CIP NETWORKS LIBRARY, Volume 1, Common Industrial Protocol*. [S.l.]: ODVA and ControlNet International Ltd., 2007.

[6] RASPBERRY Pi 3 Model B Specs. `https://www.raspberrypi.org/products/raspberry-pi-3-model-b/`. Acessado em: 04/04/2018.

[7] JESUS, T. R. de; AQUINO, J. A. V. de. *ESTRATÉGIA DE CONTROLE DE TRÁFEGO EM VIAS FÉRREAS SINGELAS, UTILIZANDO PROGRAMAÇÃO SFC SOBRE ARQUITETURA DE CONTROLADORES INDUSTRIAIS PROGRAMÁVEIS*. [S.l.]: Universidade de Brasília, 2009.

[8] IEC 61158 PROFIBUS: Physical Layer. [S.l.], mar. 2003.

[9] PLEINEVAUX P.; DECOTIGNIE, J.-D. Time critical communication networks: field buses. *IEEE Network*, IEEE, v. 2, 05 1988.

[10] CONTROLLER Area Network (CAN) Basics. [S.l.], 1999.

[11] WIRING Pi. `http://wiringpi.com/`. Acessado em: 04/04/2018.

[12] CIA-CAN. `https://www.can-cia.org/`. Acessado em: 04/06/2018.

[13] SOCKETCAN Documentation. `https://www.kernel.org/doc/Documentation/networking/can.txt`. Acessado em: 04/04/2018.

ANEXOS

I. DESCRIÇÃO DO CONTEÚDO DO CD

O CD possui quatro arquivos: o TrabalhoGraduacao.pdf, que consiste, além deste documento, dos arquivos de código feito em C++; DeviceNet.cpp, que se refere ao arquivo com a *main* e suas bibliotecas criadas; DeviceNet.h e SCan.h, que possuem funções e definições usadas no código principal. Ressalte-se que este código possui algumas linhas comentadas e outras adicionadas para que seja possível a sua realização utilizando o CAN virtual, uma vez que não foi possível testar o código para o CAN real devido a problemas anteriormente comentados.

II. PROGRAMAS UTILIZADOS

II.1 Anexo A - Biblioteca

```
typedef int BOOL;
typedef unsigned char UCHAR;
typedef unsigned int UINT;
typedef unsigned long ULONG;

#define LOBYTE(w) ((UCHAR)(w))
#define HIBYTE(w) ((UCHAR)((int)(w) >> 8))

#define MAC_ID 0x3c // Between 0-62 (63 reserved by devicenet)
#define BAUD_RATE 0x00 // 0=125k, 1=250k, 2=500k
#define VENDOR_ID 0x1234 // Dummy value
#define SERIAL_ID 0x2345 // Dummy value
#define NUM_IN 45 // Number of inputs (Max 64)
#define NUM_OUT 8 // Number of outputs (Max 8)

int pinout[8]={8, 9, 7, 0, 2, 3, 15, 16};
int pinin[9]={30, 21, 22, 23, 24, 25, 27, 28, 29};

#define FALSE 0
#define TRUE 1
#define OK 1
#define NO_RESPONSE 0
#define BUFSIZE 80
#define LENGTH BUFSIZE-1
#define MESSAGE_TAG BUFSIZE-2
#define PATH_SIZE 10

// Connection state atribbutes (CIP v1-3.14)
#define NON_EXISTENT 0x00
#define CONFIGURING 0x01
#define WAITING_FOR 0x02
#define ESTABLISHED 0x03
#define TIMED_OUT 0x04
#define DEFERRED 0x05
#define CLOSING 0x06
```

```
// Define Instance IDs and TIMER numbers
#define EXPLICIT 0x01
#define IO_POLL 0x02
#define BIT_STROBE 0x03
#define COS_CYCLIC 0x04
#define ACK_WAIT 0x05
#define UPDATE 0x06
#define INPUT_ASSEMBLY 0x01
#define OUTPUT_ASSEMBLY 0x02

// define message tags which can be used to identify message
#define RECEIVED_ACK 0x01
#define SEND_ACK 0x02
#define ACK_TIMEOUT 0x03
#define ACK_ERROR 0x04

// define bit positions for global event word
#define IO_POLL_REQUEST 0x0001
#define EXPLICIT_REQUEST 0x0002
#define DUP_MAC_REQUEST 0x0004
#define UNC_PORT_REQUEST 0x0008
#define EXPLICIT_TIMEOUT 0x0040
#define IO_POLL_TIMEOUT 0x0080
#define DEVICE_UPDATE 0x0200
#define ACK_WAIT_TIMEOUT 0x2000
#define FULL_RESET 0x8000

// Define bit positions in Allocation Choice attribute
#define EXPLICIT_CONXN  0x01
#define IO_POLL_CONXN 0x02
#define BIT_STROBE_CONXN 0x04
#define COS_CONXN 0x10
#define CYCLIC_CONXN 0x20

// Service codes (CIP v1-appendixA)
#define GET_REQUEST 0x0E
#define SET_REQUEST 0x10

// Error codes
#define SERVICE_NOT_SUPPORTED 0x08
#define ATTRIB_NOT_SUPPORTED 0x14
```

```
//Define message fragment values
#define FIRST_FRAG 0x00
#define MIDDLE_FRAG 0x40
#define LAST_FRAG 0x80
#define ACK_FRAG 0xC0

// define bit positions for global status word
#define OWNED 0x0001
#define ON_LINE 0x0002
#define CONFIGURED 0x0004
#define SELF_TESTING 0x0008
#define OPERATIONAL 0x0010
#define DEVICE_FAULT 0x0800
#define DUP_MAC_FAULT 0x1000
#define BUS_OFF 0x2000
#define NETWORK_FAULT 0x3000
#define LONELY_NODE 0x4000

// DeviceNet error codes
#define RESOURCE_UNAVAILABLE 0x02
#define SERVICE_NOT_SUPPORTED 0x08
#define INVALID_ATTRIB_VALUE 0x09
#define ALREADY_IN_STATE 0x0B
#define OBJECT_STATE_CONFLICT 0x0C
#define ATTRIB_NOT_SETTABLE 0x0E
#define PRIVILEGE_VIOLATION 0x0F
#define DEVICE_STATE_CONFLICT 0x10
#define REPLY_DATA_TOO_LARGE 0x11
#define NOT_ENOUGH_DATA 0x13
#define ATTRIB_NOT_SUPPORTED 0x14
#define TOO_MUCH_DATA 0x15
#define OBJECT_DOES_NOT_EXIST 0x16
#define NO_STORED_ATTRIB_DATA 0x18
#define STORE_OP_FAILURE 0x19
#define INVALID_PARAMETER 0x20
#define INVALID_MEMBER_ID 0x28
#define MEMBER_NOT_SETTABLE 0x29

// DeviceNet additional error codes (object specific)
#define ALLOCATION_CONFLICT 0x01
#define INVALID_ALLOC_CHOICE 0x02
#define INVALID_UNC_REQUEST 0x03
```

```
// DeviceNet service codes
#define RESET_REQUEST 0x05
#define START_REQUEST 0x06
#define STOP_REQUEST 0x07
#define CREATE_REQUEST 0x08
#define DELETE_REQUEST 0x09
#define GET_REQUEST 0x0E
#define SET_REQUEST 0x10
#define RESTORE_REQUEST0x15
#define SAVE_REQUEST 0x16
#define ALLOCATE_CONNECTIONS 0x4B
#define RELEASE_CONNECTIONS 0x4C

// Class IDs
#define IDENTITY_CLASS 0x01
#define ROUTER_CLASS 0x02
#define DEVICENET_CLASS 0x03
#define ASSEMBLY_CLASS 0x04
#define CONNECTION_CLASS 0x05
#define DISCRETE_INPUT_POINT_CLASS 0x08
#define DISCRETE_OUTPUT_POINT_CLASS 0x09

// Connection instance type (CIP v1-3.15)
//#define EXPLICIT 0x00
//#define IO_POLL 0x01

//
#define SUCCESS_RESPONSE 0x80
#define NON_FRAGMENTED 0x7F
#define ERROR_RESPONSE 0x94
#define NO_ADDITIONAL_CODE 0xFF
#define DEFAULT_MASTER_MAC_ID 0xFF
#define EXPLICIT_TYPE 0x00
#define IO_TYPE 0x01
```

