UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
DEPARTAMENTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO
CURSO DE BACHARELADO EM INFORMÁTICA

O protocolo CAN como solução para aplicações distribuídas, baseadas em objetos, entre PCs e microcontroladores

por

MARCO KASDORF HUBERT

Projeto de Diplomação

Prof. Cláudio Vianna Villela
Orientador

Pelotas, janeiro de 2001

Sumário

Lista de Abreviaturas

CAN	*Controller Area Network*
CRC	*cyclic redundance check*
CSMA/CR	*Carrier Sense Multiple Access with Collision Resolution*
CPU	*central processing unit*
DLL	*dynamic link library*
FIP	*factory instrumentation protocol*
fig.	figura
FPGA	*field programmable gate array*
HW	*hardware*
I/O	*input/output*
IDE	*identifier extension bit*
ISA	*industry standard architecture*
ISO	*international standard organization*
LLC	*logical link control*
LON	*local operation network*
LOO	linguagens orientadas a objeto
MAC	*medium access control*
Mb	*megabyte* (aproximadamente 1 milhão de bytes)
Mbit	*megabit* (aproximadamente 1 milhão de bits)
OO	orientação a objetos
OSI	*open system interconnection*
PCs	*Personal Computer(s)*
P-Net	*process network*
POO	programação orientada a objetos
Profibus	*process fieldbus*
RAM	*random access memory*
ROM	*read only memory*
RTR	*remote transmission request*
seg	segundo
SRR	*substitute remote request*
SW	*software*
TCP	*transfer control protocol*

Lista de Figuras

Lista de Tabelas

Resumo

O presente trabalho inicia por um estudo dos modelos de comunicação utilizados normalmente em modernos sistemas de automação industrial. Após esta apresentação, são apresentadas as características do protocolo CAN para a implementação do modelo de comunicação escolhido, o *multicast ou* comunicação grupal. Para a validação do modelo estudado é desenvolvido um protótipo de aplicação distribuída, utilizando-se conceitos de orientação a objetos. Como os sistemas de automação industrial são formados, em sua grande maioria, por microcomputadores e microcontroladores, e como não existem linguagens orientadas a objeto para a maioria dos microcontroladores disponíveis no mercado, é elaborado um estudo complementar de como implementar os conceitos da OO na programação dos mesmos.

1. INTRODUÇÃO

Os grandes avanços tecnológicos nas áreas de microeletrônica e engenharia de software impulsionaram o surgimento de produtos com melhor desempenho, maior versatilidade e menor custo. Estas características acompanham uma tendência mundial de descentralização das aplicações. Neste novo modelo, as novas aplicações devem possuir partes mais independentes, mas com a possibilidade de compartilhamento de serviços e informações com as demais. Isto nos leva a um modelo distribuído, onde não apenas as informações são distribuídas, mas o conhecimento da aplicação.

Um importante exemplo é o dos sistemas de automação industrial, principalmente pelas novas exigências de controle, distribuição e armazenamento de informações impostas pelo mercado. Estas exigências podem ser traduzidas em uma maior interoperabilidade entre plataformas e sistemas diferentes, como bancos de dados e a Internet, e uma maior flexibilidade dos sistemas a futuras expansões ou retrações do mercado. Por exemplo, em uma planta industrial o mesmo circuito que é responsável pela aquisição de um sinal (ex.: sensor de temperatura), pode ser responsável pelo tratamento desta informação e/ou acionamento de algum outro dispositivo (ex.: alarme).

Todas estas características somadas às inerentes a este tipo de sistema, levam a utilização de meios de comunicação especiais entre as partes da aplicação. Desde meados dos anos 80, vários protocolos de comunicação para sistemas de automação foram desenvolvidos (alguns ainda estão em desenvolvimento), tais como Profibus, Interbus-S, P-Net, LON, FIP e CAN.

O protocolo CAN vem sendo aplicado em diversas áreas como meio de comunicação de suporte a sistemas automatizados. Atualmente, a área de automação de veículos (veículos inteligentes) é o principal mercado utilizador deste protocolo, mas o mesmo também pode ser utilizado para o controle de motores, sensores e atuadores inteligentes.

Diversos autores assim como Yourdon [YOU 96], Booch [BOO 91] e Rumbaugh [RUM 94], sugerem que para resolver problemas com tantas características e restrições, deve-se utilizar os conceitos da orientação a objetos na concepção da solução. O problema é que quando se trabalha em sistemas de automação industrial, obrigatoriamente terá que se trabalhar com programação para microcontroladores, os quais, na sua maioria, não possuem compiladores OO.

O objetivo deste trabalho é o de realizar um estudo sobre os principais modelos de comunicação utilizados em sistemas de automação industrial, e como implementa-los. Para a obtenção destes objetivos será necessário realizar estudos complementares sobre o funcionamento de um protocolo de comunicação industrial, e de como é possível trabalhar com conceitos de OO em uma linguagem estruturada. Para a validação deste modelo será montado um protótipo de uma aplicação.

Para isto, esta monografia está organizada da seguinte maneira. Primeiramente, no capítulo 2, são apresentados os sistemas de automação, definindo seus componentes básicos e apresentando os dois principais modelos de comunicação existentes. Logo depois, no capítulo 3, é descrito o protocolo CAN, que implementa o modelo de comunicação *multicast*, amplamente usado nos sistemas de automação industrial. No

capítulo 4, são apresentados os diferentes componentes de hardware (placa CAN para PCs e microcontroladores) envolvidos na construção do protótipo de aplicação distribuída descrito no capítulo 6. O capítulo seguinte traz uma metodologia de programação baseada nos conceitos de OO para ser aplicada, através da linguagem C, na programação dos microcontroladores. Isto porque não existem linguagens orientadas a objeto para a maioria dos microcontroladores. O capítulo 6, como acima mencionado, descreve um protótipo de aplicação distribuída desenvolvido a fim de validar o estudo feito, e finalmente, no capítulo 7, são apresentadas as conclusões.

2. SISTEMAS DE AUTOMAÇÃO

Os sistemas de automação industrial atualmente em operação caracterizam-se por um grande número de dispositivos sensores e atuadores interligados. Estes dispositivos oferecem informações sobre o estado do sistema e permitem aos controladores atuarem para modificar o estado do mesmo. A natureza deste tipo de sistema tende a impor requisitos temporais para tomadas de decisões e de reação a determinados eventos. Desta forma, os sistemas de automação encontram-se inseridos no contexto dos sistemas distribuídos de tempo real [FRI 97].

Uma forma de minimizar os problemas inerentes de sistemas distribuídos [TAN 92], como o *overhead* no meio físico de comunicação, é criar um sistema que não apenas distribua informações brutas (não tratadas), mas também que tenha o conhecimento de como funciona a aplicação, definindo obrigações e restrições para cada elemento. Por exemplo, um sensor que é responsável pela aquisição do sinal, pode também converter o valor bruto no valor de engenharia, bem como sinalizar alarmes. Esta solução diminui o número de informações que irão transitar pela rede, o que permite respostas mais rápidas na comunicação dos nodos.

A evolução da indústria da microeletrônica fez com que componentes de alto desempenho, como microcontroladores, microprocessadores, memórias, sensores e atuadores, tivessem seu custo reduzido a patamares suficientemente baixos para incentivar o desenvolvimento de dispositivos autônomos [BRU 00]. Neste sentido pode-se idealizar sistemas onde, ao invés de grandes controladores coordenando dezenas ou centenas de dispositivos de I/O, tem-se dispositivos de I/O com capacidade de controlar e coordenar operações de forma sincronizada entre eles.

Este enfoque favorece a ampliação e manutenção das aplicações, além de favorecer a implementação de aplicações com processamento paralelo (desempenho) e/ou redundante (tolerância a falhas). Mas ele também obriga a definição de garantias de sincronização, segurança e integridade das informações disponíveis por todos os elementos distribuídos, o que aumenta a complexidade da aplicação [TAN 92].

Os componentes desses sistemas podem ser das mais diferentes plataformas de software/hardware. Em um mesmo sistema, pode-se ter, por exemplo, PCs, sensores/atuadores microcontrolados, ou qualquer outro tipo de componente. Ainda, os PCs podem ter diferentes sistemas operacionais, como Linux, Windows, Unix, etc., e os microcontroladores podem ser gerenciados por aplicações distintas. Desta forma, os sistemas de automação podem ser reclassificados como sistemas heterogêneos distribuídos de tempo real. Neste contexto, um aspecto deve ser comum em todos os componentes do sistema, o protocolo de comunicação (ver 2.1 A Comunicação em Sistemas Distribuídos).

Um exemplo de sistema de automação distribuído é um sistema de controle de temperatura de silos. Pode-se ter um PC que monitore todo o sistema, e diversos sensores instalados em pontos estratégicos dos silos, controlando a temperatura no interior dos mesmos. O sistema pode ser simples, como, por exemplo, todos os sensores controlados centralizadamente por um PC (ou outro tipo de controlador). Desta forma, o

PC é responsável por todo o processamento de informações, desde o conversão analógica/digital dos dados coletados pelos sensores até o tratamento dessas informações. Mas o sistema pode, também, ser mais complexo, onde sensores inteligentes (microcontrolados) estão ligados em rede com um PC (responsável pela apresentação dos dados ao usuário). Nesta implementação, a transformação analógico/digital citada acima e todo o tratamento desta informação são feitos diretamente no microcontrolador, que já envia o dado tratado ao PC (quando necessário). No PC, pode rodar o sistema operacional Windows, por exemplo, enquanto os microcontroladores (que controlam os sensores) possuem um software específico que os gerencia. É necessário, também, um conjunto de regras que discipline a comunicação entre os controladores e o PC, isto é, um protocolo de comunicação.

2.1 A COMUNICAÇÃO EM SISTEMAS DISTRIBUÍDOS

No ambiente industrial normalmente os elementos distribuídos de uma aplicação podem estar alguns quilômetros distantes entre si, sofrendo muitas interferências eletromagnéticas geradas por motores, sistemas de solda, etc. Estas características obrigam que o meio de comunicação seja serial, por causa da distância, digital, por causa das interferências e de alta velocidade, por causa do grande número de mensagens que circulam pelo meio de comunicação. Além disso, a topologia para a interligação dos dispositivos é o barramento, pois qualquer outro modelo se torna inviável pelo gigantesco número de ligações e conseqüente cabeamento que deve ser implementado.

2.1.1. Divisão em camadas

No modelo de distribuição dos sistemas automatizados, cada elemento é autônomo, ou seja, possui seu conjunto próprio de recursos, distribuindo apenas os resultados da utilização destes recursos. Sendo assim toda a comunicação ocorrerá pela troca e sincronização de mensagens.

Apesar disto parecer relativamente simples, é necessário que os participantes da troca de mensagens (emissor, receptor) concordem quanto ao significado dos bits enviados de um a outro. Existe a necessidade de que ambos concordem em diversos níveis, isto é, desde os bits de mais baixo nível, como detalhes da transmissão dos bits, até os de mais alto nível, como aqueles que tratam de como a informação deve ser expressa [TAN 92].

Para que se possa fazer o tratamento adequado em cada camada, a ISO (*International Standard Organization*) criou o modelo OSI (*Open System Interconnection*), um modelo de referência que identifica claramente 7 níveis envolvidos em um processo de comunicação, apontando quais os tipos de trabalhos que devem ser realizados por cada um dos níveis, como mostra a fig. 2.1.

Nível de aplicação	São as aplicações propriamente ditas, que fazem a interação com os usuários, como clientes de correio eletrônico, transferência de arquivos, ou conexões remotas a computadores diversos da rede.
Nível de apresentação	Ao contrário do níveis mais baixos, que tratam da transferência de bits, este nível é responsável por dar forma estruturada às informações
Nível de sessão	Versão aprimorada do nível de transporte, permitindo controle de diálogo, determinando qual das partes está transmitindo, para que se possa recomeçar transmissões onde as conexões eventualmente foram perdidas
Nível de transporte	Garante a consistência na transmissão de dados entre origem e destino, mesmo que isso implique em separação das informações em diversos pacotes, que sigam por rotas diferentes
Nível de rede	Responsável pelo roteamento dos quadros, para que os mesmos cheguem a seus destinos corretamente, seja em uma rede local, ou em uma rede de longa distância
Nível de enlace de dados	Agrupa os bits em quadros e controla a transmissão dos dados, realizando o tratamento de erros necessário
Nível físico	Trata especificamente da transmissão de 0's e 1's lógicos, preocupando-se com a taxa de transmissão e se a transmissão pode ser feita nos dois sentidos simultaneamente

FIGURA 2.1 - Modelo de referência OSI

Com base neste modelo são projetados os mais diversos de protocolos de comunicação. Um protocolo pode atuar em uma camada específica, como o TCP, que trabalho no nível de transporte, ou o IP, que trata da camada de rede. Mas pode também regulamentar mais de uma camada, como é o caso do CAN (ver capítulo 3 O Protocolo CAN), que define características para as duas primeiras camadas (física e de enlace de dados).

Cabe salientar ainda que existem dois tipos genéricos de protocolos: os orientados à conexão, em que antes da transmissão transmissor e receptor estabelecem uma conexão explícita, e os sem conexão, em que não há necessidade de procedimentos anteriores à troca de mensagens.

2.1.2. Modelos de comunicação

Existem dois principais modelos de comunicação usados em sistemas de automação:

- Modelo cliente-servidor:

Neste modelo, o transmissor estabelece uma conexão com o receptor, conhecida como "canal de bits" e inicia a transmissão. Não há nenhum problema aparente com este modelo, mas acontece um *overhead* considerável, já que existe um acúmulo excessivo de bits em cada quadro de dados transmitido porque cada camada do modelo OSI acrescenta um cabeçalho ao quadro. É por esta razão que a maioria dos sistemas de automação não usa todas as camadas definidas no modelo, e sim somente um subconjunto das mesmas.

