UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
CURSO DE DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Desenvolvimento de Sistemas de Automação Industrial Baseados em Objetos Distribuídos e no Barramento CAN

por

Cristiano Brudna

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Engenharia Elétrica

Prof. Carlos E. Pereira
Orientador

Porto Alegre, setembro de 2000.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Brudna, Cristiano

Desenvolvimento de Sistemas de Automação Industrial Baseados em Objetos Distribuídos e no Barramento CAN / por Cristiano Brudna. — Porto Alegre: CPGEE da UFRGS, 2000.

75 f.: il.

Dissertação (mestrado) — Universidade Federal do Rio Grande do Sul. Curso de Pós-Graduação em Engenharia Elétrica, Porto Alegre, BR–RS, 2000. Orientador: E. Pereira, Carlos.

1.Automação Industrial 2.Orientação a Objetos 3.Barramento CAN 4.Sistemas Embarcados I. E. Pereira, Carlos II. Título.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

Esta dissertação descreve uma arquitetura de suporte para a criação de sistemas de automação baseados em objetos distribuídos e no barramento CAN. Consiste basicamente da utilização de orientação a objetos para modelagem dos sistemas bem como sua implementação na forma de objetos autônomos. Os objetos são então distribuídos em uma rede de placas microcontroladas, as quais são utilizadas para o controle da planta, e PC's, os quais são utilizados para supervisão e monitoração. O suporte em tempo de execução para os objetos é dado por um sistema operacional que permite a sua implementação na forma de processos concorrentes, o qual, no caso das placas microcontroladas, é um sistema operacional do tipo embarcado. A comunicação entre os objetos é realizada através de um protocolo "publisher/subscriber" desenvolvido para o barramento CAN que é suportado por uma biblioteca e elementos de comunicação específicos. Este trabalho tem como objetivo apresentar alternativas aos sistemas de automação existentes atualmente, os quais baseiam-se geralmente em dispositivos mestre/escravo e em comunicações do tipo ponto a ponto. Dessa forma, a arquitetura desenvolvida, apropriada para sistemas embarcados, visa facilitar a criação e dar suporte para sistemas de automação baseados em objetos distribuídos.

TITLE: "DEVELOPMENT OF INDUSTRIAL AUTOMATION SYSTEMS

USING DISTRIBUTED OBJECTS AND THE CAN-BUS"

Abstract

This work describes a computer-based architecture that supports the development of automation systems based on distributed objects and the CAN bus. Based on an object oriented model of automation systems an implementation using autonomous objects, which are implemented as concurrent processes, is created. The objects can be embedded in microcontrolled boards, which are used for controlling tasks, as well as run on PC's, which are used for monitoring tasks and supervision. In both cases an operating system that supports concurrent processes is used. The object communication is suported by a CAN bus using a "publisher/subscriber" protocol provided by a library and specific communication elements. This work intends to provide an alternative to current automation systems which are usually based on master/slave devices and point-to-point communication patterns.

Agradecimentos

Gostaria de agradecer aos meus pais por todo o suporte e incentivo fornecidos durante todo o mestrado e especialmente pela apoio para que fosse possível a minha participação no WRTP'2000 da Espanha. Agradeço também a minha irmã Débora e aos meus amigos Luís e Vander pelo incentivo desde o início do curso de mestrado. Um agradecimento especial ao meu orientador Carlos Eduardo Pereira que em muito me auxiliou durante todo o curso de mestrado assim como no desenvolvimento desta dissertação, não esquecendo o apoio à participação no WRTP'2000 e à obtenção do trabalho no GMD. Agradeço aos meus colegas Carlos Mitidieri, Leandro Becker, Ronaldo Husemann, João Pacheco, Marcelo Gotz, Rafael Wild, Wilson Pardi Junior, Fernando C. de Souza e Eduardo C. da Motta por toda ajuda e colaboração fornecida e também aos demais colegas do Laboratório de Automação e professores do departamento que de uma forma ou de outra contribuíram para este trabalho. Agradeço ainda ao professor Jörg Kaiser da Universidade de Ulm da Alemanha pela colaboração e troca de idéias que contribui para o desenvolvimento de grande parte do trabalho. Por último agradeç a FAPERGS e ao CNPQ pela conceção de bolsas que me permitiram continuar trabalhando até a conclusão desta dissertação.