II.2 Anexo B - Funções de leitura/escrita e o temporizador

```
#include<linux/can.h>
#include<linux/can/raw.h>
#include<endian.h>
#include<net/if.h>
#include<sys/ioctl.h>
#include<sys/socket.h>
#include<sys/types.h>
#include<unistd.h>
#include<iostream>
#include<thread>
#include<cerrno>
#include<csignal>
#include<cstdint>
#include<cstdio>
#include<cstring>
#include "DeviceNet.h"

int write_flag = 0;
int read_flag = 0;
int sock_can_on = 1;
UINT can_id_write;
UCHAR global_CAN_write[BUFSIZE];
UINT last_id_write;
UCHAR last_global_write[BUFSIZE];

UCHAR global_CAN_buf[BUFSIZE];
UINT global_timer[10];
UINT global_status;
UINT global_event;

void strcpy_UCHAR (UCHAR *dst, char *src)
{
  int i=0;
  while(src[i]!='\0')
    {
      dst[i] = src[i];
      i++;
    }
  dst[i]='\0';
}

void sock_can_read()
{
  struct ifreq ifr;
  struct sockaddr_can addr;
  struct can_frame frame;
  int s;
  int valid_message;
  int count = 0;
  int ignore_msg = 0;
  memset(&ifr, 0x0, sizeof(ifr));
  memset(&addr, 0x0, sizeof(addr));
  memset(&frame, 0x0, sizeof(frame));

  s = socket(PF_CAN, SOCK_RAW, CAN_RAW);

  if(s==0)
    {
      printf("\n␣----------\n␣SOCKET␣CREATION␣ERROR!\n␣----------\n");
    }

  strcpy(ifr.ifr_name, "vcan0");
  ioctl(s, SIOCGIFINDEX, &ifr);

  addr.can_ifindex = ifr.ifr_ifindex;
  addr.can_family = PF_CAN;

  if (bind(s, (struct sockaddr *)&addr, sizeof(addr)) < 0)
    {
      printf("\n␣----------\n␣SOCKET␣BINDING␣ERROR!\n␣----------\n");
    }


  while(sock_can_on)
    {

      read(s, &frame, sizeof(frame));
      valid_message = 0;
```

```
ignore_msg = 0;
if((frame.can_dlc ==global_CAN_write[LENGTH])&&(frame.can_id!=can_id_write))
  {
    ignore_msg = 1;
    count = 0;
    do{
      if(global_CAN_write[count]!=frame.data[count]) ignore_msg = 0;
      count++;
    }while((ignore_msg)&&(count<frame.can_dlc));
  }
if(!ignore_msg)
  {
    switch(frame.can_id)
      {
      case 0b10111100101: // Masters I/O Poll Command/Change of State/Cyclic Message (CIPv3c3.7): Group 2 MACID = 60 Message ID =
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
          }
        //printf("\n");
        global_event |= IO_POLL_REQUEST;
        valid_message = 1;
        break;
      case 0b10111100100: // Masters Explicit Request Messages (CIPv3c3.7): Group 2 MACID = 60 Message ID = 4
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
          }
        //printf("\n");
        global_event |= EXPLICIT_REQUEST;
        valid_message = 1;
        break;
      case 0b10111100111: // Duplicate MAC ID Check Messages: Group 2 MACID = 60 Message ID = 7
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
          }
        //printf("\n");
        global_event |= DUP_MAC_REQUEST;
        valid_message = 1;
        break;
      case 0b10111100110: // Unconnected Explicit Request Messages (CIPv3c3.7): Group 2 MACID = 60 Message ID = 6
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
          }
        //printf("\n");
        global_event |= UNC_PORT_REQUEST;
        valid_message = 1;
        break;
      case 0b01111111100: // Slaves I/O Poll Response or Change of State/Cyclic Acknowledge Message (CIPv3c3.7): Group 1 MACID = 60 Message
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
          }
        //printf("\n");
        global_event |= 0x0010;
        valid_message = 1;
        break;
      case 0b10111100011: // Slaves Explicit/ Unconnected Response Messages (CIPv3c3.7): Group 2 MACID = 60 Message ID = 3
        global_CAN_buf[LENGTH] = frame.can_dlc;
        //printf("READ\t%X\t",frame.can_id);
        for(count=0;count<global_CAN_buf[LENGTH];count++)
          {
            global_CAN_buf[count]=frame.data[count];
            //printf("%02X ",global_CAN_buf[count]);
```

```
                }
                //printf("\n");
                global_event |= 0x0020;
                valid_message = 1;
                break;
                //case 0b10111100111: // Duplicate MAC ID Check Messages: Group 2 MACID = 60 Message ID = 7
                //global_event |= 0x0040;
                //valid_message = 1;
                //break;
              default:
                valid_message = 0;
                break;
              }
          }
        if(valid_message)
          {
            global_CAN_buf[LENGTH] = frame.can_dlc;
            //printf("READ\t%X\t",frame.can_id);
            for(count=0;count<global_CAN_buf[LENGTH];count++)
              {
                global_CAN_buf[count]=frame.data[count];
                //printf("%02X ",global_CAN_buf[count]);
              }
            //printf("\n");
          }
      }
    close(s);
}


void sock_can_write()
{

  struct ifreq ifr;
  struct sockaddr_can addr;
  struct can_frame frame;
  int s;
  int i;
  int count = 0;
  memset(&ifr, 0x0, sizeof(ifr));
  memset(&addr, 0x0, sizeof(addr));
  memset(&frame, 0x0, sizeof(frame));

  s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
  strcpy(ifr.ifr_name, "vcan0");
  ioctl(s, SIOCGIFINDEX, &ifr);

  addr.can_ifindex = ifr.ifr_ifindex;
  addr.can_family = PF_CAN;

  bind(s, (struct sockaddr *)&addr, sizeof(addr));
  while(1)
    {
      if(write_flag)
        {
          frame.can_id = can_id_write;
          frame.can_dlc = global_CAN_write[LENGTH];
          //printf("WRITE\t%X\t",frame.can_id);
          for(count=0;count<global_CAN_write[LENGTH];count++)
            {
              frame.data[count]=global_CAN_write[count];;
              //printf("%02X ",frame.data[count]);
            }
          printf("\n");
          write(s,&frame,sizeof(frame));
          write_flag=0;
        }
    }
  close(s);
}

void timer_func()
{



  if (global_timer[EXPLICIT])              // Explicit Connection timer
    {
      global_timer[EXPLICIT]--;             // decrement until zero
      if (global_timer[EXPLICIT] == 0) global_event |= EXPLICIT_TIMEOUT;
    }
```

```
  if (global_timer[IO_POLL])                // I/O Poll Connection timer
    {
      global_timer[IO_POLL]--;              // decrement until zero
      if (global_timer[IO_POLL] == 0) global_event |= IO_POLL_TIMEOUT;
    }
  if (global_timer[ACK_WAIT])               // Acknowlege message timer
    {
      global_timer[ACK_WAIT]--;                 // decrement until zero
      if (global_timer[ACK_WAIT] == 0) global_event |= ACK_WAIT_TIMEOUT;
    }
  if (global_timer[UPDATE])                 // Timer for updating the device
    {
      global_timer[UPDATE]--;                     // decrement until zero
      if (global_timer[UPDATE] == 0) global_event |= DEVICE_UPDATE;
    }
}
```