- *Multicast*

O modelo anterior parte do princípio de que para que haja a comunicação deve existir um transmissor e um receptor. Acontece que, em determinadas circunstâncias, a comunicação envolve muito mais do que dois elementos. Um sistema de automação é um exemplo típico: podem existir dezenas (até centenas) de sensores microcontrolados espalhados por uma fábrica, e se deseja enviar a mesma informação a todos eles. Pelo modelo cliente-servidor, seria necessário enviar a mesma mensagem para cada sensor. Demoraria muito mais para que todos recebessem e, é claro, não haveria jeito de que todos processassem a informação ao mesmo tempo. O mesmo não acontece no modelo *multicast*, ou comunicação grupal, onde pode-se enviar uma mesma mensagem a vários componentes de uma rede com garantia de que todos processarão a mesma mensagem, dando um maior controle sobre o tempo-real e utilizando de uma forma mais otimizada o barramento, já que em um sistema de automação há grandes chances de haverem diversos componentes similares, que necessitem receber as mesmas mensagens ao mesmo tempo. Para implementar este tipo de comunicação é preciso haver um mecanismo de filtragem de mensagens, ao nível de hardware, como ocorre no protocolo CAN, estudado neste trabalho.

É preciso, portanto, escolher o modelo de comunicação que melhor se adeqüe ao sistema em projeto. Em sistemas que necessitem de comunicação do tipo um-para-muitos, ou mesmo de tempo-real, o modelo de comunicação grupal é mais recomendado. Ao contrário, em sistemas que trabalhem no sistema ponto-a-ponto, isto é, a comunicação se dá entre um transmissor e um receptor, o modelo cliente-servidor é melhor.

3. O PROTOCOLO CAN

O protocolo CAN foi, inicialmente, desenvolvido pela Bosch, em 86, e era usado na interligação de componentes eletrônicos em automóveis. Obtém destaque entre outros protocolos usados em sistemas de automação (Profibus, Interbus-S, P-Net, LON e FIP) por ter uma proposta de alta velocidade (1Mbit/seg), baixo custo e com um conjunto excelente de definições em termos de protocolo. Além disso, diferentemente da comunicação ponto-a-ponto utilizada pela maioria destes protocolos [BRU 00], o CAN se utiliza do modelo grupal de comunicação, através de um sistema de filtragem de mensagens, característica que pode ser amplamente utilizada em qualquer sistema de automação.

3.1 CAN

O CAN é um protocolo digital de comunicação serial que suporta eficientemente os conceitos de controle distribuído de tempo real[1] com elevado grau de segurança [CIA 00].

Uma importante característica do CAN é a de que o protocolo é baseado na técnica CSMA/CR (*Carrier Sense Multiple Access/Collision Resolution*) de detecção e resolução de colisões no acesso ao meio de transmissão. Isto significa que, em caso de colisão, a mensagem de maior prioridade terá acesso ao canal, e a outra terá de esperar, o que evidencia outra característica do protocolo: a priorização de mensagens. Além disso, o CAN possui outras características, como grande flexibilidade de configuração, recepção *multicast*, garantia de consistência dos dados em todo o sistema, e a detecção/ sinalização de erros, além da retransmissão automática de mensagens corrompidas.

Com o objetivo de se obter transparência no projeto e flexibilidade na implementação, o CAN foi dividido em duas diferentes camadas, obedecendo ao modelo OSI/ISO: *Data Link Layer* e a *Physical Layer*. Por sua vez, a camada *Data Link Layer* foi dividida em duas outras subcamadas: *Logical Link Control (LLC) sublayer* e *Medium Access Control (MAC) sublayer*.

A camada física trata de aspectos como a temporização e codificação de bits, além da sincronização dos mesmos. A subcamada de acesso ao meio (MAC) representa o núcleo do protocolo, recebendo mensagens de/para a LLC, sendo responsável pela divisão das mensagens em quadros (*framing*), arbitragem, reconhecimento, detecção e sinalização de erros, sendo gerenciada por uma entidade chamada *Fault Confinement*, que é o mecanismo responsável pela distinção entre falhas temporárias ou permanentes. Já a subcamada lógica (LLC), está relacionada com a filtragem das mensagens, a notificação de *overload* e o gerenciamento da recuperação.

A fig. 3.1 ilustra a arquitetura do protocolo CAN, sua divisão em camadas e as tarefas de cada uma.

---

[1] Como o objetivo deste trabalho não envolve a questão do tempo-real, não haverão maiores explanações sobre o assunto.

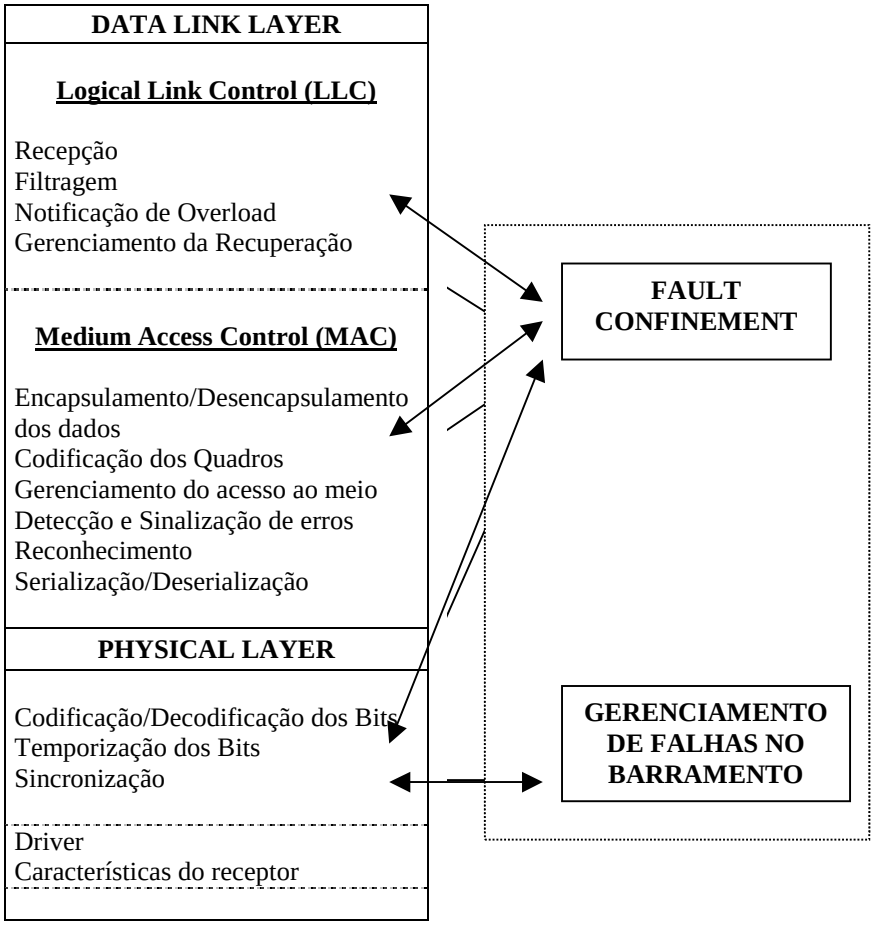

FIGURA 3.1 - Arquitetura do Protocolo CAN

3.2 ASPECTOS IMPORTANTES DO CAN

As informações transitam pelo barramento em mensagens de formato fixo e de tamanho diferente, porém limitado. Essas mensagens possuem um identificador, o qual define a prioridade de uma mensagem no momento do acesso ao barramento e pode identificar, também, o conteúdo da mensagem.

Um nodo não conhece informações sobre a configuração de outros nodos, tais como o endereço da estação, o que traz importantes conseqüências:

- flexibilidade do sistema: novos nodos podem ser adicionados ao sistema sem a necessidade de mudanças de SW/HW dos outros nodos;

- roteamento de mensagens: através do identificador das mensagens, cada nó decide se processa a mensagem ou não;

- *multicast*: por conseqüência da filtragem de mensagens, todas as estações podem processar a mesma mensagem ao mesmo tempo;

- consistência dos dados: o CAN garante a recepção simultânea de uma mensagem ou por todos os nós da rede, ou por nenhum. Isto ocorre porque todos os nodos recebem a mensagem ao mesmo tempo. Se um deles acusar um erro de recepção, a transmissão pára, ou seja, todos os nodos deixam de receber a mensagem corrompida. Caso contrário, se nenhum nodo acusar erro, a transmissão/recepção prossegue até o final.

Cada dispositivo conectado a um barramento CAN, deve transmitir a uma mesma velocidade definida para o barramento, por exemplo 1Mbit. Em um sistema complexo que envolva vários barramentos CAN distintos, pode-se configurar taxas de transmissão igualmente distintas.

Qualquer nó pode acessar o barramento quando este estiver livre. O barramento só se encontra livre quando houverem os bits de fim de mensagem e sincronização do barramento. Se houver conflitos entre mensagens, a de maior prioridade iniciará a transmissão imediatamente, os demais irão esperar até que o barramento esteja livre novamente através da técnica de arbitragem. O conflito é resolvido pela comparação bit-a-bit do identificador das mensagens, ou seja, em cada nó que disputa a transmissão, o bit transmitido ao barramento é comparado ao lá existente, e se for igual a transmissão continua. Quando um nó transmite um bit recessivo (1 lógico), e no barramento está um dominante (0 lógico), este nó aborta a transmissão e espera a liberação do barramento para tentar nova transmissão. Além disso, a arbitragem garante que não serão perdidos nem tempo nem dados. No caso de um quadro remoto e o correspondente quadro de dados serem iniciados ao mesmo tempo (ambos possuem o mesmo identificador, impossibilitando a arbitragem), a prioridade é do quadro de dados.

Para garantir o máximo de segurança possível na transferência de dados, o CAN implementa poderosos métodos de detecção/sinalização de erros e "auto-testes". Com estes métodos, e baseado em um mecanismo denominado *fault confinement* (ver 3.9 *Fault Confinement*), é possível distinguir entre falhas temporárias e erros permanentes. Desta forma, se em um nó é detectado um erro permanente, o mesmo é desligado do sistema.

O número de conexões em uma rede CAN é teoricamente ilimitado. Na prática este número é limitado pelos tempos de atraso e/ou cargas elétricas no barramento, o qual consiste de um único canal bidirecional que carrega bits. O meio físico de transmissão pode ser implementado de diferentes formas. Geralmente isto é feito através do par trançado, mas pode também ser fibra ótica ou rádio freqüência.

3.3 MENSAGENS NO CAN

3.3.1. Formato da Mensagem

Existem dois diferentes formatos de mensagens no CAN, que são diferenciados pelo tamanho do identificador: quadro padrão (*standard frame*), com identificador de 11 bits, e quadro estendido (*extended frame*), com identificador de 29 bits.

3.3.2. Tipos de Quadros (*Frames*)

A transferência de mensagens no CAN é feita através de diferentes tipos de quadros: quadro de dados (*data frame*), quadro remoto (*remote frame*), quadro de erros (*error frame*) e quadro de sobrecarga (*overload frame*).

Os quadros de dados e quadros remotos podem ser usados ambos nos formatos padrão ou estendido, sendo que sua transmissão é separada da transmissão anterior por um outro tipo de mensagem: o espaço interquadros (*interframe space*).

A seguir são descritos mais detalhadamente os tipos de quadros definidos pelo CAN.

- Quadro de Dados (*Data Frame*)

É composto por 7 (sete) diferentes campos de bits: início de quadro, campo de arbitragem, campo de controle, campo de dados, campo de *checksum*, campo de reconhecimento e fim de quadro, sendo que o campo de dados pode ter tamanho zero. A fig. 3.2 apresenta o formato geral do quadro de dados.

CAMPO	Início de Quadro	Campo de Arbitragem	Campo de Controle	Campo de Dados	CRC	ACK	Fim de Quadro
tamanho (bits)	1	12 ou 32	6	0 a 64	15	2	7

FIGURA 3.2 - Formato genérico do quadro de dados

O campo "início de quadro", como o nome já diz, marca o início do quadro, tanto no formato padrão como no estendido. Consiste em um único bit "dominante".

O campo de arbitragem tem composições diferentes de acordo com o formato do quadro. No formato padrão, consiste de um identificador de 11 bits e o bit RTR, que indica se o quadro é remoto ou um quadro de dados. Já no formato estendido, consiste de um identificador de 29 bits, o bit SRR (garante prioridade maior para quadros padrões sobre quadros estendidos, se ambos possuem o mesmo identificador básico de 11 bits), o bit IDE (identifica se o quadro é padrão ou estendido) e o bit RTR. A fig. 3.3 ilustra o campo de arbitragem nos diferentes formatos.

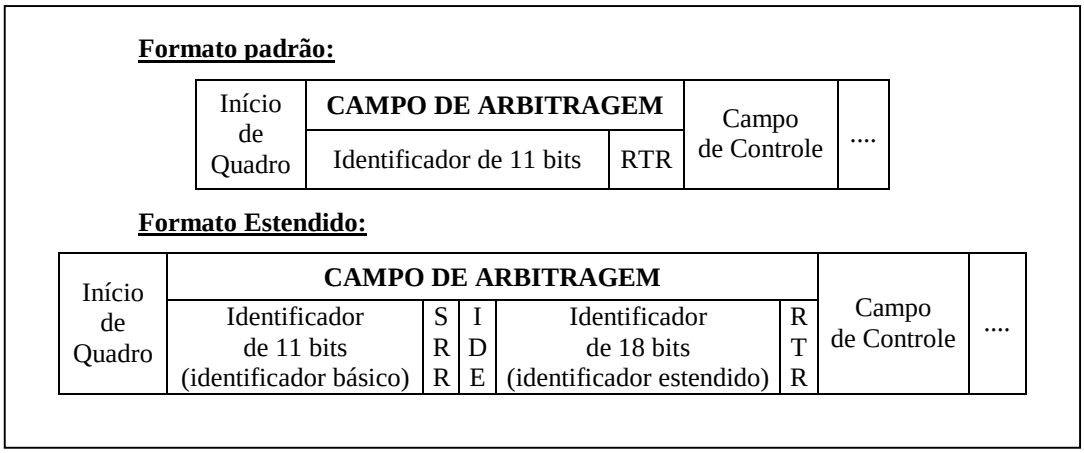

FIGURA 3.3 - Campo de Arbitragem

O campo de controle é formado por 6 bits e sua composição varia conforme o formato do quadro (padrão ou estendido), como mostra a fig. 3.4.