1 Introdução

1.1 Sistemas de Automação Industrial

A rápida evolução da eletrônica e da microeletrônica tem fornecido meios para o desenvolvimento de sistemas de automação distribuídos em contraste com o antigo modelo centralizado. Componentes com alto desempenho tais como microprocessadores, microcontroladores, memórias e sensores tem sido fabricados a um custo suficientemente baixo para possibilitar a criação de dispositivos autônomos inteligentes. Além disso, o desenvolvimento de sistemas operacionais tempo-real embarcados, técnicas de orientação à objetos (OO) assim como de ferramentas de modelagem e simulação tem também contribuído para a evolução dos sistemas de automação industrial.

A utilização de todos essas ferramentas e métodos de forma integrada torna possível desenvolver sistemas de automação compostos por uma série de sensores, atuadores, controladores e outros dispositivos conectados entre si por uma rede (barramento industrial), os quais cooperam para a realização de tarefas. Isso traz uma série de vantagens quanto à confiabilidade, modularidade, facilidade de compreensão e custo em comparação com os sistemas centralizados anteriormente utilizados.

Diferentes barramentos industriais padronizados tais como Foundation Fieldbus [WIL 99] e Profibus [POP 97] tem sido desenvolvidos nos últimos anos com o objetivo de fornecer um meio de transmissão confiável e de fácil instalação para interconexão de dispositivos em plantas industriais. Contudo, algumas restrições devido ao modelo de comunicação empregado, geralmente mestre-escravo, e limitações na capacidade de tempo-real ainda apresentam-se como questões não completamente resolvidas. Além disso, os dispositivos padronizados que atendem às especificações desses barramentos freqüentemente apresentam altos custos que só são aceitáveis no desenvolvimento de grandes planta industriais.

1.2 Motivações

Devido às limitações das técnicas "tradicionalmente" utilizadas em automação industrial, um novo modelo baseado em tecnologias mais recentes tais como sistemas operacionais embarcados, barramentos industriais, orientação à objetos e microcontroladores de grande capacidade (ver fig. 1.1) pode ser usado para a criação de sistemas mais flexíveis e adaptativos (características consideradas como "inteligentes" na literatura). A modelagem com o auxílio de técnicas de orientação à objetos traz uma série de vantagens, tais como modularidade e encapsulamento, que facilitam não somente a modelagem, mas também a implementação de sistemas distribuídos. Assim, os dispositivos de automação são modelados como objetos autônomos, o que

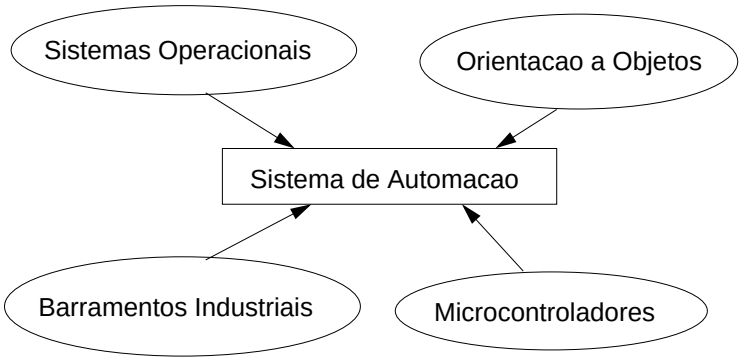

Figura 1.1 — Elementos básicos de um sistema de automação

significa que eles tem autonomia para a tomada de decisões e cooperação sempre que necessário. Além disso, as tarefas são realizadas de forma cooperativa, com a coordenação sendo feita através da troca de mensagens entre os objetos.