II.3 Anexo C - Código

```
#include <stdio.h>
#include <string.h>
#include <math.h>
//#include "DeviceNet.h"
#include "SCan.h"

int inputtest = 0x0000000000000001;

class DISCRETE_INPUT_POINT
{
private:
    BOOL value;
    UCHAR instance;
    BOOL status;
    ULONG sensor_clock;
    static UINT class_revision;
public:
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    UCHAR get_value(void);
    void init_obj(UCHAR);
};

void DISCRETE_INPUT_POINT::init_obj(UCHAR inst)
{
    instance = inst;
    sensor_clock= 0;
    value = 0;
    status = 0;
}

void DISCRETE_INPUT_POINT::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }

}

void DISCRETE_INPUT_POINT::handle_explicit(UCHAR request[], UCHAR response[])
{
```

```
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);
    printf("\nDISCRETE_INPUT_POINT_HANDLE_EXPLICIT\n");

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 3:  // value
            printf("\nDISCRETE_INPUT_POINT_HANDLE_EXPLICIT: GET_VALUE\n");
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = value;
            response[LENGTH] = 3;
            break;

        case 4:  // status
            printf("\nDISCRETE_INPUT_POINT_HANDLE_EXPLICIT: GET_STATUS\n");
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = (UCHAR)status;
            response[LENGTH] = 3;
            break;

        default:
            printf("\n\nATTRIBUTE_NOT_SUPPORTED\n\n");
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;

    default:
        printf("\n\nSERVICE_NOT_SUPPORTED\n\n");
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

UCHAR DISCRETE_INPUT_POINT::get_value(void)
{
    int count =0;
    value = 0;
    for(count=0;count<8;count++){
    }
    if((instance==1)||(instance==8)) value =1;
    return value;
}



class DISCRETE_OUTPUT_POINT
{
private:
    BOOL value;
    UCHAR instance;
    BOOL status;
    static UINT class_revision;
public:
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    void init_obj(UCHAR);
    void set_value(BOOL);

};

void DISCRETE_OUTPUT_POINT::init_obj(UCHAR inst)
{
```

```
    instance = inst;
    value = 0;
    status = 0;
}

void DISCRETE_OUTPUT_POINT::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);
    printf("\nDISCRETE_OUTPUT_POINT_HANDLE_CLASS_\n");

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }

}

void DISCRETE_OUTPUT_POINT::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);
    printf("\nDISCRETE_OUTPUT_POINT_HANDLE_EXPLICIT_\n");
    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 3:  // value
            printf("\nDISCRETE_OUTPUT_POINT_HANDLE_EXPLICIT:_GET_VALUE\n");
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = (UCHAR)value;
            response[LENGTH] = 3;
            break;

        case 4:  // status
            printf("\nDISCRETE_OUTPUT_POINT_HANDLE_EXPLICIT:_GET_STATUS_\n");
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = (UCHAR)status;
            response[LENGTH] = 3;
            break;
```

```
        default:
            printf("\n\nATTRIBUTE_NOT_SUPPORTED\n\n");
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;

    case SET_REQUEST:
        switch(attrib)
        {
        case 3: // set value
            printf("\nDISCRETE_OUTPUT_POINT_HANDLE_EXPLICIT: SET_VALUE\n");
            set_value(request[5]);
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[LENGTH] = 2;
            break;
        default:
            printf("\n\nSERVICE_NOT_SUPPORTED\n\n");
            error = SERVICE_NOT_SUPPORTED;
            break;
        }
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

void DISCRETE_OUTPUT_POINT::set_value(BOOL val)
{
    value = val;
}


class CONNECTION
{
private:
    static UINT class_revision;
    static UCHAR path_in[PATH_SIZE];
    static UCHAR path_out[PATH_SIZE];
    UCHAR state, instance, instance_type, transport_class_trigger; //required instance attributes (CIP v1-3.13)
    UINT produced_connection_id, consumed_connection_id;
    UCHAR initial_comm_characteristics;
    UINT produced_connection_size, consumed_connection_size;
    UINT expected_packet_rate;
    UCHAR watchdog_timeout_action;
    UINT produced_connection_path_length;
    UCHAR produced_connection_path[PATH_SIZE];
    UINT consumed_connection_path_length;
    UCHAR consumed_connection_path[PATH_SIZE];
    UINT production_inhibit_time;


    UCHAR rcve_index, xmit_index;
    UCHAR my_rcve_fragment_count, my_xmit_fragment_count;
    UCHAR xmit_fragment_buf[BUFSIZE];
    UCHAR rcve_fragment_buf[BUFSIZE];
    UCHAR ack_timeout_counter;

public:
    UCHAR get_state(void);
    void set_state(UCHAR);
    UCHAR get_timeout_action(void);
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    int link_consumer(UCHAR*);
    void link_producer(UCHAR*);
    CONNECTION(UCHAR inst)
    {
        instance = inst;
        if (instance == EXPLICIT) instance_type =0; //(CIP v1-3.15) instance type = 00 => Explicit Message; instance type = 01 => I/O Message
        else instance_type=1;                       //(CIP v1-3.15)
        set_state(NON_EXISTENT);
    }
```

```
};

UCHAR CONNECTION::get_state(void)
{
    return state;
}

UCHAR CONNECTION::get_timeout_action(void)
{
    return watchdog_timeout_action;
}


void CONNECTION::set_state(UCHAR st)
{
    state = st;
    if((instance == EXPLICIT)&&(state == ESTABLISHED))
    {
        transport_class_trigger = 0x83; // 1 (Dir=Serv) 000 (ProdTrig=Cyclic) 0011 (TranspClass=3 do not prepend a 16-bit sequence CIPv3-3.4) CIPv1-
        produced_connection_id = 0x5FB; // MessageGroup2 CIPv3-3.4 CIPv3-2.4
        consumed_connection_id = 0x5FC; // MessageGroup2 CIPv3-3.4 CIPv3-2.4
        initial_comm_characteristics = 0x21; // 2 = Produce across message group 2(source); 1 = Consume across message group 2(destination) CIPv3-6
        produced_connection_size = 80; // Message router maximum amount of data = 64 bit produced
        consumed_connection_size = 80; // Message router maximum amount of data = 8 bit consumed
        expected_packet_rate = 2500; // Default value (CIPv3-3.30)
        watchdog_timeout_action = 0x01; // auto delete (CIPv3-3.30)
        produced_connection_path_length = 0;
        consumed_connection_path_length = 0;
        memset(produced_connection_path, 0, 10);
        memset(consumed_connection_path, 0, 10);
        production_inhibit_time = 0;
        rcve_index = 0;
        xmit_index = 0;
        my_rcve_fragment_count = 0;
        my_xmit_fragment_count = 0;
        memset(rcve_fragment_buf, 0, BUFSIZE);
        memset(xmit_fragment_buf, 0, BUFSIZE);
        ack_timeout_counter = 0;
        global_timer[EXPLICIT] = (expected_packet_rate / 50) * 4;

    }
    if((instance == IO_POLL)&&(state == CONFIGURING))
    {

        transport_class_trigger = 0x82; // 1 (Dir=Serv) 000 (ProdTrig=Cyclic) 0010 (TranspClass=2 do not prepend a 16-bit sequence CIPv3-3.4) CIPv1-
        produced_connection_id = 0x3FF; // MessageGroup1 CIPv3-3.4 CIPv3-2.4
        consumed_connection_id = 0x5FD; // MessageGroup2 CIPv3-3.4 CIPv3-2.4
        initial_comm_characteristics = 0x01; // 0 = Produce across message group 1; 1 = Consume across message group 2(destination) CIPv3-6/7
        produced_connection_size = 8; // i/o poll response length in bytes (64 bits for 64 discrete inputs)
        consumed_connection_size = 1; // i/o poll request length in bytes (8 bits for 8 discrete outputs)
        expected_packet_rate = 0; // Default value (CIPv3-3.30)
        watchdog_timeout_action = 0x00; // go to timed-out (CIPv3-3.30)
        produced_connection_path_length = 6;
        consumed_connection_path_length = 6;
        memcpy(produced_connection_path, path_in, 6);
        memcpy(consumed_connection_path, path_out, 6);
        production_inhibit_time = 0;
        rcve_index = 0;
        xmit_index = 0;
        my_rcve_fragment_count = 0;
        my_xmit_fragment_count = 0;
        memset(rcve_fragment_buf, 0, BUFSIZE);
        memset(xmit_fragment_buf, 0, BUFSIZE);
        ack_timeout_counter = 0;

    }
    if (state == NON_EXISTENT)
    {
        global_timer[instance] = 0;    // stop connection timer
    }
}

// Handles Explicit request to the class
void CONNECTION::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);
```

```
    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

void CONNECTION::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;
    UINT test_val, temp;                                    // for EPR calculation

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:   // connection state
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = state;
            response[LENGTH] = 3;
            break;

        case 2:   // instance type
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = instance_type;
            response[LENGTH] = 3;
            break;

        case 3:   // transport class trigger
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = transport_class_trigger;
            response[LENGTH] = 3;
            break;

        case 4:   // produced connection id
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(produced_connection_id);
            response[3] = HIBYTE(produced_connection_id);
            response[LENGTH] = 4;
            break;

        case 5:   // consumed connection id
```

```
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(consumed_connection_id);
    response[3] = HIBYTE(consumed_connection_id);
    response[LENGTH] = 4;
    break;

case 6:   // initial comm characteristics
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = initial_comm_characteristics;
    response[LENGTH] = 3;
    break;

case 7:   // produced connection size
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(produced_connection_size);
    response[3] = HIBYTE(produced_connection_size);
    response[LENGTH] = 4;
    break;

case 8:          // consumed connection size
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(consumed_connection_size);
    response[3] = HIBYTE(consumed_connection_size);
    response[LENGTH] = 4;
    break;

case 9: // EPR
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(expected_packet_rate);
    response[3] = HIBYTE(expected_packet_rate);
    response[LENGTH] = 4;
    break;

case 12:   // watchdog timeout action
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = watchdog_timeout_action;
    response[LENGTH] = 3;
    break;

case 13: // produced conxn path length
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(produced_connection_path_length);
    response[3] = HIBYTE(produced_connection_path_length);
    response[LENGTH] = 4;
    break;

case 14:   // produced conxn path
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    if (instance_type == EXPLICIT_TYPE)
    {
        // Path is NULL for Explicit Connection
        response[LENGTH] = 2;
    }
    else  // Path for data sent in I/O Poll response
    {
        memcpy(&response[2], produced_connection_path, 6);
        response[LENGTH] = 8;
    }
    break;

case 15: // consumed conxn path length
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    response[2] = LOBYTE(consumed_connection_path_length);
    response[3] = HIBYTE(consumed_connection_path_length);
    response[LENGTH] = 4;
    break;

case 16:  // consumed conxn path
    response[0] = request[0] & NON_FRAGMENTED;
    response[1] = service | SUCCESS_RESPONSE;
    if (instance_type == EXPLICIT_TYPE)
    {
```

```
            // Path is NULL for Explicit Connection
            response[LENGTH] = 2;
        }
        else  // Path for data sent in I/O Poll response
        {
            memcpy(&response[2], consumed_connection_path, 6);
            response[LENGTH] = 8;
        }
        break;

    case 17:  // production inhibit time
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = service | SUCCESS_RESPONSE;
        response[2] = LOBYTE(production_inhibit_time);
        response[3] = HIBYTE(production_inhibit_time);
        response[LENGTH] = 4;
        break;


    default:
        error = ATTRIB_NOT_SUPPORTED;
        break;
    }
    break;


case SET_REQUEST:
    switch(attrib)
    {
    case 1:
    case 2:
    case 3:
    case 4:
    case 5:
    case 6:
    case 7:
    case 8:
    case 13:
    case 14:
    case 15:
    case 16:
    case 17:
        error = ATTRIB_NOT_SETTABLE;
        break;

    case 9: // Set EPR
        test_val = request[5] + 256 * request[6];

        // Limit EPR so rounding up doesn't take it over 65,535
        if (test_val > 65000) error = INVALID_ATTRIB_VALUE;
        else
        {
            // This tweaks the EPR to be a multiple of 50
            temp = test_val / 50;
            if ((temp * 50) != test_val) temp++;   // round up