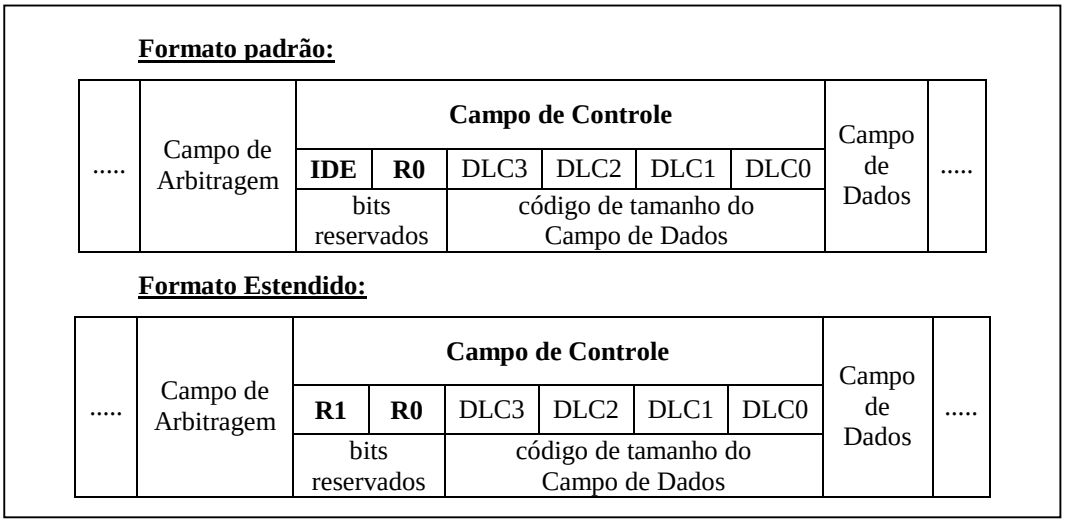

FIGURA 3.4 - Campo de Controle

A diferença está nos bits reservados (IDE/R1, R0). Quadros padrões incluem o bit IDE "dominante" e o bit R0, enquanto quadros estendidos incluem os bits R1 e R0. Os bits reservados devem ser enviados como sendo "dominantes", mas podem ser recebidos como qualquer combinação de dominante/recessivo. A codificação do tamanho do Campo de Dados obedece às indicações na tab. 3.1.

TABELA 3.1 - Codificação de tamanhos para o Campo de Dados

nº de bytes de dados	codificação do tamanho do Campo de Dados* DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d	d	d

* d = dominante; r = recessivo.

O campo de dados pode comportar de 0 (zero) a 8 (oito) bytes, de 8 bits cada.

O campo de *checksum* contém a seqüência CRC seguida por um delimitador para o campo (*CRC delimiter*), o qual consiste em um único bit recessivo.

O campo de reconhecimento (*ACK field*) tem tamanho de 2 (dois) bits. O primeiro é chamado de ACK SLOT e o segundo, de ACK *Delimiter*. O nodo que transmite o quadro seta estes dois bits como recessivos. Os receptores validam a mensagem (dizem ao transmissor que a receberam corretamente) enviando um bit dominante ao barramento quando neste está o bit recessivo referente a ACK SLOT. Como o bit CRC *Delimiter* é um bit recessivo, o bit ACK SLOT será um bit dominante cercado por dois recessivos (CRC *Delimiter* e ACK *Delimiter*).

Finalmente, o campo "fim de quadro", tanto no formato padrão como no formato estendido, constitui-se de uma seqüência delimitadora de 7 (sete) bits recessivos.

- Quadro Remoto (*Remote Frame*)

Atuando como receptor, um nó da rede solicita o envio de um quadro de dados enviando um quadro remoto. O nó que reconhecer o quadro remoto enviado responde, então, enviando o quadro de dados respectivo, com o mesmo identificador do quadro remoto.

O quadro remoto possui a mesma formação do quadro de dados, mas não possui o campo de dados. O bit RTR, nos quadros remotos, é recessivo, enquanto nos quadros de dados, é dominante, o que garante a maior prioridade do quadro de dados, já que ambos possuem o mesmo identificador.

- Quadro de Erro (*Error Frame*)

Este quadro é transmitido por qualquer nó da rede toda vez que este detectar um erro. Consiste de dois campos distintos: o *flag* de erro e o delimitador de quadro. O *flag* de erro é obtido por uma superposição de flags de erro vindos dos diferentes nós da rede. O delimitador nada mais é que uma seqüência de 8 (oito) bits recessivos. O tamanho total do quadro de erro varia de 6 a 12 bits.

Existem dois tipos de *flags* de erro: ativo (seis bits dominantes) e passivo (seis bits recessivos, a menos que outros nós o sobreponham com bits dominantes).

- Quadro de Sobrecarga (*Overload Frame*)

Formado por dois campos (*flag* de sobrecarga – seis bits dominantes; e delimitador de quadro – oito bits recessivos), este quadro é transmitido toda vez que ocorrer um das situações seguintes: necessidade de um atraso maior para o próximo quadro (de dados ou remoto) ou a detecção de um bit dominante nos dois primeiros bits do espaço interquadros.

Quando o quadro de sobrecarga é enviado para se obter um maior atraso, deve ser enviado logo após o quadro de dados (ou quadro remoto), sendo que devem ser enviados no máximo 2 (dois) quadros de sobrecarga para gerar atraso. Quando é enviado devido à detecção do bit dominante, deve ser enviado logo após esta detecção.

- Espaço Interquadros (*Interframe Space*)

Durante a transmissão de dados em um barramento CAN, os quadros enviados são separados entre si por um "espaço interquadros". Desta forma, sempre que um nó da rede quiser transmitir um quadro de dados/remoto, deverá aguardar que o barramento esteja livre, ou seja, o nó irá esperar até que detecte o espaço interquadros no barramento, para então iniciar a transmissão.

Pode-se então dizer que os quadros de dados e remoto são sempre precedidos por um espaço interquadros, o que não ocorre com os quadros de erro e/ou de sobrecarga, os quais não são nem precedidos nem separados por este espaço.

O formato do espaço interquadros depende do estado do nó. Em nós "*error active*", é formado por dois campos: "intervalo", que consiste de 3 bits recessivos, e "barramento livre", o qual não possui tamanho definido, sendo constituído de uma seqüência de bits recessivos que são transmitidos consecutivamente, até que um bit dominante seja detectado no barramento, o que é interpretado como o campo "início de quadro".

Em nós "*error passive*", o espaço interquadros possui ainda um campo adicional, "suspenda transmissão", compreendido entre os dois anteriormente citados, e formado por 8 bits recessivos. Isto significa que este tipo de nó deverá aguardar, além dos 3 bits do campo intervalo, mais 8 bits antes de iniciar uma nova transmissão. Isto ocorre porque nós em estado "*error passive*" podem estar com problemas diversos (ver 3.9 *Fault Confinement*) e, desta forma, precisam aguardar mais tempo que nós "*error active*" para se certificarem de que sua mensagem foi recebida pelos demais componentes conectados ao barramento.

3.4 FILTRAGEM DE MENSAGENS

O CAN oferece um mecanismo de filtragem de mensagens, baseado no identificador da mesma. Assim, os nós conectados a um barramento CAN, desde que

com os respectivos filtros ativos, só processarão as mensagens que passem pelo filtro, ignorando as demais.

Pode-se trabalhar com todo o identificador como máscara para o filtro ou escolher 1 (um) ou mais bits do identificador para ser a máscara e realizar a filtragem baseada nestes bits, permitindo assim que certos nós processem determinados grupos de mensagens.

Por exemplo, configuramos a máscara do filtro para apenas os 2 (dois) primeiros bits do identificador de 11 bits, ou seja, a máscara será 11000000000. Assim, dizemos que os 9 (nove) últimos bits "não importam" para a filtragem (a máscara apenas define quais nós "importam" na filtragem). Depois, configuramos o código do filtro para 10000000000, o que significa que o nó, com esta configuração, vai processar todas as mensagens que tiverem os dois primeiros bits iguais a 10, não importando, para a filtragem, o resto do identificador.

3.5 VALIDAÇÃO DE MENSAGENS

A validação das mensagens se dá em momentos diferentes para transmissores e receptores. O transmissor valida sua transmissão depois do último bit do campo "fim de quadro", enquanto o receptor valida depois do penúltimo bit: o último bit não importa para a validação da mensagem pelo receptor).

3.6 CODIFICAÇÃO

A codificação dos quadros (que nada mais são do que seqüências de bits) se dá da seguinte maneira. Os quadros remoto e de dados têm seus campos "início de quadro", campo de arbitragem, campo de controle, campo de dados e seqüência CRC codificados pelo método de *bit stuffing* (inserção de bits – sempre que detectada uma seqüência de cinco bits idênticos a serem transmitidos, a codificação insere um bit complementar nesta seqüência). Os outros campos (delimitador de CRC, campo de reconhecimento e "fim de quadro") não são codificados por possuírem formato fixo, tal qual os quadros de erro e de overload, que também não são codificados.

3.7 TRATAMENTO DE ERROS (DETECÇÃO E SINALIZAÇÃO)

Existem 5 (cinco) tipos de erro detectados pelo CAN:

- Erro de bit: ao mesmo tempo que um nó transmite um bit, ele monitora o barramento. Se o bit transmitido não for igual ao do barramento, está caracterizado o erro;

- Erro de *stuff*: este erro ocorrerá se forem encontrados 6 (seis) bits iguais consecutivos em um campo de quadro codificado pelo método de *bit stuffing*;

- Erro de CRC: ocorre quando o CRC calculado pelo receptor difere do enviado pelo transmissor;

- Erro de Formato: é caracterizado quando um campo de bits, de formato fixo, possui um ou mais bits "ilegais";

- Erro de Reconhecimento: detectado por um transmissor sempre que não monitorar um bit dominante em ACK SLOT do campo de reconhecimento.

Um nó sinaliza a detecção de um erro transmitindo um *flag* de erro. Se o erro detectado não for de CRC, a transmissão do *flag* de erro inicia no primeiro bit após a detecção, pela próprio nó que o detectou. No caso do erro de CRC, a transmissão do *flag* de erro inicia logo após o bit ACK *Delimiter*, no campo de reconhecimento.

3.8 FAULT CONFINEMENT

Existem dois contadores em cada nó conectado ao barramento, regulando o estado do mesmo: um que conta os erros de transmissão e outro, os erros de recepção. De acordo com os tipos de erros estes contadores são incrementados e decrementados de diferentes formas. Inicialmente, o nó está no estado "*error active*". Quando um de seus contadores supera os 127 pontos, o nó passa ao estado "*error passive*", e se superar 255 pontos, o nó entra no estado "*bus off*". As regras para incremento e decremento destes contadores são um pouco complexas, mas funcionam, basicamente, da seguinte maneira: erros de transmissão valem 8 pontos, e erros de recepção, 1 ponto. Mensagens bem recebidas ou transmitidas decrementam os respectivos contadores.

Um nó *error active* participa normalmente da comunicação no barramento e, ao detectar um erro, envia o flag de erro *error active*, enquanto um nó *error passive* também participa normalmente da comunicação no barramento mas, ao detectar um erro, envia o flag de erro *error passive*. Ainda, após uma transmissão, o nó *error passive* aguarda um certo tempo antes de fazer nova transmissão (como explicado no item 2.4.2). Já um nó em estado *bus off* não participa de nenhuma comunicação do barramento ao qual está conectado.

Portanto, o estado de um nó depende do valor destes contadores, e é desta forma que os controladores CAN distinguem falhas temporárias (eventuais) de erros permanentes. Esse mecanismo de controle de falhas é chamado de *fault confinement*.

3.9 PROTOCOLOS DE ALTO NÍVEL

O padrão do protocolo CAN define somente a camada física (MAC) e a camada de enlace de dados (LLC). Isto significa que o protocolo apenas especifica como transportar pequenos pacotes de dados entre diferentes pontos de um meio comum de comunicação.

Então, é necessário que haja um meio de gerenciar a comunicação em um sistema CAN (controle do fluxo dos dados, envio de mensagens com tamanho superior a 8 bytes, endereços dos nós, estabelecimento das conexões, etc.), o que é feito através dos Protocolos de Alto Nível (*High Layer Protocols*), termo derivado do modelo OSI e seus sete níveis.

Um protocolo de alto nível opera na camada de aplicação do modelo OSI, sendo responsável por diferentes tarefas, tais como:

- Inicialização dos diversos componentes do sistema;
- Distribuição dos identificadores de mensagens entre os diferentes nós de um sistema;
- Tradução (interpretação) do conteúdo dos quadros de dados;
- Gerenciamento do *status* sistema.

Existem diversos desses protocolos implementados, tais como o CanOpen, o DeviceNet, o e o CAN Kingdom, entre outros. Estes protocolos, mesmo sendo desenvolvidos por diferentes grupos/empresas, são abertos, permitindo sua livre utilização para o desenvolvimento de novos produtos.

No protótipo desenvolvido para validar o presente trabalho não foi utilizado nenhum protocolo de alto nível existente. As funcionalidades exigidas para a construção do protótipo, tais como níveis de prioridade, identificação e filtragem das mensagens, além do gerenciamento/monitoramento do sistema como um todo, foram implementadas diretamente na própria linguagem de programação utilizada.