O suporte para os objetos pode ser dado por sistemas operacionais embarcados orientados à processos rodando em microcontroladores com capacidade de processamento apropriada. A comunicação entre os objetos, por sua vez, pode ser realizada com a ajuda do sistema operacional, no caso de objetos no mesmo nó, e entre objetos de nós diferentes através do barramento. De acordo com este padrão de comunicação, o barramento CAN (Controller Area Network) [BOS 91][RUF 97], com seu baixo custo, alta confiabilidade, simplicidade e com características de tempo-real apresenta-se como uma alternativa interessante no desenvolvimento de sistemas de automação industrial e de sistemas embarcados. Seu esquema de mensagens com identificação baseada em conteúdo é particularmente adequado ao modelo de objetos autônomos distribuídos sobre uma rede. Além disso, suas características temporais o qualificam para uso em sistemas de automação com características de tempo-real.

A partir deste modelo inicial a questão que se apresenta é como integrar estes elementos básicos de forma que se possa desenvolver, de forma produtiva, sistemas de automação baseados em objetos distribuídos. Esta lacuna pode ser preenchida por uma infra-estrutura de *software* e *hardware* que forneça os meios necessários para se criar e rodar objetos embarcados em placas microcontroladas assim como o suporte para a comunicação entre os objetos.

1.3 Objetivos

Este trabalho tem como objetivo fornecer o suporte mínimo necessário para a criação de sistemas de automação baseados em objetos distribuídos e no barramento CAN. Neste contexto é definido um *hardware* básico e o sistema operacional a ser utilizado pelo sistema de automação. Além disso, são fornecidas as ferramentas para a compilação das aplicações bem como as rotinas de *software* que permitem a comunicação entre os objetos distribuídos através do barramento CAN.

O suporte computacional desenvolvido deve possibilitar, a partir de um modelo orientado a objetos, criar o código necessário para um sistema baseado em objetos distribuídos. Este suporte permite também que o projetista se concentre na fun-

cionalidade do seu sistema, deixando os detalhes de funcionamento da comunicação entre os objetos para a biblioteca e o *software* de suporte desenvolvidos.

1.4 Organização do Texto

Inicialmente, no capítulo 2, é feita uma revisão teórica do barramento CAN e de modelagem orientada a objetos. No capítulo seguinte são discutidos os requisitos básicos de *software* e *hardware* necessários ao desenvolvimento de sistemas baseados em objetos distribuídos e é apresentada a proposta de arquitetura. No capítulo 4 são descritas todas as partes do sistema desenvolvido, seu funcionamento e utilização. O capítulo 5 trata do estudo de caso com uma aplicação de exemplo e finalmente, no capítulo 6, são apresentadas as conclusões.

2 Revisão Teórica

2.1 Barramentos Industriais

Dentre as diferentes possíveis topologias para interconexão de dispositivos de automação (ex: estrela, árvore) [TAN 89], sem dúvida a mais utilizada é a de barramento (ver fig. 2.1). Especialmente quando comparado com o esquema "tradicional", no qual para cada ponto de medição é necessário uma conexão independente (vide fig. 2.2), a conexão usando barramento traz uma série de vantagens. Dentre eles pode-se citar:

- Flexibilidade para estender a rede e adicionar módulos na mesma linha.
- Permite atingir maiores distâncias do que com conexões tradicionais.
- Redução substancial de cabeamento.
- Redução dos custos globais.
- Simplificação da instalação e operação.
- Disponibilidade de ferramentas para instalação e diagnóstico.
- Possibilidade de conectar dispositivos de diferentes fornecedores.

Contudo, a substituição de um sistema existente por um barramento industrial possui algumas desvantagens aparentes: necessidade de adquirir *know-how* e alto investimento inicial. Além disso, a interoperabilidade nem sempre é garantida.