            expected_packet_rate = temp * 50;

            // see if I/O Poll connection should be established
            if ((instance == IO_POLL) && (state == CONFIGURING))
            {
                set_state(ESTABLISHED);
                // IF EPR > 2500 use it, else use 2500 (200 counts)
                // This ensures that timeout time is at least 10 seconds
                if ((expected_packet_rate > 2500) ||
                        (expected_packet_rate == 0))
                {
                    global_timer[IO_POLL] = (expected_packet_rate / 50) * 4;
                }
                else global_timer[IO_POLL] = 200;
            }
            else global_timer[instance] = (expected_packet_rate / 50) * 4;

            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(expected_packet_rate);
            response[3] = HIBYTE(expected_packet_rate);
            response[LENGTH] = 4;
        }
        break;

    case 12:  // Set watchdog timeout action
```

```
            if (instance == IO_POLL) error = ATTRIB_NOT_SETTABLE;
            else if ((request[5] != 1) &&
                     (request[5] != 3)) error = INVALID_ATTRIB_VALUE;
            else
            {
                watchdog_timeout_action = request[5];
                response[0] = request[0] & NON_FRAGMENTED;
                response[1] = service | SUCCESS_RESPONSE;
                response[LENGTH] = 2;
            }
            break;


        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;

    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

int CONNECTION::link_consumer(UCHAR request[])
{
    UCHAR i, length, fragment_count, fragment_type;

    if ((instance == EXPLICIT) && (state == DEFERRED)) state = ESTABLISHED;
    if (state != ESTABLISHED) return NO_RESPONSE;

    global_timer[instance] = (expected_packet_rate / 50) * 4;    // restart connxn timer
    // Handle an I/O Poll request
    if (instance == IO_POLL)
    {
        // In I/O message master must send 8 bits (1 byte) of discrete outputs

        if (request[LENGTH] == 0x01) return 2;
        else if (request[LENGTH] == 0x00) return 1;
        else return NO_RESPONSE;


    }

    // From this point on we are dealing with an Explicit message
    // If it is not fragmented, reset frag counters to initial state
    // and process the current incoming message.
    if ((request[0] & 0x80) == 0)
    {
        rcve_index = 0;
        xmit_index = 0;
        my_rcve_fragment_count = 0;
        my_xmit_fragment_count = 0;
        ack_timeout_counter = 0;
        global_timer[ACK_WAIT] = 0;
        return OK;
    }

    // At this point we are dealing with either an explicit fragment
    // or an ACK to an explicit message we sent earlier
    fragment_type =  request[1] & 0xC0;
    fragment_count = request[1] & 0x3F;
    length = request[LENGTH];


    switch(fragment_type)
    {
    case ACK_FRAG:
        // Received an ack from the master to an explicit fragment I sent earlier
        // Send the request to the link producer along with a tag so it knows
        // what do do with the message
```

```
        request[MESSAGE_TAG] = RECEIVED_ACK;
        link_producer(request);
        break;


    case FIRST_FRAG:
        if (fragment_count != 0)
        {
            // Reset fragment counters to initial state and drop fragment
            rcve_index = 0;
            my_rcve_fragment_count = 0;
            return NO_RESPONSE;
        }

        // get rid of any existing fragments and process first fragment
        rcve_index = 0;
        my_rcve_fragment_count = 0;
        memset(rcve_fragment_buf, 0, BUFSIZE);
        rcve_fragment_buf[rcve_index] = request[0] & 0x7F; // remove fragment flag
        rcve_index++;
        for (i = 2; i < length; i++)       // do not copy fragment info
        {
            rcve_fragment_buf[rcve_index] = request[i];
            rcve_index++;
        }

        // Check to see if Master has exceeded byte count limit
        if (rcve_index > consumed_connection_size)
        {
            rcve_index = 0;                         // reset fragment counters
            my_rcve_fragment_count = 0;
            request[MESSAGE_TAG] = ACK_ERROR;
            link_producer(request);
        }
        else  // fragment checked out OK so send ACK
        {
            request[MESSAGE_TAG] = SEND_ACK;
            link_producer(request);
            my_rcve_fragment_count++;
        }
        break;



    case MIDDLE_FRAG:
        if (my_rcve_fragment_count == 0) return NO_RESPONSE;  // just drop fragment

        // See if Master has sent the same fragment again.  If so just send ACK
        if (fragment_count == (my_rcve_fragment_count - 1))
        {
            request[MESSAGE_TAG] = SEND_ACK;
            link_producer(request);
        }
        // See if received fragment count does not agree with my count
        // If so, reset to beginning
        else if (fragment_count != my_rcve_fragment_count)
        {
            rcve_index = 0;
            my_rcve_fragment_count = 0;
        }

        else  // Fragment was validated so process it
        {
            // Copy the fragment to reassembly buffer, omit first 2 bytes
            for (i = 2; i < length; i++)
            {
                rcve_fragment_buf[rcve_index] = request[i];
                rcve_index++;
            }
            // Check to see if Master has exceeded byte count limit
            if (rcve_index > consumed_connection_size)
            {
                rcve_index = 0;
                my_rcve_fragment_count = 0;
                request[MESSAGE_TAG] = ACK_ERROR;
                link_producer(request);
            }
            else  // send good ACK back to Master
            {
                request[MESSAGE_TAG] = SEND_ACK;
                link_producer(request);
```

```
                my_rcve_fragment_count++;
            }
        }
        break;



    case LAST_FRAG:
        if (my_rcve_fragment_count == 0) return NO_RESPONSE;  // just drop fragment


        // See if received fragment count does not agree with my count
        // If so, reset to beginning
        else if (fragment_count != my_rcve_fragment_count)
        {
            rcve_index = 0;
            my_rcve_fragment_count = 0;
        }
        else  // Fragment was validated, so process it
        {
            // Copy the fragment to reassembly buffer, omit first 2 bytes
            for (i = 2; i < length; i++)
            {
                rcve_fragment_buf[rcve_index] = request[i];
                rcve_index++;
            }

            // Check to see if Master has exceeded byte count limit
            if (rcve_index > consumed_connection_size)
            {
                rcve_index = 0;
                my_rcve_fragment_count = 0;
                memset(rcve_fragment_buf, 0, BUFSIZE);
                request[MESSAGE_TAG] = ACK_ERROR;
                link_producer(request);
            }
            else  // Fragment was OK, send ACK and reset everything
            {
                request[MESSAGE_TAG] = SEND_ACK;
                link_producer(request);
                // We are done receiving a fragmented explicit message
                // Copy from temporary buffer back to request buffer
                // and process complete Explicit message
                memcpy(request, rcve_fragment_buf, BUFSIZE);
                request[LENGTH] = rcve_index;
                rcve_index = 0;
                my_rcve_fragment_count = 0;
                memset(rcve_fragment_buf, 0, BUFSIZE);
                return OK;
            }
        }
        break;


    default:
        break;
    }

    return NO_RESPONSE;
}

void CONNECTION::link_producer(UCHAR response[])
{
    UCHAR length, bytes_left, i, fragment_count, ack_status;
    static UCHAR copy[BUFSIZE];

    length = response[LENGTH];

    if (instance == IO_POLL)
    {
        // load io poll response into can chip object #9
        for (i=0; i < length; i++)                                  // load CAN data
        {
            global_CAN_write[i] = response[i];
        }       // load config register
        global_CAN_write[LENGTH] = length;
        can_id_write = 0b01111111100;                               // set transmit request
        write_flag = 1;
    }
```

```
else if (response[MESSAGE_TAG] == ACK_TIMEOUT)
{
    ack_timeout_counter++;
    if (ack_timeout_counter == 1)
    {
        // Load last explicit fragment send again
        length = copy[LENGTH];
        for (i=0; i < length; i++)                                  // load data into CAN
        {
            global_CAN_write[i] = copy[i];
        }
        // load config resister
        global_CAN_write[LENGTH] = length;                          // set transmit request
        can_id_write= 0b10111100011;
        write_flag=1;
        global_timer[ACK_WAIT] = 20;

    }
    if (ack_timeout_counter == 2)
    {
        // abort trying to send fragmented message
        xmit_index = 0;
        my_xmit_fragment_count = 0;
        ack_timeout_counter = 0;
        global_timer[ACK_WAIT] = 0;
    }
}


else if (response[MESSAGE_TAG] == RECEIVED_ACK)
{
    fragment_count = response[1] & 0x3F;
    ack_status = response[2];
    if (my_xmit_fragment_count == fragment_count)
    {
        // If Master returned a bad ACK status, reset everything
        // and abort the attempt to send message
        if (ack_status != 0)
        {
            xmit_index = 0;
            my_xmit_fragment_count = 0;
            ack_timeout_counter = 0;
            global_timer[ACK_WAIT] = 0;
        }

        // Master's ACK was OK, so send next fragment unless we were done sending
        // Keep a copy of what we are sending
        else
        {
            if (xmit_index >= xmit_fragment_buf[LENGTH])
            {
                // got ACK to out final fragment so reset everything
                xmit_index = 0;
                my_xmit_fragment_count = 0;
                ack_timeout_counter = 0;
                global_timer[ACK_WAIT] = 0;
            }
            else
            {
                // Send another fragment
                // Figure out how many bytes are left to send
                bytes_left = xmit_fragment_buf[LENGTH] - xmit_index;
                my_xmit_fragment_count++;
                ack_timeout_counter = 0;
                global_timer[ACK_WAIT] = 20;                        // restart timer for 1 second
                // Load the first byte of the fragment
                copy[0] = response[0] | 0x80;
                can_id_write= 0b10111100011;
                global_CAN_write[0] = copy[0];
                if (bytes_left > 6)                 // this is a middle fragment
                {
                    copy[1] = MIDDLE_FRAG | my_xmit_fragment_count;
                    global_CAN_write[1] = copy[1];
                    length = 8;
                }
                else                                                // this is the last fragment
                {
                    copy[1] = LAST_FRAG | my_xmit_fragment_count;
                    global_CAN_write[1] = copy[1];
                    length = bytes_left + 2;
                }
```

```
                // Put in actual data
                for (i = 2; i < length; i++)  // put up to 6 more bytes in CAN chip
                {
                    copy[i] = xmit_fragment_buf[xmit_index];
                    global_CAN_write[i] = copy[i];
                    xmit_index++;
                }
                copy[LENGTH] = length;
                global_CAN_write[LENGTH] = length;
                can_id_write= 0b10111100011;
                write_flag = 1;
            }
        }
    }
}


// Send this message in response to receiving an explicit fragment
// from the Master that the link comsumer has validated as OK
else if (response[MESSAGE_TAG] == SEND_ACK)
{
    length = 3;                                                                    // This is a 3 byte message
    global_CAN_write[0] = (response[0] | 0x80);
    global_CAN_write[1] = (response[1] | ACK_FRAG);
    global_CAN_write[2] = 0;
    global_CAN_write[LENGTH] = length;
    write_flag = 1;
}



// Send this message in response to receiving an explicit fragment
// from the Master that exceeded the byte count limit for the connection
else if (response[MESSAGE_TAG] == ACK_ERROR)
{
    length = 3;                                                                    // This is a 3 byte message
    global_CAN_write[0] = (response[0] | 0x80);
    global_CAN_write[1] = (response[1] | ACK_FRAG);
    global_CAN_write[2] = 1;
    global_CAN_write[LENGTH] = length;
    can_id_write= 0b10111100011;