4. HARDWARE ENVOLVIDO

Um sistema de automação típico possui o mais variado tipo de componentes, implementados sobre as mais diversas plataformas. Normalmente uma aplicação possui microcomputadores e microcontroladores(sensores/atuadores), desempenhando funções de monitorização e controle.

O protótipo desenvolvido neste trabalho prevê este tipo de sistema, utilizando um barramento CAN, com um PC em que funciona como supervisor da rede, e um conjunto de sensores microcontrolados. No PC, existe uma placa controladora CAN, que através do barramento, está conectada aos microcontroladores. Neste modelo de comunicação, toda mensagem transmitida pelo barramento é recebida por todos os elementos distribuídos. Cabe ao receptor a tarefa de filtragem destas mensagens.

Estes itens são descritos mais detalhadamente a seguir.

4.1 PLACA CAN PARA PCS

Existem diversos modelos e fabricantes de placas CAN para PCs. Elas podem ser encontradas no padrão ISA, PCI ou cartão PCMCIA, possuindo 1, 2 ou 4 canais CAN.

A placa utilizada para a realização deste trabalho foi a PCCan da Kvaser [KVA 00], com dois canais de comunicação CAN, os quais são formados pelos controladores CAN SJA1000, da Phillips. É possível instalar até 4 placas PCCan em um mesmo PC. A conexão da placa ao barramento é feita através de conectores macho DB9 (idêntico ao conector serial de 9 pinos).

Um *layout* aproximado da placa é mostrado na fig. 4.1.

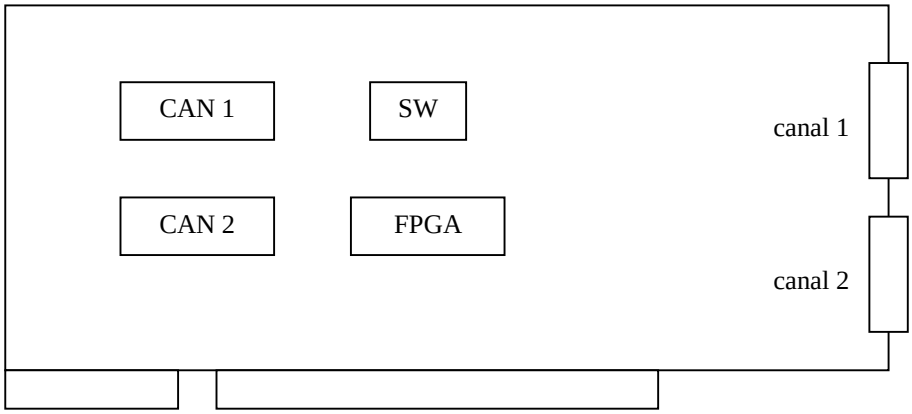

FIGURA 4.1 - *Layout* aproximado da placa PCCan

A placa é acompanhada de uma biblioteca de ligação dinâmica (DLL) com todas as funções necessárias à programação da placa, o que deixa esta tarefa bastante facilitada. Além disso, está disponível por download, na Internet, um componente para ser instalado e utilizado através das linguagens de alto nível da Borland, como o CBuilder ou Delphi.

4.2 MICROCONTROLADORES

4.2.1. Descrição Geral

Os microcontroladores são utilizados nas mais diversas áreas no mundo moderno, em aplicações que vão desde a instrumentação médica e industrial até aparelhos para o grande consumidor e de uso automotivo [BRA 93]. Este capítulo apresenta um pouco sobre os microcontroladores, em geral, mostrando sua arquitetura e seu funcionamento, e traz uma descrição específica do microcontrolador DS80C390, da Dallas Semiconductor, em que se baseia este projeto. Depois são discutidos os recursos e metodologias de programação existentes para os microcontroladores.

Os microcontroladores podem ser vistos como "computadores de um só chip", pois englobam em um só invólucro vários dos componentes existentes nas placas de um computador, tais como uma unidade de processamento, memórias (RAM e ROM), temporizadores, contadores de eventos, canal de comunicação serial, portas de I/O, entre outros. Desta forma, é possível construir sistemas mais compactos mas não menos poderosos aos baseados em microprocessadores.

As principais vantagens dos microcontroladores são o tamanho reduzido e as facilidades do software, já que todos os periféricos são vistos pela UCP como memória. Uma outra vantagem é a de que a gravação do software é feita internamente, na hora da fabricação, impedindo a cópia ilegal do mesmo [SIL 94].

A grande maioria das aplicações comerciais com microcontroladores existente é baseada no 8051, da Philips Components, devido à sua fácil aquisição no comércio especializado, e o seu baixo custo.

Na fig. 4.2 é mostrado um diagrama de blocos de um microcontrolador genérico.

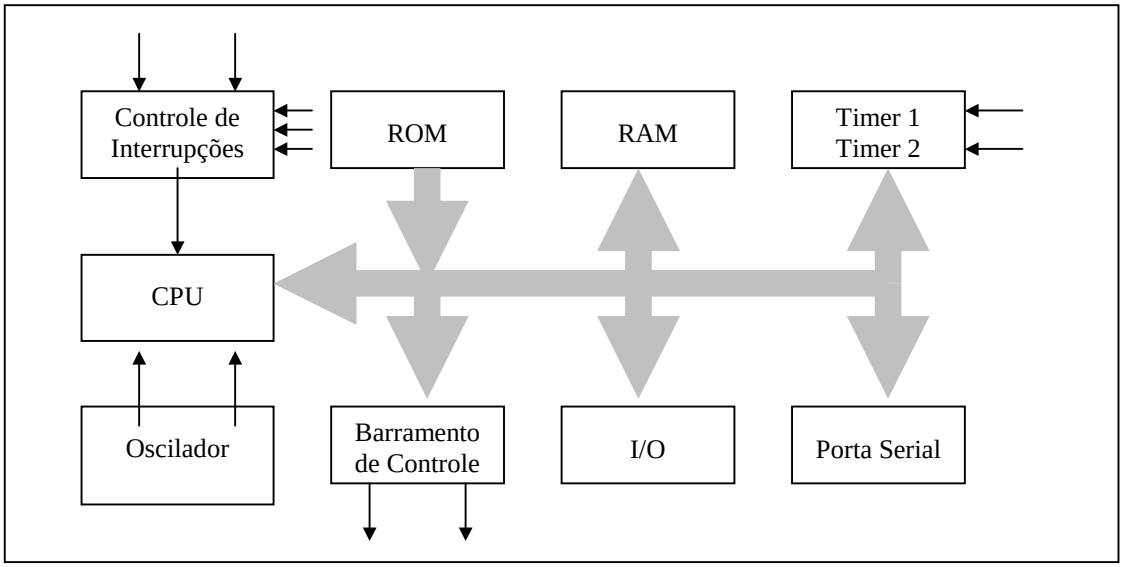

FIGURA 4.2 - Arquitetura genérica de um microcontrolador

Como se pode observar, o microcontrolador incorpora a maioria dos componentes que, em um computador, ficam fora do encapsulamento do microprocessador. No microcontrolador, além de um microprocessador (indicado na fig. 3.1 pelo bloco CPU), existem vários outros componentes:

- ROM (*Read Only Memory* – memória somente de leitura): onde fica armazenado a aplicação propriamente dita, a qual é executada no microcontrolador;

- RAM (*Random Access Memory* – memória volátil): usada para o armazenamento temporário de informações;

- *Timers* (Temporizadores): usados, em geral, para gerar periódicos e precisos pedidos de interrupção, medir largura de pulsos externos, contagem de tempo, entre outras funções;

- Oscilador: responsável pelo fornecimento de energia aos componentes do microcontrolador;

- Barramento de Controle: para controlar dispositivos externos, caso haja;

- I/O: portas de entrada e saída de dados;

- Porta Serial: uma porta para recepção/transmissão serial de dados.

O funcionamento de um microcontrolador é semelhante ao de um PC, com a diferença de que, como já foi dito, os periféricos se encontram todos dentro de um só chip, e são acessados como se fossem posições de memória convencionais, e não através de drivers específicos, como no PC. Nos microcontroladores, o próprio software que roda é, ao mesmo tempo, sistema operacional e aplicativo.

4.2.2. O microcontrolador DS80C390

A escolha deste microcontrolador como base para o presente trabalho se deu devido a sua total compatibilidade com o 8051 e a presença de controladores CAN no mesmo, além do que o mesmo foi fornecido pela empresa para testes.

Trata-se de um microcontrolador de alta velocidade, trabalhando a taxas de até 40Mhz, contra os 12Mhz do 8051. Além disso o seu desempenho foi melhorado em diversos itens, como por exemplo a adição de um coprocessador matemático, resultando em uma performance geral de um microcontrolador de 120 MHz. O sistema ainda possui portas de I/O de 8 bits, três temporizadores de 16 bits, duas portas seriais *full-duplex*. Existem 16 fontes de interrupções, sendo que destas 6 são externas. Além disso, pode-se endereçar até 4Mb de memória externa de dados e 4Mb de memória externa de programa, o que permite o uso de linguagens de alto nível no desenvolvimento de programas/aplicações para o microcontrolador [DAL 00].

O microcontrolador DS80C390 possui dois controladores CAN, totalmente compatíveis com a CAN 2.0B. Cada controlador possui espaço reservado para até 14 tipos de mensagens pré-programadas, isto é, pode-se programar todo o cabeçalho de

diversos tipos de mensagens para, quando necessário, atualizar apenas os campos necessários enviar a mensagem. Além disso, cada controlador pode configurar até 5 tipos diferentes de máscaras de filtragem de mensagens.

4.2.3. Programação de Microcontroladores

A programação de um microcontrolador está sempre muito relacionada ao hardware, isto é, as definições, tais como endereços de memória ou de interrupções variam de um microcontrolador para outro, ou nem isso, de kit para kit (placa onde o microcontrolador está montado). Daí a importância do conceito de programação em camadas.

A programação em camadas se dá da seguinte maneira: tem-se um conjunto de funções responsáveis especificamente pela interação com o hardware, outro conjunto um pouco mais genérico, e assim por diante, até que se chegue ao programa principal. Estes conceitos se assemelham muito à metodologia de programação orientada a objetos, onde as classes de objetos seriam as "camadas" do software.

Pode-se, por exemplo, migrar uma aplicação de um microcontrolador para outro, somente adaptando, ou alterando a classe responsável pelo baixo nível, que é o que realmente se altera de um microcontrolador para outro, ou de kit para kit.

Não existem compiladores de linguagens orientadas a objetos para os microcontroladores de baixo custo, como é o caso do 8051 e de seus sucessores. Existem alguns poucos para outros microcontroladores, mas o alto custo dos mesmos inviabiliza grandes projetos.

A linguagem C vem cada vez mais sendo utilizada para a programação de microcontroladores, o que permite ao programador explorar ao máximo a arquitetura dos microcontroladores sem precisar conhecer muito do *assembly*.

Dito isso, ficam obvias as vantagens da modelagem e implementação dos conceitos da OO neste tipo de aplicação. Mas como aplicar os conceitos de programação em camadas, ou orientada a objetos, se a linguagem C não é OO? O próximo capítulo, apresenta um conjunto de metodologias que permitem o desenvolvimento de programas orientados a objetos em C.

5. ORIENTAÇÃO A OBJETOS EM C

Este capítulo apresenta uma metodologia que, se adotada pelos programadores, possibilita o desenvolvimento de programas que seguem os conceitos de orientação a objetos, na linguagem C, onde tais conceitos não são nativos.

Uma questão básica confronta está metodologia. Se existem tantos compiladores que implementam perfeitamente os conceitos das linguagens orientadas a objetos (LOO), porque então utilizar uma linguagem estruturada? A resposta é simples: não existem compiladores que implementam os conceitos da orientação a objetos para todas as plataformas computacionais que existem. Um exemplo disso são sistemas baseados em microcontroladores como o 8051 e 80196 da Intel. Mesmo não sendo muito comum, existem alguns compiladores de LOO para famílias de microcontroladores específicos, como a família 683xx da Motorola. Entretanto, estes sistemas computacionais são mais caros, o que normalmente inviabiliza a construção da maioria das aplicações em sistemas dedicados.

5.1 OS CONCEITOS DA ORIENTAÇÃO A OBJETOS EM C

Antes de começar a apresentar esta aproximação da orientação a objetos, convém relembrar alguns conceitos básicos referentes ao assunto [YOU 96] e apresentar sua forma aproximada na linguagem C [SIC 97].

Classe: pode-se dizer que uma classe nada mais é do que um modelo para um objeto. Trata-se de uma estrutura definida pelo programador como sendo um tipo de dados, a qual é composta por atributos e métodos. Estes atributos possuem valores, os quais definem o estado atual do objeto e podem ser manipulados somente pelos métodos da classe.

Atributos: são variáveis que representam as características de um objeto. Como dito acima, de seus valores depende o estado de um objeto. Os atributos são internos a um objeto, ou seja, não podem ser acessados diretamente por um programa a não ser que sejam declarados como públicos. A linguagem C não oferece esta forma de proteção e, por isso, os programadores deverão obedecer a uma certa disciplina para que esta "proteção" seja alcançada.

Métodos: os métodos consistem nas funções usadas, geralmente, para manipular os atributos de um objeto, podendo também ser utilizados para efetuar outras operações funcionais de sua classe.

Construtor e Destrutor: tratam-se de dois métodos em especial. O construtor é responsável por alocar um espaço de memória suficiente para guardar uma nova instância da classe a que pertence, e inicializar os respectivos atributos. Já ao destrutor cabe a tarefa de desalocar a memória utilizada por um objeto, devolvendo o controle da área liberada ao sistema operacional.