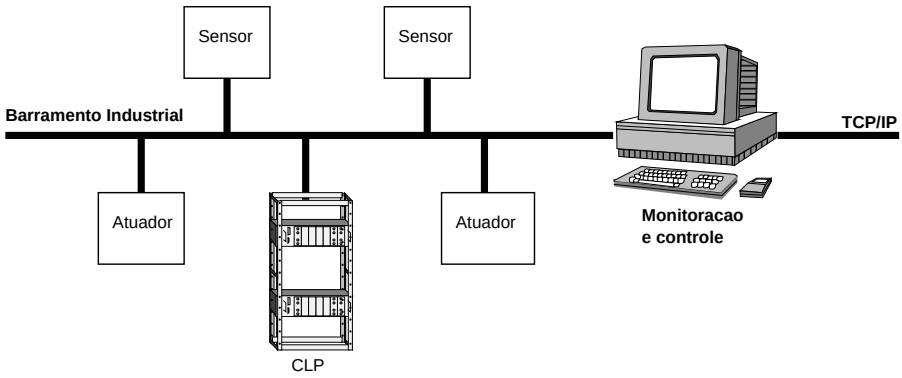

Figura 2.1 — Exemplo de sistema de automação com barramento

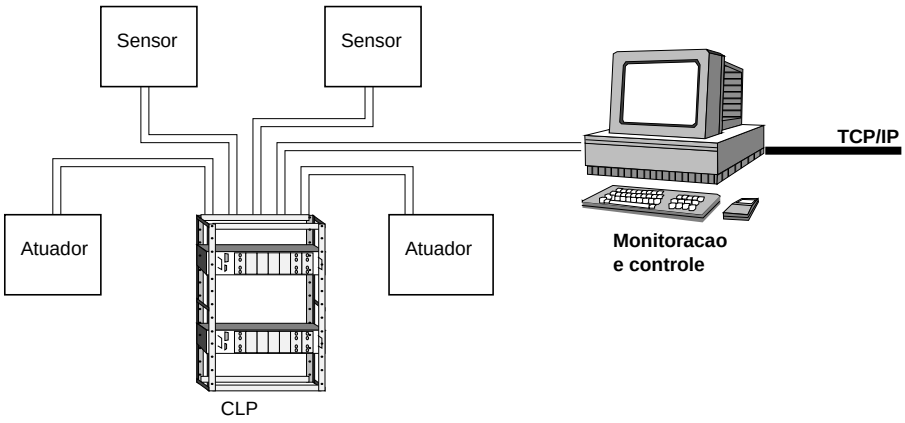

Figura 2.2 — Exemplo de sistema de automação sem barramento

2.2 Escolha do Barramento

Dentre os protocolos existentes para comunicação industrial, tais como o Foundation Fieldbus [WIL 99] e o Profibus [POP 97], o protocolo CAN apresenta-se como uma alternativa bastante adequada para o desenvolvimento de sistemas de automação industrial. A utilização de objetos distribuídos em conjunto com os outros barramentos industriais traz algumas dificuldades relacionadas com o modelo de controle geralmente utilizado por eles. O barramento Fundation Fieldbus, por exemplo, apesar de permitir o uso de modelos cliente/servidor e "publisher/subscriber", utiliza-se do conceito de blocos funcionais padronizados que dificulta a utilização de orientação a objetos no mesmo. Já com relação ao Profibus, tem-se que a utilização de mestres e escravos com comunicação ponto a ponto restringe o uso de objetos distribuídos, já que a autonomia dos mesmos é uma das características que se deseja preservar.

Diferentemente da comunicação ponto a ponto geralmente utilizada pelos outros barramentos, o barramento CAN possui a característica de *broadcast* que pode ser convenientemente explorada pelos sistemas de automação industrial. Além disso, a priorização de mensagens utilizada no barramento CAN oferece algumas propriedades de tempo-real que podem ser convenientemente aproveitadas. O barramento CAN oferece ainda a possibilidade de se definir um protocolo de alto nível para as aplicações de acordo com as necessidades do usuário, já que o seu padrão básico cobre somente as duas primeiras camadas do padão ISO/OSI.

Assim, quanto à utilização de objetos distribuídos, tem-se que o barramento CAN apresenta-se como uma alternativa bastante adequada, já que este oferece bastante liberdade ao projetista para a utilização de qualquer modelo de controle e padrão de comunicação desejado. Por último, seu custo é suficientemente baixo para ser utilizado em placas microcontroladas, o que favorece o desenvolvimento de sitemas de automação com controle distribuído.