    write_flag = 1;
}



else if (length <= 8)               // Send complete Explicit message
{
    // load explicit response into can chip object #3
    for (i=0; i < length; i++)                                          // load data into CAN
    {
        global_CAN_write[i] = response[i];
    }
    global_CAN_write[LENGTH] = length;
    can_id_write= 0b10111100011; // load config resister
    write_flag = 1; // set transmit request
}



else if (length > 8)        // Send first Explicit message fragment
{
    // load explicit response into a buffer to keep around a while
    memcpy(xmit_fragment_buf, response, BUFSIZE);
    length = 8;
    xmit_index = 0;
    my_xmit_fragment_count = 0;
    ack_timeout_counter = 0;
    // Load first fragment into can chip object #3
    copy[0] = response[0] | 0x80;
    global_CAN_write[0] = copy[0];
    xmit_index++;
    // Put in fragment info
    copy[1] = FIRST_FRAG | my_xmit_fragment_count;
    global_CAN_write[1] = copy[1];
    // Put in actual data
    for (i = 2; i < 8; i++)  // put 6 more bytes in CAN chip
    {
        copy[i] = xmit_fragment_buf[xmit_index];
        global_CAN_write[i] = copy[i];
        xmit_index++;
```

```
        }
        copy[LENGTH] = length;
        global_CAN_write[LENGTH] =length;
        can_id_write= 0b10111100011; // load config resister
        write_flag=1; // set msg object transmit request
        global_timer[ACK_WAIT] = 20;                                    // start timer to wait for ack
    }
}

class ASSEMBLY
{
private:
    static UINT class_revision;
    UCHAR data[NUM_IN];
    UCHAR instance;

    DISCRETE_INPUT_POINT *inputsas;
    DISCRETE_OUTPUT_POINT *outputsas;

public:
    ASSEMBLY(DISCRETE_INPUT_POINT *inputsmain, DISCRETE_OUTPUT_POINT *outputmain, UCHAR ins)
    {
        inputsas = inputsmain;
        outputsas = outputmain;
        instance = ins;
    }
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    void update_data(void);
    void handle_io_poll_request(UCHAR*, UCHAR*);

};

// Handle an Explicit request directed to the class
void ASSEMBLY::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = FALSE;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // handle revision attribute of class
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}
```

```
// Handle an explicit request to the assembly object
void ASSEMBLY::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;
    UCHAR temp;
    service = request[1];
    attrib = request[4];
    error = FALSE;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 3:   // get data - array of 3 bytes
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = data[0];
            response[3] = data[1];
            response[4] = data[2];
            response[LENGTH] = 5;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

// Runs every 0.25 seconds to update the local copy
// This keeps the values up to date, and ready to send.
void ASSEMBLY::update_data(void)
{
    int count = 0;
    int quotient, remainder;
    memset(data,0,8);
    if(instance == INPUT_ASSEMBLY)
    {
        while(count<NUM_IN)
        {
            quotient = count/8;
            remainder = count%8;
            data[quotient] |= (inputsas[count].get_value()<<remainder);
            count++;
        }
    }
    // the assembly object keeps local copies of the values it needs to
    // send in the I/O POLL response.  It is faster that way.
    //data[0] = identity->get_state();
    //data[1] = temperature_sensor->get_value();
    //data[2] = humidity_sensor->get_value();
}



// Handle an I/O Poll request for data
void ASSEMBLY::handle_io_poll_request(UCHAR request[], UCHAR response[])
{
    BOOL val;
    int count;
    // This is an input assembly so it does not expect to receive data
    // from master. It returns 3 bytes of data to send in the io poll response
    if(instance == OUTPUT_ASSEMBLY)
    {
```

```
        for (count = 0; count<NUM_OUT; count++)
        {
            val = (request[0] & (0b00000001 << count));
            if (val) val = 0x01;
            else val = 0x00;
            outputsas[count].set_value(val);
        }
        memset(response, 0, BUFSIZE);
        response[LENGTH] = 0;
    }else if(instance == INPUT_ASSEMBLY)
    {
        memset(response, 0, BUFSIZE);
        memcpy(response, data, 8);
        response[LENGTH] = 8;
    }
}


class DEVICENET  // DeviceNet class
{
private:
    static UINT class_revision;
    // declare pointers of type CONNECTION to allow
    // sending messages to the connection objects
    CONNECTION *explicit_conxn, *io_poll_conxn;

public:
    UCHAR mac_id;
    ULONG serial;
    UCHAR baud_rate;
    UINT vendor_id;
    BOOL bus_off_int;
    struct
    {
        UCHAR choice;
        UCHAR my_master;
    } allocation;
    UCHAR physical_port;
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    int handle_unconnected_port(UCHAR*, UCHAR*);
    void handle_timeout(UCHAR);
    void handle_explicit(UCHAR*, UCHAR*);
    int consume_dup_mac(UCHAR*);
    void send_dup_mac_response(void);
    DEVICENET(UCHAR mac, UCHAR baud, UINT id, ULONG ser, CONNECTION *exp, CONNECTION *io)
    {
        explicit_conxn = exp;
        io_poll_conxn = io;
        mac_id = mac;
        baud_rate = baud;
        vendor_id = id;
        serial = ser;
        allocation.choice = 0;                                  // no connections
        allocation.my_master = DEFAULT_MASTER_MAC_ID;   // device not yet allocated
        bus_off_int = 0;
        physical_port = 0;
    }
};



// Handle Explicit request to class
void DEVICENET::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
```

```
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}



// When a connection timer expires, this function transitions the connection
// to the appropriate state.  After this, if neither connection is in
// the established state, it releases all connections and the device is no
// longer "owned" by the master.
void DEVICENET::handle_timeout(UCHAR conxn)
{
    UCHAR timeout_action;

    if (conxn == EXPLICIT_CONXN)
    {
        timeout_action = explicit_conxn->get_timeout_action();
        // If in auto-delete mode, or if in deferred delete mode
        // and I/O Poll conxn is not established, delete Explicit conxn.
        if ((timeout_action == 1) ||
                ((timeout_action == 3) && (io_poll_conxn->get_state() != ESTABLISHED)))
        {
            allocation.choice &= (~EXPLICIT_CONXN);
            explicit_conxn->set_state(NON_EXISTENT);
        }
        // If in deferred-delete mode and I/O Poll conxn is established
        // go to deferred state (still considered allocated)
        else if ((timeout_action == 3) && (io_poll_conxn->get_state() == ESTABLISHED))
        {
            explicit_conxn->set_state(DEFERRED);
        }
    }

    if (conxn == IO_POLL_CONXN)
    {
        allocation.choice &= (~IO_POLL_CONXN);
        io_poll_conxn->set_state(TIMED_OUT);
        // If Explicit is in deferred state, delete Explicit connnection
        if (explicit_conxn->get_timeout_action() == 3)
        {
            allocation.choice &= (~EXPLICIT_CONXN);
            explicit_conxn->set_state(NON_EXISTENT);
        }
    }

    // If neither connection is in established state, release everything
    if ((explicit_conxn->get_state() != ESTABLISHED) &&
            (io_poll_conxn->get_state() != ESTABLISHED))
    {
        explicit_conxn->set_state(NON_EXISTENT);
        io_poll_conxn->set_state(NON_EXISTENT);
        allocation.choice = 0;
        global_status &= (~OWNED);
        allocation.my_master = DEFAULT_MASTER_MAC_ID;
    }
}
```

```
// Handles a request to the unconnected port.  This will be a request
// from the Master to allocate or release connections.
int DEVICENET::handle_unconnected_port(UCHAR request[], UCHAR response[]) //CIPV3 Pag 3-15
{
    UINT service, error;
    UCHAR additional_code, master_mac_id, alloc_request, release_request;

    service = request[1];
    error = 0;
    additional_code = 0;
    memset(response, 0, BUFSIZE);

    // Note:  In this context, "allocated" means that a connection is in
    // either the CONFIGURING, ESTABLISHED, or DEFERRED states, as opposed
    // to being in the NON_EXISTENT or TIMED_OUT states

    switch(service)
    {
    case ALLOCATE_CONNECTIONS:       // request to allocate connections
        // first validate the request
        alloc_request = request[4];
        master_mac_id = request[5] & 0x3F;
        if ((master_mac_id != allocation.my_master) &&
                (allocation.my_master != DEFAULT_MASTER_MAC_ID))
        {
            printf("\n*******\n1\n********\n");
            error = OBJECT_STATE_CONFLICT;
            additional_code = ALLOCATION_CONFLICT;
        }
        else if (alloc_request == 0)     // master must allocate something
        {
            printf("\n*******\n2\n********\n");
            error = INVALID_ATTRIB_VALUE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        // see if the master sets ack bit but neither COS nor Cyclic are set
        else if ((alloc_request & 0x40) && ((alloc_request & 0x30) == 0))
        {
            printf("\n*******\n3\n********\n");
            error = INVALID_ATTRIB_VALUE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        // see if master tries to allocate something not supported
        else if (alloc_request & 0xFC)
        {
            printf("\n*******\n4\n********\n");
            error = RESOURCE_UNAVAILABLE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        // Master must at least allocate explicit unless it is already allocated
        else if (((alloc_request & EXPLICIT_CONXN) == 0) &&
                 ((allocation.choice & EXPLICIT_CONXN) == 0))
        {
            printf("\n*******\n5\n********\n");
            error = INVALID_ATTRIB_VALUE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        // Master must allocate either explicit or poll or both
        else if ((alloc_request & (EXPLICIT_CONXN | IO_POLL_CONXN)) == 0)
        {
            printf("\n*******\n6\n********\n");
            error = INVALID_ATTRIB_VALUE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        // check to see if Master is trying to allocate a connection already allocated
        else if (((alloc_request & EXPLICIT_CONXN) && (allocation.choice & EXPLICIT_CONXN)) ||
                 ((alloc_request & IO_POLL_CONXN) && (allocation.choice & IO_POLL_CONXN)))
        {
            printf("\n*******\n7\n********\n");
            error = ALREADY_IN_STATE;
            additional_code = INVALID_ALLOC_CHOICE;
        }
        else // passed validation tests
        {
            allocation.my_master = master_mac_id;
            global_status |= OWNED;

            // send message to explicit connection object
            if (alloc_request & EXPLICIT_CONXN)
            {
```

```
            allocation.choice |= EXPLICIT_CONXN;
            explicit_conxn->set_state(ESTABLISHED);
        }

        if (alloc_request & IO_POLL_CONXN)
        {
            allocation.choice |= IO_POLL_CONXN;
            io_poll_conxn->set_state(CONFIGURING);
        }

        // prepare response
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = service | SUCCESS_RESPONSE;
        response[2] = 0;    // 8/8 message format
        response[LENGTH] = 3;
    }
    break;



case RELEASE_CONNECTIONS:
    release_request = request[4];
    if (release_request == 0)
    {
        error = INVALID_ATTRIB_VALUE;
        additional_code = INVALID_ALLOC_CHOICE;
    }
    // must not try to release something we don't support
    else if (release_request & 0xFC)
    {
        error = RESOURCE_UNAVAILABLE;
        additional_code = INVALID_ALLOC_CHOICE;
    }
    // the connection must exist to be released
    else if (((release_request & EXPLICIT_CONXN) &&
              (explicit_conxn->get_state() == NON_EXISTENT)) ||
             ((release_request & IO_POLL_CONXN) &&
              (io_poll_conxn->get_state() == NON_EXISTENT)))
    {
        error = ALREADY_IN_STATE;
        additional_code = INVALID_ALLOC_CHOICE;
    }
    else // passed validation tests
    {
        if (release_request & EXPLICIT_CONXN)
        {
            allocation.choice &= (~EXPLICIT_CONXN);
            explicit_conxn->set_state(NON_EXISTENT);
        }
        if (release_request & IO_POLL_CONXN)
        {
            allocation.choice &= (~IO_POLL_CONXN);
            io_poll_conxn->set_state(NON_EXISTENT);
        }
        // If neither connection is in established state, release everything
        if ((explicit_conxn->get_state() != ESTABLISHED) &&
                (io_poll_conxn->get_state() != ESTABLISHED))