Encapsulamento: este importante conceito refere-se a guardar, em uma única unidade, todos os membros de uma classe, tanto atributos como métodos. Em LOO, a própria linguagem proíbe o acesso aos atributos privados. Novamente, deverá haver um acordo entre os programadores para que se mantenha o encapsulamento, ou seja, os atributos da classe deverão ser acessados apenas pelos métodos da mesma, já que o C não possui essas funcionalidades de proteção.

Privado, Protegido e Público: em uma LOO, "privado" é uma forma de proteção que faz com que os membros da classe possam ser acessados somente pelos métodos da mesma; "protegido" significa que, além de poder ser acessado pelos métodos da classe, um membro pode ser acessado também pelos métodos de sua(s) superclasse(s); e, por último, definir um membro como "público" significa permitir o acesso ao mesmo de qualquer parte do programa. Por falta, o C define todos os atributos e métodos como públicos: daí a importância de haver uma disciplina rígida entre os programadores para manter o encapsulamento.

Herança: outra importante característica da orientação a objetos é a herança, a qual permite a uma nova classe herdar (copiar, ter acesso a) os atributos e métodos de uma outra classe já existente. Esta nova classe poderá acrescentar alguns outros atributos e/ou métodos, tornando-se mais específica.

Subclasse, Superclasse: são termos utilizados no ambiente da orientação a objetos que significam, respectivamente, uma classe que herda características de outra, e uma classe cujas características são herdadas por outras classes.

Classe Abstrata: uma classe abstrata é uma classe que serve apenas para que outras classes dela herdem atributos e métodos, isto é, trata-se de uma superclasse, apenas. Isto significa que não haverá nenhuma instância desta classe.

Instanciação: pode-se dizer que é o processo pelo qual uma nova instância de uma classe é criada. Como foi dito, a classe é um modelo de objeto, e objeto é a instância de uma classe. Pode-se criar uma instância de uma classe (um objeto) pela execução de seu método construtor.

Objeto: é a instância de uma classe.

Polimorfismo: esta característica da orientação a objetos permite a diferentes subclasses ter um método de mesmo nome, com implementações diferenciadas. Normalmente, isto não é possível, já que é necessário saber, em tempo de ligação, para qual função aponta o método de cada subclasse, o que não ocorre. Utilizando o modelo de POO proposto, é perfeitamente possível implementar esta funcionalidade em C.

Função virtual: chama-se de função virtual uma função que é "ligada" em tempo de execução, e não de ligação.

Como base nestes conceitos foram implementados os exemplos propostos por Ted Van Sickle [SIC 97]. Todos os exemplos foram testados com a linguagem Borland C++ 4.5 e realmente funcionam. Apesar de ser uma linguagem com suporte à OO, a mesma permite a compilação dos programas sem o uso da OO presente na linguagem.

5.2 CLASSES, OBJETOS E ENCAPSULAMENTO

Primeiramente, o arquivo de cabeçalho abaixo (defines.h) define alguns tipos de dados usados no decorrer da explicação.

```
#ifndef DEFINES_H
#define DEFINES_H

#ifndef NULL
    #define NULL (void *) 0
#endif

typedef int Boolean;
typedef unsigned int WORD;

enum {FALSE, TRUE};
#endif
```

FIGURA 5.1 - Arquivo de definições defines.h

Considere-se um simples contador. Este contador pode possuir características que definem o seu estado, como por exemplo, o seu valor atual. Existem, ainda, operações que podem alterar, ou não, o estado do contador. Na terminologia das LOO, pode-se dizer que o contador é um objeto, que o seu valor é um atributo e as operações que agem neste objeto são os métodos deste objeto. O arquivo counter0.h (fig. 5.2) ilustra como é possível definir a classe deste exemplo, ou seja, um contador.

```
#ifndef COUNTER_H       // proteção contra múltiplas inclusões
#define COUNTER_H

#include "defines.h"    // inclusão do arquivo de definições

static struct {         // definição da estrutura que
    WORD count;         // representa a classe Counter
} Counter;
```

FIGURA 5.2 - Arquivo de definições counter0.h

Logo após a inclusão do arquivo de definições, é declarada uma estrutura que define a classe Counter, isto é, a estrutura comporta todos os atributos da classe Counter. Depois são definidos os protótipos dos métodos da classe (*increment* e *query*). Como se observa, neste arquivo existe a declaração da estrutura que representa a classe e dos protótipos dos métodos da mesma, mas nenhum código executável é encontrado. Este é o chamado **arquivo de cabeçalho** da classe. Ao contrário, o **arquivo de implementação** da classe (fig. 5.3) traz os códigos executáveis referentes aos métodos declarados no arquivo de cabeçalho.

```
#include "counter0.h"
```

FIGURA 5.3 - Arquivo de implementação counter0.c

Após a inclusão do arquivo de cabeçalho correspondente, este arquivo traz a implementação das duas funções definidas no arquivo de cabeçalho. A fig. 5.4 apresenta o programa que testa o funcionamento deste objeto.

```
#include "counter0.h"
#include <stdio.h>

void main(){
    int i;
    for(i= 0; i<12; i++)
           increment();
printf("a=%d \n", query());
}
```

FIGURA 5.4 - Programa de testes do contador

Assume-se que counter0.c é compilado em separado e sua versão compilada (o arquivo objeto) é ligado com o programa acima quando este é compilado. Desta forma consegue-se “esconder” a implementação dos métodos da classe do usuário (programador), o qual não precisará saber como os métodos funcionam, e sim, somente como acessá-los, e isto é definido através dos protótipos no arquivo de cabeçalho da classe. O resultado da execução do programa de teste é a=12.

Examinando o programa, vê-se que, de fato, *Count* é um objeto. Entretanto, é possível identificar diversos aspectos da OO não alcançados, como o encapsulamento, por exemplo, já que não há nenhuma forma de proibir o programa principal de acessar diretamente o atributo count, depois do objeto ter sido criado. Outro problema é a não possibilidade da criação de várias instâncias do objeto Contador. Pelo contrário, *Count* é, por si só, a instância de uma estrutura. Ainda, as funções *increment* e *query* trabalham diretamente no membro *count* da estrutura *Counter*, ou seja, mesmo que existissem outras instâncias da estrutura em questão, não seria possível acessá-las.

Utilizando uma metodologia um pouco diferente da apresentada inicialmente, pode-se resolver estes problemas. Os métodos terão como argumento um ponteiro para o objeto que está sendo acessado, permitindo, assim, que os mesmos trabalhem com diversas instâncias da classe *Counter*. Serão acrescentados dois novos métodos para cada classe: o construtor e o destrutor, responsáveis pela alocação e liberação de memória na criação e eliminação das instâncias dos objetos, respectivamente. A fig. 5.5 consiste na declaração desta nova classe.

```
#ifndef COUNTER_H
#define COUNTER_H

#include "defines.h"

typedef struct {
    WORD count;
} Counter;

void increment(Counter *);
int query(Counter *);

Counter *count_(void);        // construtor
void count__(Counter *);      // destrutor

#endif
```

FIGURA 5.5 - Arquivo de cabeçalho Counter1.h

O arquivo de implementação para esta classe é mostrado na fig. 5.6.

```
#include "counter1.h"
#include <stdlib.h>     // para uso da função malloc
#include <mem.h>        // para uso da função memmove

void increment(Counter *this) {
    this->count++;
}

int query(Counter *this) {
    return this->count;
}

Counter *counter_(void) {
    Counter *this;
    if( (this= (Counter *) malloc(sizeof(Counter))) == NULL )
          error_handler();
    this->count= 0;
    return this;
}

void counter__(Counter *this) {
    free(this);
}
```

FIGURA 5.6 - Arquivo de implementação Counter1.c

Os métodos *increment* e *query* trabalham de forma semelhante à anterior, com a única diferença de que agora trabalham com base no objeto que é passado para elas como parâmetro. O método count_ (construtor) não tem argumentos. Primeiramente, aloca a memória necessária para guardar uma nova instância da estrutura tipo Counter (caso não haja memória suficiente, faz o tratamento necessário através da função *error_handler*, que não será tratada aqui de forma mais detalhada). Depois inicializa os atributos da estrutura, neste caso, apenas count e, por fim, retorna o ponteiro para a instância criada. O método destrutor, count__, libera a memória utilizada pela instância de Counter passada como parâmetro, devolvendo a específica área de memória para controle do sistema operacional. O programa a seguir (fig. 5.7) testa o funcionamento desta nova versão do contador.

```
#include "counter1.h"
#include <stdio.h>

void main(){
    Counter *a= counter_();
    Counter *b= counter_();
    int i;

    for(i= 0; i<12; i++) {
           increment(a);
           increment(b);
    }
    for(i= 0; i<6; i++) {
           increment(a);
    }

    printf("a=%d \nb=%d \n", query(a), query(b));
    counter__(a);
    counter__(b);
}
```

FIGURA 5.7 - Teste do contador

Duas instâncias de Counter (*a* e *b*) são criadas pelo método construtor (counter_). Ambos são incrementados 12 vezes e *a* é incrementado novamente mais 6 vezes, resultando no seguinte: a=18; b=12.

Desta forma consegue-se resolver o problema da instanciação, permitindo a existência de tantas instâncias de *Counter* quantas forem necessárias. O único problema desta aproximação é que a mesma necessita de chamadas adicionais, no programa principal, para criação e eliminação dos objetos.

Já que é possível ter múltiplas instâncias de um objeto, é necessário ter argumentos para os métodos dos objetos. O problema, neste ponto, é que não há meios de se determinar qual objeto está enviando mensagens para um método específico. A solução para este problema será mostrada mais adiante.

5.3 HERANÇA, POLIMORFISMO E FUNÇÕES VIRTUAIS

Na forma em que está, a implementação atual não permite herança nem polimorfismo, dois conceitos importantíssimos na POO. Para implementar a herança, é necessário um mecanismo que permita a alteração da estrutura que define uma classe em outra parte do programa. Deve-se permitir que as características de uma estrutura possam ser incorporadas em uma nova estrutura no processo de herança dessa nova estrutura.

Uma vez que uma estrutura é definida em um programa, não é mais possível alterá-la, ao contrário do que se pretende com a herança, onde é possível incluir a definição de uma outra estrutura de classe como parte da estrutura da nova classe. Isto pode levar a utilização de diferentes níveis de endereçamento indireto para o acesso a atributos e métodos, o que deixaria os programas uma verdadeira confusão.

A solução apresentada requer um certo endereçamento indireto, complicando um pouco a programação. Entretanto, os benefícios alcançados compensam o sacrifício. O arquivo de cabeçalho apresentado na fig. 5.8 mostra esta solução.

```
ifndef COUNTER_H
#define COUNTER_H

#include "defines.h"

#define COUNT \
    WORD count; \
    void (*increment)(struct cnt *); \
    WORD (*query)(struct cnt *);

typedef struct cnt{
    COUNT
} Count;

Count *count_(void);
void count__(Count *);

#endif
```

FIGURA 5.8 - Arquivo de cabeçalho Counter2.h

Pode-se ver que a declaração da classe mudou bastante. Primeiramente, declara-se uma constante, COUNT, a qual define tanto os atributos como ponteiros para os métodos da classe. Depois, como declaração da classe *Count*, tem-se a estrutura *cnt*. A principal diferença entre esta implementação e a anterior é a declaração dos atributos e métodos da classe dentro da cláusula *#define*. Isto para que esta mesma definição possa ser aproveitada na declaração de outra classe, como se verá mais adiante. O arquivo de implementação da classe (fig 5.9) mostra o funcionamento desta nova metodologia.

```
#include "counter2.h"
#include <stdlib.h>
#include <mem.h>

static void error_handler(){
}

static void increment(Count *this){
    this->count++;
}

static WORD query(Count *this) {
    return this->count;
}

Count *count_(void) {
    Count *this;

    if( (this= (Count *) malloc(sizeof(Count))) == NULL )
          error_handler();
    this->count= 0;
    this->increment= increment;
    this->query= query;
    return this;
}

void count__(Count *this) {
    free(this);
}
```

FIGURA 5.9 - Arquivo de implementação Counter2.c

Os métodos são declarados estáticos (através da cláusula *static*) para que não haja conflitos de nomes se existir outro método no programa com o mesmo nome. Nesta nova forma, o construtor acrescenta, a cada objeto, ponteiros para os métodos de sua classe (definidos neste mesmo arquivo). Desta forma o acesso de um objeto a um método de sua classe tomará a seguinte forma (supondo que exista um ponteiro para um objeto do tipo count chamado *a*): (*a->increment)(a)

Pode parecer meio estranho ter de referenciar um objeto duas vezes ao chamar um método, mas é necessário. A primeira chamada serve para que se possa acessar o método do objeto específico e a segunda, para identificar o objeto com o qual se está trabalhando. Este *overhead* é uma das desvantagens desta implementação, mas será visto mais adiante que os benefícios trazidos pela mesma são mais fortes. Para testar esta nova implementação pode-se utilizar o programa mostrado na fig. 5.10.

```
#include "counter2.h"
#include <stdio.h>

void main(){
    Count *a= count_();
    Count *b= count_();
    int i;

    for(i= 0; i<12; i++) {
           (*a->increment)(a);
           (*b->increment)(b);
    }
    for(i= 0; i<6; i++) {
           (*a->increment)(a);
    }

    printf("a=%d \nb=%d \n", (*a->query)(a), (*b->query)(b));
    count__(a);
count__(b);
}
```

FIGURA 5.10 - Teste do contador

Novamente, duas instâncias de Counter (*a* e *b*) são criadas pelo método construtor. Ambos os contadores são incrementados 12 vezes e o contador *a* é incrementado mais 6 vezes, resultando no seguinte: a=18; b=12.