2.3 Barramento CAN (Controller Area Network)

2.3.1 Introdução

O barramento CAN é um protocolo de comunicação serial desenvolvido inicialmente pela Bosch em 1986 para aplicações automotivas que recentemente vem sendo utilizado em sistemas de automação industrial. O protocolo CAN é baseado na técnica de CSMA/CR (*Carrier Sense Multiple Access/Collision Resolution*), às vezes também chamado de CSMA/CD + AMP (*Carrier Sense Multiple Access/Collision Detection and Arbitration on Message Priority*), de acesso ao meio de transmissão. Isto significa que sempre que ocorrer uma colisão entre duas ou mais mensagens, a de mais alta prioridade terá o acesso ao meio físico assegurado e prosseguirá a transmissão.

As características básicas do barramento CAN são as seguintes: 8 bytes de dados, velocidade de até 1Mbit/s, priorização de mensagens, recepção *multicast* com sincronização, detecção de erros e sinalização e retransmissão automática de mensagens corrompidas. Todas estas características propiciam simplicidade, alta confiabilidade e segurança, além de baixo custo. A primeira versão do protocolo, CAN 2.0A [BOS 91], especificava somente mensagens do tipo padrão com identificadores de 11 bits enquanto que o CAN 2.0B [BOS 91] admite também mensagens estendidas com identificadores de 29 bits. O protocolo CAN foi adotado em 1993/94 como padrão mundial ISO11898 pela International Standardization Organization.

O protocolo é usualmente implementado em um controlador na forma de um circuito integrado, mas também pode-se encontrar no mercado microcontroladores de 8, 16 e 32 bits com controladores CAN integrados. Os controladores CAN e os microcontroladores com controladores CAN integrados são fabricados por um grande número de indústrias tais como Intel, Motorola, Philips, Siemens e Texas Instruments, resultando em torno de 50 circuitos integrados de 15 fabricantes diferentes. A utilização do protocolo CAN na industria automobilística resultou numa produção em grande escala de controladores CAN, atualmente na casa dos milhões ao ano.

A afirmação e crescimento do protocolo CAN está fortemente baseado na organização dos fabricantes em torno de associações tais como a CiA (CAN in Automation) [1] assim como na existência de uma série de ferramentas de *software* para o desenvolvimento, simulação, configuração e monitoração de aplicações bem como na disponibilidade de *hardware* na forma de placas de controle, placas ISA, PCI e outros.

2.3.2 Características Gerais

De acordo com o modelo OSI/ISO o padrão CAN 2.0B (tabela 2.1) é constituído somente de duas camadas: *Data Link Layer* e *Physical Layer*. As características do barramento CAN podem ser resumidas por:

- É baseado no conceito de *broadcast*.

[1] www.can-cia.de

Camada	Função	Especificação
OSI-Layer 7	Application	especificado pelo projetista
OSI-Layer 6	Presentation	vazio
OSI-Layer 5	Session	vazio
OSI-Layer 4	Transport	vazio
OSI-Layer 3	Network	vazio
OSI-Layer 2	Data Link	coberto pelo CAN e padrão ISO
OSI-Layer 1	Physical	coberto pela ISO e parcialmente pelo CAN

Tabela 2.1 — Modelo OSI/ISO do protocolo CAN

- Um esquema de arbitragem não destrutiva (*bitwise arbitration*) descentralizada, baseada na adoção dos níveis "dominante" e "recessivo", é usada para controlar o acesso ao barramento.
- As mensagens de dados são pequenas (no máximo oito bytes de dados) e são conferidas por *checksum*.
- Não há endereço explícito nas mensagens. Em vez disso, cada mensagem carrega um identificador que controla sua prioridade no barramento e que pode servir como uma identificação do conteúdo da mesma.
- Utiliza um elaborado esquema de tratamento de erros que resulta na retransmissão das mensagens que não são apropriadamente recebidas.
- Fornece meios efetivos para isolar falhas e remover nós com problemas do barramento.
- Oferece meios para filtragem das mensagens.
- O meio físico de transmissão pode ser escolhido de acordo com as necessidades. O mais comum é o par trançado, mas também podem ser utilizados outros meios de transmissão tais como fibra ótica e rádio freqüência.