        {
            explicit_conxn->set_state(NON_EXISTENT);
            io_poll_conxn->set_state(NON_EXISTENT);
            allocation.choice = 0;
            global_status &= (~OWNED);
            allocation.my_master = DEFAULT_MASTER_MAC_ID;
        }

        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = service | SUCCESS_RESPONSE;
        response[LENGTH] = 2;
    }
    break;


default:
    error = RESOURCE_UNAVAILABLE;
    additional_code = INVALID_UNC_REQUEST;
    break;
}

if (error)
{
    response[0] = request[0] & NON_FRAGMENTED;
```

```
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = additional_code;
        response[LENGTH] = 4;
    }
    return OK;
}

// Handles Explicit request to the DeviceNet Object
void DEVICENET::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1: // get this device's mac id
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = mac_id;
            response[LENGTH] = 3;
            break;

        case 2: // get baud rate
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = baud_rate;
            response[LENGTH] = 3;
            break;

        case 3: // get bus off int value
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = bus_off_int;
            response[LENGTH] = 3;
            break;

        case 5: // get allocation information struct
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = allocation.choice;
            response[3] = allocation.my_master;
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    case ALLOCATE_CONNECTIONS:
        // request to allocate connections received through the message router
        // forward the request to the unconnected port handler
        handle_unconnected_port(request, response);
        break;


    case RELEASE_CONNECTIONS:
        // request to release connections received through the message router
        // forward the request to the unconnected port handler
        handle_unconnected_port(request, response);
        break;



    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
```

```
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}


// Send a DUP MAC check response message.  This is a message sent in
// response to receiving a DUP MAC check message (with my MAC ID) from
// another device which is hoping to go on-line.  Sending this message
// keeps the offending device from going on-line.
void DEVICENET::send_dup_mac_response(void)
{
    // put bytes into CAN chip msg object #7 and send
    // load CAN message config register - msg length = 7
    //pokeb(CAN_BASE, 0x76, 0x78);
    // load data area of CAN chip

    can_id_write = 0b10111100111;
    global_CAN_write[0] = 0x80;
    global_CAN_write[1] = LOBYTE(vendor_id);
    global_CAN_write[2] = HIBYTE(vendor_id);
    global_CAN_write[3] = (UCHAR)(serial);
    global_CAN_write[4] = (UCHAR)(serial >> 8);
    global_CAN_write[5] = (UCHAR)(serial >> 16);
    global_CAN_write[6] = (UCHAR)(serial >> 24);
    global_CAN_write[LENGTH] = 7;
    write_flag = 1;
}


// Consume an incoming DUP MAC check message from another device
int DEVICENET::consume_dup_mac(UCHAR request[])
{
    // If message is a response to our dup MAC ID check message
    // or if we are not on-line yet, error out.  Do not go on-line
    if (((request[0] & 0x80) != 0) ||            // message is a response
            ((global_status & ON_LINE) == 0))       // we are off line
    {
        global_status |= DUP_MAC_FAULT;         // set dup MAC error
        printf("\nDUPMACFAULT\n");
        return NO_RESPONSE;                             // send no response
    }
    return OK;                                  // message consumed successfully
}


class IDENTITY
{
private:
    static UINT class_revision;
    UINT vendor_id;
    UINT device_type;
    UINT product_code;
    struct
    {
        UCHAR major;
        UCHAR minor;
    } revision;

    ULONG serial;
    char product_name[40];
    ULONG device_clock, dup_mac_clock;
    CONNECTION *explicitcon, *io_poll;


public:
    UINT status;
    UCHAR state;
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    UCHAR get_state(void);
    void device_self_test(void);
    void send_dup_mac_check_message(void);
    void update_device(void);
    IDENTITY(UINT id, ULONG s, CONNECTION *ex, CONNECTION *io)  // constructor
    {
        device_clock = 0;
        dup_mac_clock = 0;
```

```
        vendor_id = id;
        serial = s;
        explicitcon = ex;
        io_poll = io;
        device_type = 0;                                // generic device
        product_code = 1;                       // product model within a device type
        revision.major = 1;                     // rev level of product model
        revision.minor = 0;
        status = 0;                             // status of the device
        state = 0;                      // state of the device
        memset(product_name, 0, 40);
        strcpy(product_name, "MODULO_IO");
    }
};



// Handle Explicit requests to class
void IDENTITY::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}


// Handle Explicit requests to Identity Object
void IDENTITY::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;
    UCHAR reset_type;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get ODVA vendor ID
            response[0] = request[0] & NON_FRAGMENTED;
```

```
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(vendor_id);
            response[3] = HIBYTE(vendor_id);
            response[LENGTH] = 4;
            break;

        case 2:  // get device type
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(device_type);
            response[3] = HIBYTE(device_type);
            response[LENGTH] = 4;
            break;

        case 3:  // get product code
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(product_code);
            response[3] = HIBYTE(product_code);
            response[LENGTH] = 4;
            break;

        case 4:  // get revision level
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = revision.major;
            response[3] = revision.minor;
            response[LENGTH] = 4;
            break;

        case 5:  // get summary status
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(status);
            response[3] = HIBYTE(status);
            response[LENGTH] = 4;
            break;

        case 6:  // get serial number - 4 byte
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = (UCHAR)(serial);
            response[3] = (UCHAR)(serial >> 8);
            response[4] = (UCHAR)(serial >> 16);
            response[5] = (UCHAR)(serial >> 24);
            response[LENGTH] = 6;
            break;

        case 7:  // get product name (short string format)
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = (UCHAR)strlen(product_name);
            strcpy_UCHAR(&response[3], product_name);
            response[LENGTH] = response[2] + 3;
            break;

        case 8:  // get device state
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = state;
            response[LENGTH] = 3;
            break;


        default:
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    case RESET_REQUEST:
        // If reset type is not specified, default to type zero
        if (request[LENGTH] == 4) reset_type = 0;
        else reset_type = request[4];

        if (reset_type == 0)                    // simulate off/on cycle
        {
            // return success response now, can't do it after reset
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
```

```
            response[LENGTH] = 2;
            global_event |= FULL_RESET;       // set reset bit
        }
        else if (reset_type == 1)
        {
            // reset to out-of-box condition, then simulate power off/on cycle
            // return success response before if can't do it after
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[LENGTH] = 2;
            global_event |= FULL_RESET;       // set reset bit
        }
        else  error = INVALID_PARAMETER;
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

// Returns the state of the device
UCHAR IDENTITY::get_state(void)
{
    return state;
}

// Handles device startup & LED states
void IDENTITY::device_self_test(void)
{
    //In case there is an test in the module
}

// Sends a DUP MAC ID check request.
// Two of these are sent during the startup sequence.
void IDENTITY::send_dup_mac_check_message(void)
{
    // Send first dup MAC ID check message
    // Put message into CAN chip msg object #7 and send
    // Load CAN message config register - msg length = 7
    //pokeb(CAN_BASE, 0x76, 0x78);
    printf("\n\_DUPMACREQUEST(SEND)\n");
    can_id_write = 0b10111100111;
    global_CAN_write[0] = 0x00;
    global_CAN_write[1] = LOBYTE(vendor_id);
    global_CAN_write[2] = HIBYTE(vendor_id);
    global_CAN_write[3] = (UCHAR)(serial);
    global_CAN_write[4] = (UCHAR)(serial >> 8);
    global_CAN_write[5] = (UCHAR)(serial >> 16);
    global_CAN_write[6] = (UCHAR)(serial >> 24);
    global_CAN_write[LENGTH] = 7;
    write_flag = 1;
}



// Handles operation of overall device, checks for errors, handles LEDs,
// This runs every 0.25 second.  If the startup self-test is OK,
// the Device sends two dup MAC ID check messages
void IDENTITY::update_device(void)
{

    // For first 2 seconds (8 clock ticks) do self-test
    if (device_clock < 8) device_self_test();
    else if ((dup_mac_clock <= 8) && ((global_status & DEVICE_FAULT) == 0) && ((global_status & NETWORK_FAULT) == 0))
    {
        if (global_status & LONELY_NODE) dup_mac_clock = 0;
        else
        {
            // This generates two dup mac check messages one second apart
            // If we are the only node on the network, keep sending
```

```
            switch(dup_mac_clock)
            {
            case 0:       // start
                send_dup_mac_check_message();
                break;

            case 4:       // 1 second has elapsed
                send_dup_mac_check_message();
                break;

            case 8:       // 2 seconds have elapsed
                global_status |= ON_LINE;
                printf("\n STATUS: ONLINE\n");
                break;
            }
            dup_mac_clock++;
        }
    }

    device_clock++;
}

class ROUTER
{
private:
    static UINT class_revision;
    // address of objects the router will need to send messages to

    DISCRETE_INPUT_POINT *discretein;
    DISCRETE_OUTPUT_POINT *discreteout;
    ASSEMBLY *assembly_in;
    ASSEMBLY *assembly_out;
    IDENTITY *identity;
    DEVICENET *devicenet;
    CONNECTION *explicito, *io_poll;

public:
    static void handle_class_inquiry(UCHAR*, UCHAR*);
    void handle_explicit(UCHAR*, UCHAR*);
    void route(UCHAR*, UCHAR*);
    ROUTER(DISCRETE_INPUT_POINT *in, DISCRETE_OUTPUT_POINT *out,IDENTITY *id, DEVICENET *dn, CONNECTION *ex, CONNECTION *io, ASSEMBLY *asin, ASSEMBL
    {
        // Initialize address of other objects
        discretein = in;
        discreteout = out;
        identity = id;
        devicenet = dn;
        explicito = ex;
        io_poll = io;
        assembly_in = asin;
        assembly_out = asout;
    }
};



// handle Explicit request to class
void ROUTER::handle_class_inquiry(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(service)
    {
    case GET_REQUEST:
        switch(attrib)
        {
        case 1:  // get revision attribute
            response[0] = request[0] & NON_FRAGMENTED;
            response[1] = service | SUCCESS_RESPONSE;
            response[2] = LOBYTE(class_revision);
            response[3] = HIBYTE(class_revision);
            response[LENGTH] = 4;
            break;

        default:
```

```
            error = ATTRIB_NOT_SUPPORTED;
            break;
        }
        break;


    default:
        error = SERVICE_NOT_SUPPORTED;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}



// Handles Explicit request to the Router Object
void ROUTER::handle_explicit(UCHAR request[], UCHAR response[])
{
    UINT service, attrib, error;

    service = request[1];
    attrib = request[4];
    error = 0;
    memset(response, 0, BUFSIZE);

    error = SERVICE_NOT_SUPPORTED;  // no services supported
    printf("\n\n ROUTER DOESNT SUPPORT THIS SERVICE\n\n");

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}



// Pass Explicit requests to appropriate object and get response
void ROUTER::route(UCHAR request[], UCHAR response[])
{
    UINT class_id, instance, error;
    int i = 1;
    class_id = request[2];
    instance = request[3];
    error = 0;
    memset(response, 0, BUFSIZE);

    switch(class_id)
    {
    case DISCRETE_INPUT_POINT_CLASS:
        if(instance == 0){