Com este novo método é possível implementar a herança. Imagine-se um novo tipo de contador, que possa tanto incrementar quanto decrementar o seu valor (um contador *up*/*down*). Pode-se facilmente avaliar que ele apresenta algumas características semelhantes ao contador já apresentado, possuindo como único atributo um valor, e como um dos métodos, um para incrementar. Então para que rescrever o código de novo? Pode-se aproveitar o código do contador *Count* já existente, da seguinte maneira (fig. 5.11):

```
#ifndef UDCOUNT_H
#define UDCOUNT_H

#include "defines.h"
#include "counter2.h"

#define UDCOUNT \
    COUNT \
    void (*decrement)(struct udcount *);

typedef struct udcount{
    UDCOUNT
} Udcount;

Udcount *udcount_(void);
void udcount__(Udcount *);

#endif
```

FIGURA 5.11 - Implementando a herança

O arquivo de cabeçalho acima (counter3.h) mostra o porque de ter-se declarado anteriormente toda a estrutura da classe *Count* na definição da constante COUNT. Agora, seu conteúdo pode ser reaproveitado, como visto acima, na declaração da constante UDCOUNT, a qual consiste na estrutura da classe do novo contador, ou seja, os atributos e métodos já existentes mais o método *decrement* (decrementar).

Como os métodos *increment* e *query* não necessitam de modificações para uso nesta nova classe, o arquivo de implementação da classe define apenas o método *decrement*, além do construtor e destrutor, como é mostrado na fig. 5.12.

```
#include "counter3.h"
#include <stdlib.h>
#include <mem.h>

static void decrement(Udcount *this){
    this->count--;
}

Udcount *udcount_(void) {
    Udcount *this;
    Count *temp= count_();

    if( (this= (Udcount *) malloc(sizeof(Udcount)))  == NULL )
          error_handler();
    memmove(this, temp, sizeof(Count));
    count__(temp);
    this->decrement= decrement;
    return this;
}

void udcount__(Udcount *this) {
    free(this);
}
```

FIGURA 5.12 - Arquivo de cabeçalho Counter3.h

Como se vê, o método construtor foi bastante modificado. Primeiramente, cria-se uma nova instância (temporária) da classe *Count*, da qual a classe *Udcount* está sendo herdada. Depois, o construtor aloca espaço para guardar uma nova instância da classe *Udcount*, e para este espaço copia a instância de *Count* criada. Em seguida elimina essa instância temporária, e inicializa o ponteiro do novo método *decrement*. Para testar o contador *up*/*down* pode-se utilizar o programa abaixo:

```
#include "counter3.h"
#include <stdio.h>

void main(){
    Udcount *a= udcount_();
    Udcount *b= udcount_();
    int i;

    for(i= 0; i<12; i++) {
           (*a->increment)(a);
           (*b->increment)(b);
    }
    for(i= 0; i<6; i++) {
           (*a->increment)(a);
           (*b->decrement)(b);
    }

    printf("a=%d \nb=%d \n", (*a->query)(a), (*b->query)(b));
    udcount__(a);
    udcount__(b);
return 0;
}
```

FIGURA 5.13 - Teste do contador *up/down*

Este programa retorna como resultado: a=18; b=6.

Portanto, utilizando-se a implementação atual consegue-se implementar, também, a herança em C. Neste exemplo *Udcount* herdou os métodos e atributos de *Count*, e estendeu as operações do objeto acrescentando o método *decrement*.

Pode-se então partir para a discussão do polimorfismo, cuja idéia principal é permitir a diversos métodos que executam funções semelhantes (esta função depende de qual classe foi herdado o método em questão) terem o mesmo nome [SIC 97]. A escolha de qual método usar deve ser feita em tempo de execução. Para tanto, as funções virtuais (métodos de mesmo nome) são acessados não pelo nome e sim, por ponteiros para o método específico a ser usado.

Deseja-se, por exemplo, criar duas classes que herdem as características da classe *Count*: uma que incremente e decremente o valor do contador em uma unidade e a outra, em duas unidades (este exemplo pode não ter nenhuma aplicação prática, mas serve para explicar como funciona o polimorfismo). Pode-se chamar estas classes de *SCounter* (*simple counter*) e *DCounter* (*double counter*), respectivamente. A criação da classe *SCounter* se dará da mesma forma como foi feita a da classe *Udcount*, pois se tratam de classes idênticas. Já para classe *DCounter*, o processo se dará de uma forma um pouco diferente:

Primeiramente, faz-se a criação da classe *DCounter*, herdando as características da classe *Count* da mesma forma que a classe *SCounter*. Depois, no arquivo de implementação da classe, é preciso redefinir o método *increment*, para que o mesmo incremente de duas em duas unidades e, ainda, implementar o novo método *decrement*. Para a criação destas duas classes pode-se utilizar o código mostrado nas figs. 5.14 e 5.15.

```
#ifndef SCOUNTER_H
#define SCOUNTER_H

#include "defines.h"
#include "counter2.h" // arquivo em que se define COUNT

#define SCOUNTER \
    COUNT \
    void (*decrement)(struct scounter *);

typedef struct scounter{
    SCOUNTER
} SCounter;

SCounter *scounter_(void);
void scounter__(SCounter *);

#endif
```

FIGURA 5.14 - Arquivo de cabeçalho SCounter.h

```
#ifndef DCOUNTER_H
#define DCOUNTER_H

#include "defines.h"
#include "counter2.h"

#define DCOUNTER \
    COUNT \
    void (*decrement)(struct dcounter *);

typedef struct dcounter{
    DCOUNTER
} DCounter;

DCounter *dcounter_(void);
void dcounter__(DCounter *);

#endif
```

FIGURA 5.15 - Arquivo de Cabeçalho DCounter2.h

Vê-se que a estrutura básica dos dois arquivos é a mesma. O arquivo de implementação de *SCounter*, como foi dito, ficará idêntico ao arquivo de implementação da classe do contador *up*/*down*, salvo as alterações dos nomes de variáveis, constantes, etc. Pode-se visualizar este arquivo na fig. 5.16.

```
#include "scounter.h"
#include <stdlib.h>
#include <mem.h>

static void decrement(SCounter *this){
    this->count--;
}

SCounter *scounter_(void) {
    SCounter *this;
    Count *temp= count_();

    if( (this= (SCounter *) malloc(sizeof(SCounter))) == NULL )
           error_handler();
    memmove(this, temp, sizeof(Count));
    count__(temp);
    this->decrement= decrement;
    return this;
}

void scounter__(SCounter *this) {
    free(this);
}
```

FIGURA 5.16 - Arquivo de Implementação SCounter.c

Já o arquivo dcounter.c, responsável pela implementação da classe *DCounter*, precisa redefinir o método *increment* herdado de *Count*, da seguinte maneira (fig. 5.17).

```
#include "dcounter.h"
#include <stdlib.h>
#include <mem.h>

static void increment(DCounter *this){     // re-implementação do
    this->count+= 2;                 // método increment.
}

static void decrement(DCounter *this){
    this->count-= 2;
}

DCounter *dcounter_(void) {
    DCounter *this;
    Count *temp= count_();

    if( (this= (DCounter *) malloc(sizeof(DCounter))) == NULL )
           error_handler();
    memmove(this, temp, sizeof(Count));
    count__(temp);
    this->increment= increment;
    this->decrement= decrement;
    return this;
}

void dcounter__(DCounter *this) {
    free(this);
}
```

FIGURA 5.17 - Arquivo de Implementação DCounter.c

Este exemplo pode facilmente ser testado através do programa mostrado na fig. 5.18.

```
#include "scounter.h"
#include "dcounter.h"
#include <stdio.h>

void main(void){
    SCounter *a= scounter_();
    DCounter *b= dcounter_();
    int i;

    for(i= 0; i<12; i++) {
           (*a->increment)(a);
           (*b->increment)(b);
    }

    (*a->decrement)(a);
    (*b->decrement)(b);

    printf("a=%d \nb=%d \n", (*a->query)(a), (*b->query)(b));
    scounter__(a);
    dcounter__(b);
}
```

FIGURA 5.18 - Teste do Contador (com herança e polimorfismo)

Dois contadores são criados: *a* (contador simples) e *b* (contador duplo). Ambos são incrementados doze vezes, pelo método *increment* de cada objeto e, logo em seguida, decrementados uma vez, pelo seu método *decrement*. Como estes métodos possuem diferentes implementações, cada classe responderá de forma diferente ao chamado do método. Assim sendo, o resultado deste programa de teste é: a=11; b=22, provando que a implementação funciona.

Consegue-se, desta forma, implementar o polimorfismo na linguagem C. Classes diferentes podem ter métodos de mesmo nome, e os objetos destas classes reagirão de formas diferentes ao chamado destes métodos, uma vez que cada classe pode possuir uma implementação diferente do método.

5.4 MELHORANDO A INTERFACE DOS OBJETOS

Um problema apresentado pela aproximação atual é a interface dos objetos. Ter de chamar métodos através de ponteiros indiretos torna os programas mais complexos e dificulta a própria compreensão dos mesmos. Pode-se simplificar o uso destes objetos utilizando-se as macros, que nada mais são do que esquemas de substituição de caracteres.

Como exemplo, arquivo couter2.h (fig. 5.8) será alterado da seguinte maneira (as alterações estão em destaque), e será salvo com o nome counter5.h (fig. 5.19).

```
#ifndef COUNTER_H
#define COUNTER_H

#include "defines.h"

#define COUNT \
    WORD count; \
    void (*inc)(struct cnt *); \
    WORD (*que)(struct cnt *);

typedef struct cnt{
    COUNT
} Count;

#define increment(a)    (*(a)->inc)((a))
#define query(a)        (*(a)->que)((a))

Count *count_(void);
void count__(Count *);

#endif
```

FIGURA 5.19 - Arquivo de cabeçalho Counter5.h

O arquivo de implementação *counter5.c* (fig. 5.20) não sofre muitas alterações, apenas os nomes dos métodos são modificados para corresponderem ao arquivo de cabeçalho (counter5.h):

```
#include "counter5.h"
#include <stdlib.h>

static void error_handler() {
}

static void inc(Count *this){
    this->count++;
}

static WORD que(Count *this) {
    return this->count;
}

Count *count_(void) {
    Count *this;

    if( (this= (Count *) malloc(sizeof(Count))) == NULL )
           error_handler();
    this->count= 0;
    this->inc= inc;
    this->que= que;
    return this;
}

void count__(Count *this) {
    free(this);
}
```

FIGURA 5.20 - Arquivo de implementação Counter5.c

Com este "melhoramento" na interface dos objetos, pode-se simplificar bastante o programa principal, obtendo uma interface uniforme entre o mesmo e os métodos dos objetos. Deve-se ter o cuidado de escolher bem os nomes para as funções

definidas pelas macros, já que as mesmas não estão protegidas pela cláusula *static* que antes as tornava locais. Agora, estes nomes valem para todo o programa. O programa mostrado na fig.5.21 testa esta implementação:

```
#include "counter5.h"
#include <stdio.h>

void main(){
    Count *a= count_();
    Count *b= count_();
    int i;

    for(i= 0; i<12; i++) {
           increment(a);
           increment(b);
    }

for(i= 0; i<6; i++) {
           increment(a);
    }

    printf("a=%d \nb=%d \n", query(a), query(b));
    count__(a);
    count__(b);
}
```

FIGURA 5.21 - Teste do Contador (com macros)

resultando em a=18 e b=12.

Existe, ainda, uma outra questão referente à implementação da OO alcançada com as técnicas apresentadas. Na forma em que está, cada objeto deverá ser alocado na memória, reservando espaço para os seus atributos e para os ponteiros de seus métodos. Em programas mais complexos, o espaço necessário para armazenar cada objeto na memória pode aumentar bastante. Pode-se no entanto, amenizar esta situação através de uma "tabela de métodos": cada objeto terá um ponteiro para uma estrutura onde estarão armazenados os ponteiros para os métodos de sua classe. Isto aumentará mais ainda o nível de endereçamento indireto. Entretanto, com o uso das macros, esta maior complicação é amenizada.

Implementando o mesmo contador apresentado acima, porém com o novo conceito da tabela de métodos, tem-se o seguinte arquivo de cabeçalho (fig. 5.22).

```
#ifndef COUNTER_H
#define COUNTER_H

#include "defines.h"

#define METHODS \
    void (*incr)(); \
    WORD (*quer)();

typedef struct {
    METHODS
} Methods;

#define COUNT \
    WORD count; \
    Methods *methods;

typedef struct cnt{
    COUNT
} Count;

#define increment(a)    (*((a)->methods->incr))((a))
#define query(a)        (*((a)->methods->quer))((a))

Count *count_(void);
void count__(Count *);

#endif
```

FIGURA 5.22 - Arquivo de cabeçalho Counter6.h

Existe agora apenas um ponteiro para uma tabela de métodos (**methods*) ao invés de um ponteiro para cada método na estrutura que compõe o objeto, e o acesso aos métodos é feito de forma simplificada através das macros *increment* e *query*. O arquivo de implementação counter6.c (fig. 5.23) traz dois novos itens estáticos: uma instância da tabela de métodos e um *flag*. Quando ocorre a primeira instanciação de uma classe, ou seja, quando o primeiro objeto desta classe é criado, o construtor se encarrega de inicializar a tabela de métodos, e os objetos criados posteriormente terão acesso aos métodos da classe através desta mesma tabela. O *flag* serve justamente para que a tabela seja criada somente na primeira instanciação da classe.

```
#include "counter6.h"
#include <stdlib.h>
#include <mem.h>

static void error_handler(){
}

static void inc(Count *this){
    this->count++;
}

static WORD que(Count *this) {
    return this->count;
}

static Methods meth;
static Boolean count_flag= FALSE;

Count *count_(void) {
    Count *this;

    if( (this= (Count *) malloc(sizeof(Count))) == NULL )
          error_handler();
    this->count= 0;
    this->methods= &meth;
    if(!count_flag) {
          count_flag= TRUE;
          this->methods->incr= inc;
          this->methods->quer= que;
    }
    return this;
}

void count__(Count *this) {
    free(this);
}
```

FIGURA 5.23 - Arquivo de implementação Counter6.c

Para testar esta implementação, pode-se utilizar o mesmo programa que testa a counter5.c (fig. 5.21). Isto ocorre graças a utilização das macros *increment* e *query*. Esta possibilidade demonstra mais uma vantagem da OO: pode-se alterar o comportamento de uma classe, ou de um método de uma classe sem que seja necessário alterar o programa como um todo.

Também é interessante apresentar como se aplica a técnica da tabela de métodos juntamente com a herança e o polimorfismo. Isto pode ser mostrado, novamente, pelo exemplo dos contadores simples e duplo. Quando se faz a herança de uma classe, deve-se decidir onde adicionar os ponteiros para os novos métodos, o que pode ser feito de duas maneiras: ampliando a tabela original ou adicionando nas próprias novas instâncias da classe. Ambas as técnicas têm prós e contras. A primeira é um pouco mais difícil de ser implementada, mas melhora o uso da memória, já que haverá apenas uma tabela de métodos para cada classe herdada. Já a segunda é implementada de forma mais fácil, porém recai sobre o problema do *overhead* introduzido a cada nova instanciação. Por isso, deve-se medir bem estes parâmetros e decidir qual técnica utilizar. No exemplo a seguir utiliza-se apenas a segunda técnica.

Os arquivos de cabeçalho scounter.h e dcounter.h (figs. 5.24 e 5.25) trazem praticamente a mesma estrutura. É importante fazer uma observação no que diz respeito

às macros. Pode-se observar que ambos têm a definição de *decrement(a)*. Isto, na verdade, não deveria ocorrer, pois não se pode redefinir, a não ser que se tenha certeza de que não se vá mais utilizar uma ou outra. Por isso a importância de se escolher bem a técnica para fazer a expansão das classes, já que é necessário que todas as subclasses usem a mesma técnica e os mesmos nomes de métodos para que seja possível utilizar as macros.

```
#ifndef SCOUNTER_H
#define SCOUNTER_H

#include "defines.h"
#include "counter6.h"

#define SCOUNTER \
    COUNT \
    void (*decr)(struct scounter *);

typedef struct scounter{
    SCOUNTER
} SCounter;

#define decrement(a) (*(a)->decr)((a))

SCounter *scounter_(void);
void scounter__(SCounter *);

#endif
```

FIGURA 5.24 - Arquivo de cabeçalho SCounter.h

```
#ifndef DCOUNTER_H
#define DCOUNTER_H

#include "defines.h"
#include "counter6.h"

#define DCOUNTER \
    COUNT \
    void (*decr)(struct dcounter *);

Typedef struct dcounter{
    DCOUNTER
} Dcounter;

#define decrement(a) (*(a)->decr)((a))

DCounter *dcounter_(void);
void dcounter__(DCounter *);

#endif
```

FIGURA 5.25 - Arquivo de cabeçalho DCounter.h

Existe também uma mudança no arquivo de implementação da superclasse (fig. 5.26): a declaração da tabela de métodos deixa de ser estática e torna-se pública, para que outras classes possam utilizá-la na herança.

```
#include "counter6.h"
#include <stdlib.h>
#include <mem.h>

static void error_handler(){
}

//-------------------------------
static void inc(Count *this){
    this->count++;
}

//-------------------------------
static WORD que(Count *this) {
    return this->count;
}

Methods meth;
static Boolean count_flag= FALSE;

//-------------------------------
Count *count_(void) {
    Count *this;

    if( (this= (Count *) malloc(sizeof(Count))) == NULL )
           error_handler();
    this->count= 0;
    this->methods= &meth;
    if(!count_flag) {
           count_flag= TRUE;
           this->methods->incr= inc;
           this->methods->quer= que;
    }
    return this;
}

//-------------------------------
void count__(Count *this) {
    free(this);
}
```

FIGURA 5.26 - Superclasse (Counter6.h)

A seguir são apresentados os arquivos de implementação das duas subclasses (figs. 5.27 e 5.28).

```
#include "scounter.h"
#include <stdlib.h>
#include <mem.h

static void error_handler() {
}

static void decr(SCounter *this){
    this->count--;
}

static Methods s_methods;
static Boolean s_flag= FALSE;
extern Methods meth;

SCounter *scounter_(void) {
    SCounter *this;
    Count *temp= count_();

    if( (this= (SCounter *) malloc(sizeof(SCounter))) == NULL )
          error_handler();

    memmove(this, temp, sizeof(Count));
    memmove(&s_methods,&meth, sizeof(meth));

    count__(temp);

    if(!s_flag) {
          s_flag= TRUE;
          this->methods= &s_methods;
    }

    this->decr= decr;
    return this;
}

void scounter__(SCounter *this) {
    free(this );
}
```

FIGURA 5.27 - Arquivo de implementação SCounter.c

```
#include "dcounter.h"
#include <stdlib.h>
#include <mem.h

static void error_handler() {
}

static void inc(DCounter *this){
    this->count+= 2;
}

static void dec(DCounter *this){
    this->count-= 2;
}

static Methods d_methods;
static Boolean d_flag= FALSE;
extern Methods meth;

DCounter *dcounter_(void) {
    DCounter *this;
    Count *temp= count_();

    if( (this= (DCounter *) malloc(sizeof(DCounter))) == NULL )
           error_handler();

    memmove(this, temp, sizeof(Count));
    memmove(&d_methods, &meth, sizeof(meth));

    count__(temp);

    if(!d_flag) {
           d_flag= TRUE;
           this->methods= &d_methods;
           this->methods->incr= inc;
    }


    this->decr= dec;
    return this;
}

void dcounter__(DCounter *this) {
    free(this);
}
```

FIGURA 5.28 - Arquivo de implementação DCounter.c

As novas alterações estão em destaque. Cada subclasse possui uma tabela de métodos estática, e um ponteiro para a tabela de métodos da superclasse. A tabela da subclasse tem o mesmo tipo da tabela da superclasse. Isto porque está se utilizando a primeira técnica de expansão da tabela de métodos, ou seja, os ponteiros para os novos métodos estarão guardados no próprio objeto. A classe *SCounter* (scounter.c) copia a tabela da superclasse para dentro de sua tabela, e depois inicializa o ponteiro para o método *dec*. Já a classe *Dcounter* (dcounter.c) copia a tabela da superclasse, na em sua primeira instanciação, inicializa, além do ponteiro de *dec*, o ponteiro para o método *inc*, e este permanece guardado na tabela copiada da superclasse.

Novamente pode-se utilizar o mesmo programa que testava a implementação anterior para testar a atual, ou seja, altera-se a implementação das classes, mas a implementação do programa principal não muda.

6. PROTÓTIPO DE APLICAÇÃO DISTRIBUÍDA

Este protótipo de uma aplicação distribuída heterogênea (a partir de agora referenciado somente como protótipo), tem objetivo de verificar o modelo de comunicação proposto, as técnicas de programação OO em linguagem C padrão e as funcionalidades do protocolo CAN. Para isso, foi definida uma aplicação de sistema de automação industrial, baseada em poucos elementos diferenciados, mas com capacidade de transmitir mensagens para elementos individuais, em *broadcast* ou *multicast*.

6.1 ARQUITETURA DO PROTÓTIPO

O protótipo proposto possui três tipos de componentes: sensores de temperatura microcontrolados, um controlador/monitor (chamado a partir de agora de monitor), e um *display*.

Os sensores enviam periodicamente mensagens com o valor da temperatura, as quais são mostradas no *display*. Este mostra as temperaturas lidas por todos os sensores conectados ao barramento. O monitor tem a responsabilidade de verificar cada mensagem enviada ao barramento e enviar comandos gerais de controle de inicialização(Reset) e de programação.

Com este protótipo consegue-se alcançar os objetivos de verificar a comunicação através do uso do protocolo CAN, tais como priorização, identificação e filtragem de mensagens, *broadcast*, *multicast*, entre outras. A fig. 6.1 ilustra a arquitetura deste protótipo.

Os primeiros componentes a serem ligados e conectados ao barramento são o controlador e o *display*. Neste ponto, o sistema já pode ser considerado em funcionamento.

Os nós sensores possuem um temporizador. A cada determinado intervalo de tempo é enviada uma mensagem através do barramento ao *display* com a leitura do sensor de temperatura em um dado momento.

Cada sensor, ao ser ligado, gera um identificador para si mesmo (aleatório) e envia uma mensagem ao barramento solicitando permissão para utilizar o identificador gerado. Se houverem outros sensores já conectados ao barramento, os mesmos comparam o identificador solicitado com o próprio, e se forem iguais, respondem ao recém chegado sensor que o identificador já está em uso. Este, então, gera um novo identificador e tenta confirmá-lo novamente. Quando não receber a mensagem de que o identificador gerado já existe, passa a fazer parte do sistema, e começa a enviar a leitura de temperatura do sensor ao *display*.

O *display* exibe uma lista (dinâmica) dos sensores conectados ao barramento, com as respectivas temperaturas, e as projeta em um gráfico.

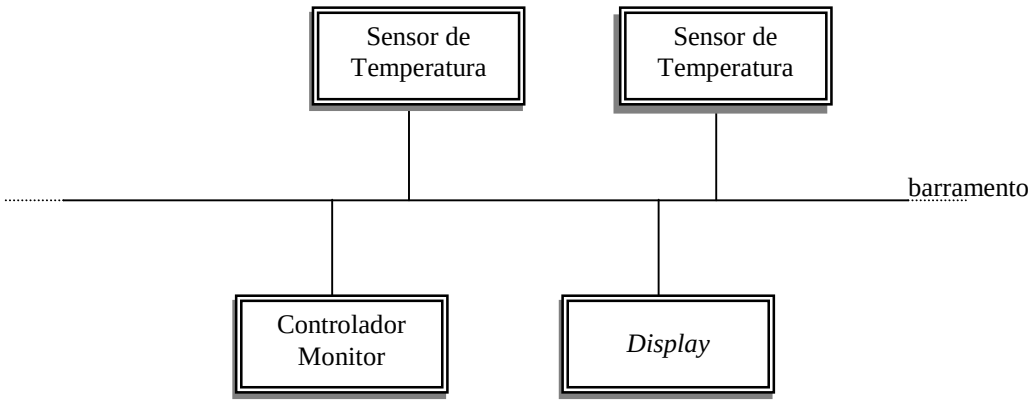

FIGURA 6.1 - Arquitetura do protótipo

O monitor é responsável pela programação do sistema como um todo, e pela monitoração das mensagens que circulam pelo barramento. Através dele é possível reinicializar e programar os diferentes componentes conectados ao barramento, além de visualizar todas as mensagens enviadas ao barramento. O monitor pode enviar mensagens de reinicialização geral (sensores e *display*), específica (sensores ou *display*), mensagens de programação do temporizador (*timer*) dos sensores ou de programação do gráfico exibido no display.

6.2 MENSAGENS

Para este protótipo foi adotado o *standard frame* do CAN, ou seja, 11 bits para a identificação de mensagens. Essa escolha deve-se ao pequeno número de tipos de mensagens utilizados neste protótipo, sendo por isso desnecessária a utilização do formato estendido do protocolo. Estes identificadores foram estruturados como mostra a fig. 6.2, sendo:

- os 3 primeiros bits identificam a prioridade da mensagem (conforme tab. 6.1);
- o 4º e o 5º bit estão reservados e correspondem à indicação de destino da mensagem, ou seja, é baseado nestes dois bits que ocorrerá a filtragem das mensagens. O 4º bit corresponde ao sensor e o 5º, ao display. Quando se deseja enviar mensagens para os sensores o 4º bit será 1 lógico e, quando o destino for o *display*, o 5º bit será 1 lógico. Quando o destino for ambos (como em uma reinicialização geral), ambos terão valor 1 lógico.
  Obs.: O controlador não precisa ser indicado como destino pois o mesmo irá receber todas as mensagens, já que exerce a função de monitor;
- os próximos 4 bits identificam o tipo da mensagem (conforme tab. 6.2);
- os últimos 2 bits estão reservados para eventual uso, não tendo função específica neste protótipo.

Prioridade			Destino		Tipo da mensagem				Reservados	

FIGURA 6.2 - Estrutura do identificador das mensagens

Foram estipulados seis níveis de prioridades de acordo com os tipos das mensagem existentes para o protótipo, conforme a tab. 6.1.

TABELA 6.1 - Prioridade das Mensagens

Prioridade	Tipo da Mensagem	Descrição
001	IDJáExiste	aviso a um componente que o id solicitado já existe
010	IDValor	Transmissão de valores de temperatura ao display
011	Resets	reset de qualquer componente ligado ao barramento
100	Erros	erros diversos
101	RequestID	Requisição de identificador
110	Progs	Qualquer tipo de programação

Os identificadores de todas as possíveis mensagens estão na tab. 6.2, já respeitando os níveis de prioridade, como dito acima.