            printf("\n\n DISCRETE INPUT POINT CLASS\n\n");
            DISCRETE_INPUT_POINT::handle_class_inquiry(request, response);
        }
        else{
            while((i != -1)&&( i <= NUM_IN))
            {
                if(instance == i)
                {
                    printf("\n\n DISCRETE INPUT POINT INSTANCE %d\n\n",i);
                    discretein[i-1].handle_explicit(request, response);
                    i=-2;
                }
                i++;
            }
        }
        if(instance>NUM_IN)
            error = OBJECT_DOES_NOT_EXIST;
        break;
```

```
case DISCRETE_OUTPUT_POINT_CLASS:
    if(instance == 0){
        printf("\n\n DISCRETE OUTPUT POINT CLASS\n\n");
        DISCRETE_OUTPUT_POINT::handle_class_inquiry(request, response);
    }
    else{
        while((i != -1)&&( i <= NUM_OUT))
        {
            if(instance == i)
            {
                printf("\n\n DISCRETE OUTPUT POINT INSTANCE %d\n\n",i);
                discreteout[i-1].handle_explicit(request, response);
                i=-2;
            }
            i++;
        }
    }
    if(instance>NUM_IN)
        error = OBJECT_DOES_NOT_EXIST;
    break;

case ASSEMBLY_CLASS:
    switch(instance)
    {
    case 0:  // direct this to the class
        printf("\n\n ASSEMBLY CLASS\n\n");
        ASSEMBLY::handle_class_inquiry(request, response);
        break;

    case 1:  // direct this to instance 1
        printf("\n\n INPUT ASSEMBLY INSTANCE\n\n");
        assembly_in->handle_explicit(request, response);
        break;

    case 2:  // direct this to instance 2
        printf("\n\n OUTPUT ASSEMBLY INSTANCE\n\n");
        assembly_out->handle_explicit(request, response);
        break;

    default:
        error = OBJECT_DOES_NOT_EXIST;
        break;
    }
    break;



case IDENTITY_CLASS:
    switch(instance)
    {
    case 0:     // direct this to the class
        printf("\n\n IDENTITY CLASS\n\n");
        IDENTITY::handle_class_inquiry(request, response);
        break;

    case 1:     // direct this to instance 1
        printf("\n\n IDENTITY INSTANCE\n\n");
        identity->handle_explicit(request, response);
        break;

    default:
        error = OBJECT_DOES_NOT_EXIST;
        break;
    }
    break;


case DEVICENET_CLASS:
    switch(instance)
    {
    case 0:    // direct this to the class
        printf("\n\n DEVICENET CLASS\n\n");
        DEVICENET::handle_class_inquiry(request, response);
        break;

    case 1:    // direct this to instance 1
        devicenet->handle_explicit(request, response);
        break;

    default:
```

```
            error = OBJECT_DOES_NOT_EXIST;
            break;
        }
        break;


    case CONNECTION_CLASS:
        switch(instance)
        {
        case 0:                    // direct this to the class
            printf("\n\n CONNECTION CLASS\n\n");
            CONNECTION::handle_class_inquiry(request, response);
            break;

        case 1:          // direct this to instance 1
            printf("\n\n EXPLICIT CONNECTION INSTANCE\n\n");
            explicito->handle_explicit(request, response);
            break;

        case 2:          // direct this to instance 2
            printf("\n\n IO CONNECTION INSTANCE\n\n");
            io_poll->handle_explicit(request, response);
            break;

        default:
            error = OBJECT_DOES_NOT_EXIST;
            break;
        }
        break;


    case ROUTER_CLASS:
        switch(instance)
        {
        case 0:           // direct this to the class
            printf("\n\n ROUTER CLASS\n\n");
            handle_class_inquiry(request, response);
            break;

        case 1:      // direct this to instance 1
            printf("\n\n ROUTER INSTANCE\n\n");
            handle_explicit(request, response);
            break;

        default:
            error = OBJECT_DOES_NOT_EXIST;
            break;

        }
        break;


    default:  // no such class
        error = OBJECT_DOES_NOT_EXIST;
        break;
    }

    if (error)
    {
        response[0] = request[0] & NON_FRAGMENTED;
        response[1] = ERROR_RESPONSE;
        response[2] = error;
        response[3] = NO_ADDITIONAL_CODE;
        response[LENGTH] = 4;
    }
}

// Class Revisions
UINT CONNECTION::class_revision = 1;
UINT DISCRETE_INPUT_POINT::class_revision = 1;
UINT DISCRETE_OUTPUT_POINT::class_revision = 1;
UINT IDENTITY::class_revision = 1;
UINT ASSEMBLY::class_revision = 2;
UINT DEVICENET::class_revision = 2;
UINT ROUTER::class_revision = 1;

UCHAR CONNECTION::path_in[10] =  {0x20, 0x04, 0x24, 0x01, 0x30, 0x03};
// 0010 0000 0000 0100 / 0010 0100 0000 0001 / 0011 0000 0000 0011 (Apendix C - CIPv1)
// 001 - Logical Segment
// 000 - Class ID
// 00 - 8-bit logical address
```

```
// 00000100 - Class ID #4  => Assembly Object (CIPv1-5.43)
// 001 - Logical segment
// 001 - Instance ID
// 00 - 8-bit logical address
// 00000001 - Instance ID #1  => 1st instance of assembly class
// 001 - Logical Segment
// 100 - Attribute ID
// 00 - 8-bit logical address
// 00000011 - Attribute #3 => Data (CIPv1-5-44)

UCHAR CONNECTION::path_out[10] =  {0x20, 0x04, 0x28, 0x01, 0x30, 0x03};
// 0010 0000 0000 0100 / 0010 1000 0000 0001 / 0011 0000 0000 0011 (Apendix C - CIPv1)
// 001 - Logical Segment
// 000 - Class ID
// 00 - 8-bit logical address
// 00000100 - Class ID #4  => Assembly Object (CIPv1-5.43)
// 001 - Logical segment
// 010 - Instance ID
// 00 - 8-bit logical address
// 00000001 - Instance ID #1  => 1st instance of assembly class
// 001 - Logical Segment
// 100 - Attribute ID
// 00 - 8-bit logical address
// 00000011 - Attribute #3 => Data (CIPv1-5-44)

int main()
{

    UINT e; // Temporary variable that stores global_event
    int i;
    int assembly_type=0;
    UCHAR request[BUFSIZE];
    UCHAR response[BUFSIZE];
    DISCRETE_INPUT_POINT *input_point = new DISCRETE_INPUT_POINT[NUM_IN-1];
    DISCRETE_OUTPUT_POINT *output_point = new DISCRETE_OUTPUT_POINT[NUM_OUT-1];
    CONNECTION expl(EXPLICIT);
    CONNECTION io_poll(IO_POLL);
    ASSEMBLY assembly_in(input_point, output_point, INPUT_ASSEMBLY);
    ASSEMBLY assembly_out(input_point, output_point, OUTPUT_ASSEMBLY);
    DEVICENET devicenet(MAC_ID, BAUD_RATE, VENDOR_ID, SERIAL_ID, &expl, &io_poll);
    IDENTITY identity(VENDOR_ID, SERIAL_ID, &expl, &io_poll);
    ROUTER router(input_point, output_point, &identity,
                  &devicenet, &expl, &io_poll, &assembly_in, &assembly_out);

    for(UCHAR i=1; i<=NUM_IN; i++)
        input_point[i-1].init_obj(i); // sensor instance 1 is in[0]

    for(UCHAR i=1; i<=NUM_OUT; i++)
        output_point[i-1].init_obj(i); // sensor instance 1 is out[0]

    std::thread t1(&sock_can_read);
    std::thread t2(&sock_can_write);
    //std::thread t3(&timer_func);

    memset(global_CAN_buf, 0, BUFSIZE);
    global_timer[EXPLICIT] = 0;          // explicit connection timer
    global_timer[IO_POLL] = 0;           // io poll connection timer
    global_timer[ACK_WAIT] = 0;          // only used for sending fragmented msg
    global_timer[UPDATE] = 10;           // do first device update in 0.5 secs
    global_event = 0;                    // no events to service
    global_status = ON_LINE;


    while(1)
    {

        e = global_event;
        if ((e & IO_POLL_REQUEST) && !(e & (IO_POLL_REQUEST - 1)))
        {
            printf("\n IOPOLLREQUEST\n");
            global_event &= ~IO_POLL_REQUEST; // clear bit
            // must be online with no critical errors
            if ((global_status & ON_LINE) && !(global_status & NETWORK_FAULT))
            {
                // Get request message from global ISR buffer
                memcpy(request, global_CAN_buf, BUFSIZE);
                switch(io_poll.link_consumer(request))
                {
                    case 1:     // try to consume the request
                        printf("\t INPUT\n");
                        assembly_in.handle_io_poll_request(request, response);
```

```
                io_poll.link_producer(response);        // produce response
                break;
            case 2:
                printf("\t OUTPUT\n");
                assembly_out.handle_io_poll_request(request, response);
                io_poll.link_producer(response);        // produce response
                break;
        }
        printf("\n ASSEMBLY TYPE: %d \n",assembly_type);
    }
}


// Incoming Explicit request message
e = global_event;
if ((e & EXPLICIT_REQUEST) && !(e & (EXPLICIT_REQUEST - 1)))
{
    printf("\n EXPLICITREQUEST\n");
    global_event &= ~EXPLICIT_REQUEST; // clear bit
    // must be online with no critical errors
    if ((global_status & ON_LINE) && !(global_status & NETWORK_FAULT))
    {
        // make sure message is from my master
        if ((global_CAN_buf[0] & 0x3F) == devicenet.allocation.my_master)
        {
            // Get request message from global ISR buffer
            memcpy(request, global_CAN_buf, BUFSIZE);
            if (expl.link_consumer(request))    // try to consume the request
            {
                router.route(request, response);        // route request & get response
                expl.link_producer(response);   // produce response
            }
        }
    }
}


e = global_event;
if ((e & DUP_MAC_REQUEST) && !(e & (DUP_MAC_REQUEST - 1))) // Incoming message from another device
{
    printf("\n DUPMACREQUEST(RECEIVED)\n");
    global_event &= ~DUP_MAC_REQUEST; // Clear bit on global event
    memcpy(request, global_CAN_buf, BUFSIZE);
    if(devicenet.consume_dup_mac(request)) // If receives a dup mac request with the same macid, respond to give error
    {
        devicenet.send_dup_mac_response();
    }

}

// Incoming message to the Unconnected Port
e = global_event;
if ((e & UNC_PORT_REQUEST) && !(e & (UNC_PORT_REQUEST - 1)))
{
    printf("\n UNCPORTREQUEST\n");
    global_event &= ~UNC_PORT_REQUEST;                          // clear bit
    // must be online with no critical errors
    if ((global_status & ON_LINE) && !(global_status & NETWORK_FAULT))
    {
        // Get request message from global ISR buffer
        memcpy(request, global_CAN_buf, BUFSIZE);
        if (devicenet.handle_unconnected_port(request, response));
        {
            // Note: I am not using link producer to send response because
            // the Unconnected port should have its own direct connection to
            // the network - So load response directly into can chip object #3
            can_id_write= 0b10111100011;
            global_CAN_write[LENGTH] = response[LENGTH];
            for (i=0; i < global_CAN_write[LENGTH]; i++)                                    // load data into CAN
            {
                global_CAN_write[i] = response[i];
            }
            write_flag = 1;
        }
    }
}

// Timeout of the Explicit connection
e = global_event;
if ((e & EXPLICIT_TIMEOUT) && !(e & (EXPLICIT_TIMEOUT - 1)))
{
```

```
            global_event &= ~EXPLICIT_TIMEOUT;  // clear bit
            devicenet.handle_timeout(EXPLICIT_CONXN);
        }


        // Timeout of the I/O Poll connection
        e = global_event;
        if ((e & IO_POLL_TIMEOUT) && !(e & (IO_POLL_TIMEOUT - 1)))
        {
            global_event &= ~IO_POLL_TIMEOUT;   // clear bit
            devicenet.handle_timeout(IO_POLL_CONXN);
        }

        // Timeout waiting for an acknowlege message(to a fragment I sent)
        e = global_event;
        if ((e & ACK_WAIT_TIMEOUT) && !(e & (ACK_WAIT_TIMEOUT - 1)))
        {
            global_event &= ~ACK_WAIT_TIMEOUT;
            // Notify link producer that we have timed-out while waiting for an ACK
            request[MESSAGE_TAG] = ACK_TIMEOUT;
            expl.link_producer(request);
        }

        e = global_event;
        if ((e & FULL_RESET) && !(e & (FULL_RESET - 1)))
        {
            global_event &= ~FULL_RESET;
            //reset();
        }
        assembly_in.update_data();
        //identity.update_device();
    }
    return 0;
}


/*
 *
 * TEST
 *
 * 5E6#0A4B0301010A
 *      10 111100 110 - Group 2 MACID60 msgID6 (Unconnected Explicit)
 *      0A - Frag0 Xid0 MACID0A
 *      4B - RR0 ServiceCode(AllocMasterSlave)4B
 *      03 - ClassID(DeviceNet)
 *      01 - InstanceID
 *      01 - AllocationChoice (Explicit)
 *      0A - AllocatorMACID
 *
 * 5E3#0ACB00
 *      10 111100 011 - Group2 MACID60 msgID3 (Slave Explicit/Unc Response)
 *      0A - Frag0 Xid0 MACID0A
 *      CB - RR1 ServiceCode 4B
 *      00 - ReservedBits MessageBody
 *
 * 5E6#0A4B0301020A
 * 5E3#0ACB00
 *
 * 5E4#0A100502090010
 *
 * 5E5#
 *
 * 5E5#01
 *
 * 5E4#0A0E080103
 *
 * 5E4#0A0E090103
 *
 * 5E4#0A1009010301
 *
 * 5E4#0A0E090103
 *
*/
```

II.4 Anexo D - Configuração do Driver MCP 2515

Para se fazer o Raspberry Pi reconhecer o módulo CAN é necessário instalar o Driver correspondente. Para se instalar o módulo é necessário ativar o modo SPI, para isso basta acessar o arquivo `/boot/config.txt` e descomentar a seguinte linha

```
dtparam=spi=on
```

e depois adicionar ao final do arquivo as linhas

```
dtoverlay=mcp2515-can0,oscillator=8000000,interrupt=25
dtoverlay=spi-bcm2835-overlay
```

Para estas linhas adicionadas, está sendo configurado o dispositivo para um cristal de oscilação de 8 MHz (8000000 Hz) e o interrupt estará ligado ao GPIO25 (Pino 22 do Raspberry Pi 3 Model B). Desta forma os pinos de GPIO estão configurados para receber o sinal CAN.

Para instalar o driver CAN basta abrir o terminal e colocar

```
$ sudo apt-get install can-utils
```

Desta forma se instala o driver e também o SocketCAN. Agora basta adicionar os módulos e depois reiniciar:

```
$ sudo modprobe can
$ sudo modprobe can_dev
$ sudo modprobe can_raw
```

Finalizada todas as configurações, basta ativar o CAN pelo comando (como a rede do Scanner DeviceNet disponível é de 125k baud definiu-se o bitrate de 125000)

```
$ sudo ip link set can0 up type can bitrate 125000 loopback on
```

Para realizar o monitoramento do tráfego do cabo CAN, basta utilizar a função `candump can0` e para enviar dados, `cansend can0`.

SocketCAN

Por último, é necessário definir uma interface entre o C++ e o CAN. Para isso usa-se um socket API oferecido pelo SocketCAN. Através desta API, é possível criar sockets do tipo CAN e utilizar como um socket UDP, através de comandos *read* e *write* [13].

A estrutura de dados definida para o frame básico do CAN é definido em `include/linux/can.h`:

```
struct can_frame{
        canid_t can_id;  /* 32 bit CAN_ID + EFF/RTR/ERR flags */
        __u8    can_dlc; /* frame payload length in byte */
        __u8    __pad;   /* padding */
        __u8    __res0;  /* reserved / padding */
        __u8    __res1;  /* reserved / padding */
        __u8    data[8] __attribute__((aligned(8)));
};
```

Para se compilar um programa que faz uso da biblioteca wiringPi utilizando o compilador g++, é necessário adicionar a flag `-lwiringPi`. Para a biblioteca pthread, utiliza-se a flag `-pthread`

II.5 Anexo E - Procedimentos para identificação do erro CAN

É possível utilizar o comando `candump can0` para se monitorar a rede CAN. Através do monitoramento é possível observar as mensagens enviadas pela CLP e pelo Raspberry.

Através do comando `cansend can0` é possível transmitir dados pela rede CAN. Porém, ao se transmitir no barramento através do circuito produzido, o controlador ia para o modo *bus-off*. Para monitorar o estado CAN, utiliza-se o comando `sudo ip -s -d link show can0`. Quando utilizado antes de se enviar uma mensagem, obtém o resultado da figura II.1

```
pi@raspberrypi:~ $ sudo ip -s -d link show can0
3: can0: <NOARP,UP,LOWER_UP,ECHO> mtu 16 qdisc pfifo_fast state UNKNOWN mode DEFAULT group defau
lt qlen 10
    link/can  promiscuity 0
    can state ERROR-ACTIVE restart-ms 0
          bitrate 125000 sample-point 0.875
          tq 500 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
          mcp251x: tseg1 3..16 tseg2 2..8 sjw 1..4 brp 1..64 brp-inc 1
          clock 4000000
          re-started bus-errors arbit-lost error-warn error-pass bus-off
          0         0         0         1         1         0         numtxqueues 1 numrxqu
eues 1 gso_max_size 65536 gso_max_segs 65535
    RX: bytes  packets  errors  dropped overrun mcast
    4798       8597     0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

Figura II.1: Resposta do comando ip -s -d link show can0 antes de enviar um cansend

Após o envio de uma mensagem, verificava-se o que se encontra na figura II.2

```
pi@raspberrypi:~ $ sudo ip -s -d link show can0
3: can0: <NO-CARRIER,NOARP,UP,ECHO> mtu 16 qdisc pfifo_fast state DOWN mo
de DEFAULT group default qlen 10
    link/can  promiscuity 0
    can state BUS-OFF restart-ms 0
          bitrate 125000 sample-point 0.875
          tq 500 prop-seg 6 phase-seg1 7 phase-seg2 2 sjw 1
          mcp251x: tseg1 3..16 tseg2 2..8 sjw 1..4 brp 1..64 brp-inc 1
          clock 4000000
          re-started bus-errors arbit-lost error-warn error-pass bus-off
          0          0          0          2          2          1
  numtxqueues 1 numrxqueues 1 gso_max_size 65536 gso_max_segs 65535
    RX: bytes  packets  errors  dropped overrun mcast
    38535      76078    0       0       0       0
    TX: bytes  packets  errors  dropped carrier collsns
    0          0        0       0       0       0
```

Figura II.2: Resposta do comando ip -s -d link show can0 após enviar um cansend

Desta forma, conclui-se que há um erro de transmissão no protocolo CAN. O primeiro procedimento tomado foi verificar se todos os componentes estavam em bom funcionamento (existia a possibilidade de algum componente eletrônico estar com defeito), desta forma, remontamos todo o circuito em uma *protoboard* com peças novas, porém o erro persistiu.

Também foi realizado um teste utilizando uma fonte de tensão externa, na possibilidade que a fonte do

Raspberry Pi pudesse estar saturando a corrente e impossibilitando o funcionamento correto do controlador CAN.

Observou-se através de um osciloscópio que o controlador MCP2515 estava enviando dados ao barramento, porém, não era possível identificar se era o dado correto.

Para realizar a depuração deste problema, será necessário fazer a análise dos dados sem o uso do *driver* fornecido pelo SocketCAN, pois ele não permite fazer a análise dos *frames* de erros gerados pelo CAN.

Portanto, se for necessário manter a configuração do MCP2515 e MCP2551, será necessário a criação de um *driver* próprio que permita analisar o que está ocorrendo por trás da lógica implementada no controlador e depurar o erro de transmissão de dados.