TABELA 6.2 - Identificadores das Mensagens

Mensagem	Prioridade				S	D	M	M	M	M	x	x
IDJáExiste	0	0	0	1	1	0	0	0	0	0	0	0
IDValor	0	0	1	0	0	1	0	0	0	1	0	0
Reset Global	0	0	1	1	1	1	0	0	1	0	0	0
Reset Display	0	0	1	1	0	1	0	0	1	0	0	0
Reset Sensores	0	0	1	1	1	0	0	0	1	0	0	0
ErroSensor	0	1	0	0	0	1	0	0	1	1	0	0
ErroDisplay	0	1	0	0	0	0	0	1	0	0	0	0
RequestID	0	1	0	1	1	0	0	1	0	1	0	0
ProgSensor	0	1	1	0	1	0	0	1	1	0	0	0
ProgDisplay	0	1	1	0	0	1	0	1	1	0	0	0

6.3 IMPLEMENTAÇÃO

Inicialmente pretendia-se criar um protótipo de aplicação interligando-se a placa CAN (com duas portas de I/O) a dois kits baseados no microcontrolador DS80C390 da Dallas (cada qual com apenas uma porta de I/O). Neste protótipo seriam inicializados um monitor, na saída 1 da placa CAN do PC, um display, na saída 2 da placa CAN do PC e dois sensores, um em cada kit microcontrolador. Devido a imprevistos ocorridos, como a não conclusão da montagem dos kits para os microcontroladores, em tempo hábil, não foi possível a conclusão total da implementação do protótipo. Para possibilitar a validação do modelo proposto, foram feitas simulações diretamente na placa PCCan, e um código aproximado para os microcontroladores foi elaborado.

Neste sentido, não foram necessárias modificações nos programas do monitor e do display (fig. 6.3) mas foi necessário o desenvolvimento um programa que simula o

comportamento de um sensor, ou seja, um programa em Delphi que simula todas as funcionalidades de um sensor microcontrolado conectado ao barramento.

A escolha do Delphi para implementar esta aplicação deve-se ao fato de que todos os elementos foram desenvolvidos para a plataforma PC, e de que a placa PCCan oferece um conjunto de recursos e facilidades para linguagens de alto nível como Delphi e VB. Apesar de não ter sido possível implementar os sensores na plataforma dos microcontroladores, todo o código para o funcionamento destes foram criados (a fig. 6.5 mostra parte deste código), sendo validados através de compilação, em um compilador compatível com o 8051, e simulação, através de um programa Delphi (fig. 6.4).

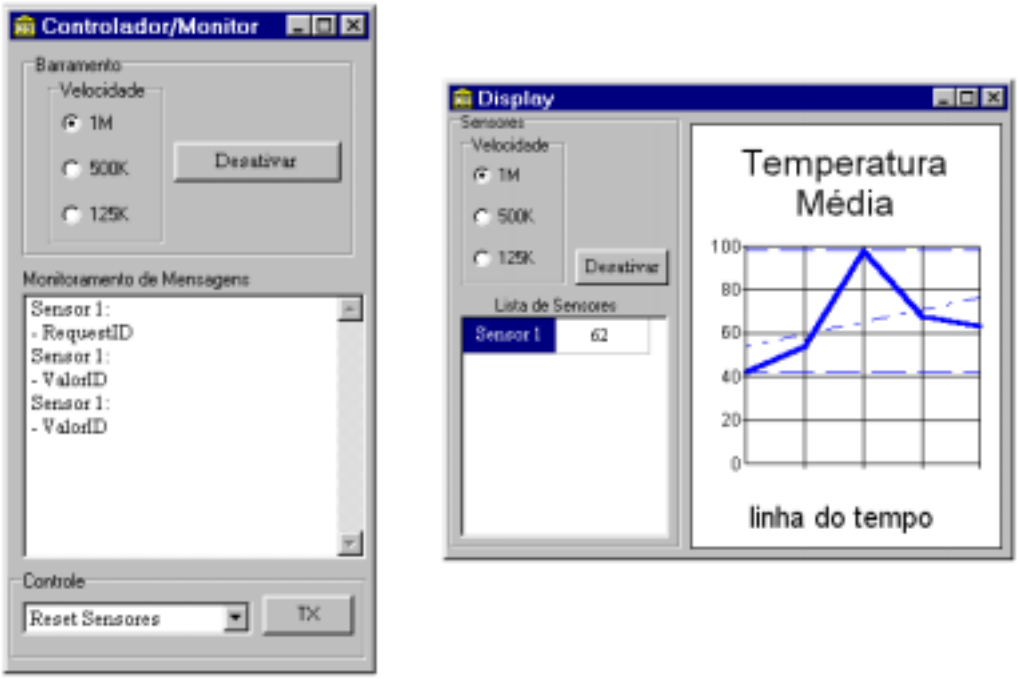

FIGURA 6.3 - Monitor e *Display*

```
procedure TfrmCAN.requisitaId;
var tam: byte;
begin
    randomize;
    ID:= random(255);
    tam:= 1;
    frmCAN.canChannelEx1.Write(idRequestId,ID,tam,ID11);
    frmCAN.canChannelEx1.WriteSync(500);
    frmCAN.Timer.Enabled:= true;
end;
```

FIGURA 6.4 - Código Delphi de solicitação de ID do Sensor

Em C++:

```
void TSensor::ValorID(void) {
BYTE mens[3];
mens[0]= Id;
mens[1]= Valor;
mens[2]= '\0';
(*pCanal).Send(MSG_IDVALOR, mens, ID11);
}
```

Em C:

```
static void ValorID(TSensor *this) {
BYTE mens[3];
randomize();
mens[0]= this->Id;
mens[1]= this->Valor;
mens[2]= '\0';
(*this->pCanal->Send)(this->pCanal,MSG_IDVALOR,mens,ID11);
}
```

FIGURA 6.5 - Código exemplo em C++ e C do envio de mensagens do sensor

6.4 TESTES

Como haviam apenas duas portas de comunicação disponíveis, foram utilizados pares dos elementos da aplicação para realizar a simulação. Foram testadas as comunicações entre Monitor/Sensor, Monitor/Display, Display/Sensor e Sensor/Sensor. Nestas simulações tanto os sistemas de envio de mensagens, como o de filtragem funcionaram perfeitamente.

Para a validação do código C (baseado em C++), que deveria implementar os sensores nos microcontroladores, foi utilizado o compilador Arqmed para 8051 (fig. 6.6). O código compilado não apresentou erros, o que significa ser um código sintaticamente correto também para o DS80C390. A semântica foi validada pelo simulador de sensor, comprovando a operacionalidade da proposta.

```
//-------------------------------------------------------
TCan11 *Bus= TCan11_();          // instancia um Canal CAN
TSensor *S1= TSensor_(Bus);      // instancia um Sensor
TSensor *S2= TSensor_(Bus);
// configura a Velocidade e o Canal
(*Bus->Config)(Bus, TCAN_1MB, TCAN_CANAL1);
// inclui o Sensor na lista do Canal
(*Bus->Lst->AddPointer)(Bus->Lst, S1);
(*Bus->OnOff)(Bus);       // ativa o barramento
// configura o filtro do canal
(*Bus->SetaFiltro)(Bus, MASK, CODE);
(*Bus->OnOffFiltro)(Bus); // ativa o filtro
(*S1->OnOffAD)(S1);       // ativa o conversor AD
// requisita um identificador p/ o sensor
(*S1->RequestID)(S1);
(*S1->ValorID)(S1);       // envia um valor + ID
(*S1->OnOffAD)(S1);       // desativa o conversor AD
(*Bus->OnOffFiltro)(Bus); // desativa o filtro
(*Bus->OnOff)(Bus);       // desativa o canal
TCan11__(Bus);            // elimina a instancia do Canal CAN
TSensor__(S1);            // elimina a instancia do Sensor
//-------------------------------------------------------
```

FIGURA 6.6 - Trecho do código-fonte do protótipo

7. CONCLUSÕES

Aplicações em sistemas de automação industrial exigem, pela a sua complexidade e características especiais, tais como distribuição física dos da aplicação e a heterogeneidade dos sistemas computacionais que dão suporte a estas aplicações, utilização de técnicas de desenvolvimento de software compatíveis com essas necessidades. O uso de técnicas baseadas no modelo de objetos, conforme a bibliografia, consegue retratar de forma simplificada e eficiente conceitos complexos como encapsulamento, herança, concorrência e a forma de troca de mensagens entre os elementos distribuídos.

O problema da utilização dos conceitos da OO em sistemas de automação industrial, reside no fato de que grande parte destes sistemas está implementada sobre plataformas de hardware baseadas em microcontroladores, as quais em sua maioria não possuem compiladores que suportem estes conceitos.

O problema acima citado é, então, contornado através da técnica de POO em linguagem C apresentada neste trabalho. Viu-se que é perfeitamente possível implementar conceitos como objetos, classes, herança, polimorfismo e funções virtuais nesta linguagem, tornando possível a utilização destes conceitos na programação destes microcontroladores. Alguns conceitos, no entanto, não podem ser completamente aplicados, como, por exemplo, esconder os atributos de uma classe do resto do programa. A utilização de macros pode diminuir a complexidade da programação e melhorar o entendimento do programa final, mas deve-se ter cuidado com o uso deste recurso, já que, em funções, ou operações em geral, declaradas através de macros, não há um controle de tipos, por parte do compilador, o que pode gerar erros no programa muito difíceis de serem detectados na hora da depuração.

A programação orientada a objetos em microcontroladores também vem ao encontro da idéia de programação em camadas, mencionada no capítulo 4, bastante útil neste tipo de dispositivo, uma vez que a programação de baixo nível (interação com o hardware) muda de microcontrolador para microcontrolador, e muitas vezes até de kit para kit. Assim, a POO pode facilitar a migração dos sistemas entre as diversas plataformas de hardware existentes.

Infelizmente não foi possível implementar a aplicação da forma desejada, mas apesar disso, e com a realização dos testes e simulações, foi possível contornar este problema e verificar as características do modelo de comunicação proposto, as técnicas de programação OO sob uma linguagem C padrão e das funcionalidades do protocolo CAN.

Com base neste protótipo e no estudo feito pôde-se observar que o protocolo CAN se mostra uma ótima alternativa como solução para a comunicação em sistemas de automação, principalmente naqueles em que existe uma grande quantidade de componentes de mesmo tipo, como por exemplo, o sistema de sensores em silos, onde existem diversos componentes do tipo sensor e um controlador centralizado. Neste tipo de sistema ocorre uma grande demanda pela comunicação do tipo broadcast/multicast, já que é desejável poder-se enviar uma mesma mensagem de forma rápida e precisa a

vários componentes de mesmo tipo. Um outro ponto a favor do uso deste protocolo é o de que existem, no mercado, microcontroladores com *chips* CAN embutidos, de baixo custo e alta performance. Isto é de grande valia uma vez que esses sistemas necessitam, geralmente, de muitos componentes (microcontroladores), o que normalmente encarece os projetos nesta área.

Em extensão a este trabalho, sugere-se como trabalhos futuros, um estudo mais detalhado sobre o outro modelo de comunicação apresentado: o cliente-servidor, e a realização de uma comparação entre os dois modelos aplicados a um mesmo sistema de automação industrial. Também seria interessante a implementação do protótipo de aplicação distribuída proposto neste trabalho diretamente nos microcontroladores DS80C390, como inicialmente pretendido e, a partir de então, desenvolver um sistema de porte maior, que tivesse aplicação realmente prática, e verificar o comportamento do protocolo CAN nesta situação.

BIBLIOGRAFIA

[BOO 91] BOOCH, Grady. **Object Oriented Design.** Redwood City: The Benjamin/Cummings, 1991.

[BRA 93] BRAGA, Newton C. Microcontrolador de 8 bits 80C51. **Saber Eletrônica**, São Paulo, número 247, p. 06-12, Ago. 1993.

[BRU 00] BRUDNA, Cristiano. **Desenvolvimento de Sistemas de Automação Industrial Baseados em Objetos Distribuídos e no Barramento CAN**. CPGEE - UFRGS, Porto Alegre, 2000.

[CIA 00] CAN IN AUTOMATION**. CAN in Automation (CiA): Controller Area Network (CAN)**. Disponível por WWW em http://www.can-cia.de (21/12/2000)

[DAL 00] DALLAS SEMICONDUCTOR CORP**. DS80C390 Dual CAN High-Speed Microprocessor**. Disponível por WWW em http://www.dalsemi.com/news/pr/product/1999/80c390.html (21/12/2000).

[FRI 97] FRIGERI, Alceu H. e PEREIRA, Carlos E. **OORTAC: Um Método de desenvolvimento de Sistemas de Automação em Tempo Real Usando Técnicas de orientação a Objetos**. DELET - UFRGS, Porto Alegre, 1997

[KVA 00] KVASER. **Products - PCCan-Q**. Disponível por WWW em http://www.kvaser.com/products/kvaser/pccan-q.htm (21/12/2000).

[LAW 97] LAWRENZ, Wolfhard. **CAN System Engineering: From Theory to Practical Applications**. New York: Springer-Verlag, 1997.

[RUM 94] RUMBAUGH, James. **Modelagem e projetos baseado em objetos.** Rio de Janeiro: Campus, 1994.

[SIC 97] SICKLE, Ted Van. **Reusable software components - object oriented embedded systems programming in C.** New Jersey: PTR Prentice Hall, 1997.

[SIL 94] SILVA JÚNIOR, Vidal Pereira da. **Microcontrolador 8051**. São Paulo: Érica, 1994. 6ª edição.

[TAN 92] TANNENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Rio de Janeiro: Prentice Hall do Brasil, 1992.

[YOU 96] YOURDON, Edward, COAD, Peter. **Object Oriented Analysis**. Rio de Janeiro: Campus, 1996. 2ª edição.

UNIVERSIDADE FEDERAL DE PELOTAS
INSTITUTO DE FÍSICA E MATEMÁTICA
DEPARTAMENTO DE MATEMÁTICA, ESTATÍSTICA E COMPUTAÇÃO
CURSO DE BACHARELADO EM INFORMÁTICA

O protocolo CAN como solução para aplicações distribuídas, baseadas em objetos, entre PCs e microcontroladores

por

MARCO KASDORF HUBERT

Dissertação apresentada aos Senhores:

_____
Prof. Msc. Marcello da Rocha Macarthy

_____
Prof. Msc. Gil Carlos Medeiros

Vista e permitida a impressão.
Pelotas, _____/_____/_____.

_____
Prof. Cláudio Vianna Villela,
Orientador.