Uma das propriedades mais importantes do barramento é o esquema de arbitração binária (*bitwise arbitration)* que fornece uma boa maneira de resolver colisão de mensagens. Sempre que dois nós começarem a transmitir mensagens ao mesmo tempo, o mecanismo de arbitragem garante que a mensagem de mais alta prioridade será enviada. Isto é conseguido através da definição de dois níveis de barramento chamados "recessivo" e "dominante". Um nível "dominante" sempre sobrescreve um nível "recessivo". Assim, enquanto um nó está enviando uma mensagem ele compara o nível do bit transmitido com o nível monitorado no barramento. Se um nó tenta enviar um nível "recessivo" e detecta um "dominante" ele perde a arbitragem e interrompe o processo de transmissão.

A taxa de transmissão, por sua vez, é limitada pelo comprimento do barramento utilizado. No caso do uso de par trançado, por exemplo, temos uma determinada relação entre a taxa de transmissão e o comprimento máximo do barramento, como

Taxa (Kbit/s)	Distancia máx. (m)
1000	40
500	130
250	270
125	530
100	620
50	1300
20	3300
10	6700
5	10000

Tabela 2.2 — Taxa de transmissão x distância para barramento CAN

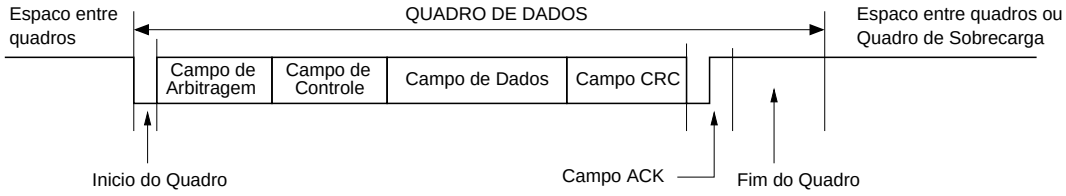

Figura 2.3 — Quadro de Dados

mostra a tabela 2.2. Essa limitação decorre da necessidade de sincronização entre as estações componentes do barramento a qual da suporte ao esquema de arbitração binária.

2.3.3 Quadros de Uma Mensagem CAN

O barramento CAN na sua versão 2.0B utiliza-se de quatro tipos de quadros (*frames*) para controlar a transferência de mensagens:

- Quadro de Dados (*Data Frame*)
- Quadro Remoto (*Remote Frame*)
- Quadro de Erro (*Error Frame*)
- Quadro de Sobrecarga (*Overload Frame*)

2.3.3.1 Quadro de Dados

O Quadro de Dados leva dados do transmissor para os receptores e é composto de sete campos diferentes mostrados na figura 2.3:

- **Início do quadro (SOF)**: marca o início do quadro com um bit "dominante" simples.
- **Campo de arbitragem**: no formato padrão (fig. 2.4) consiste de um identificador de 11 bits e do bit RTR, o qual indica se é um Quadro de Dados

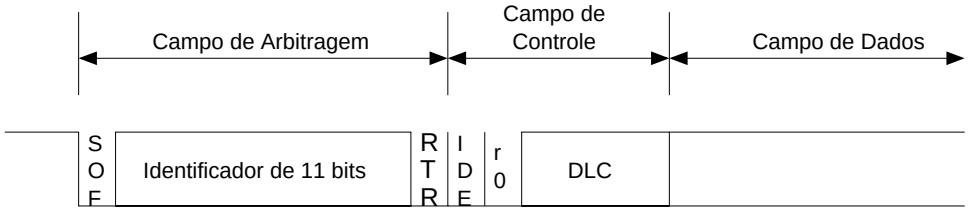

Figura 2.4 — Quadro de Dados Padrão

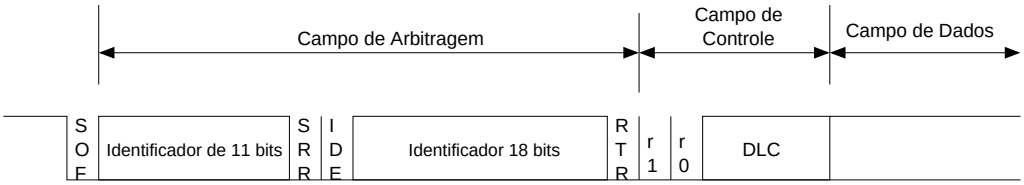

Figura 2.5 — Quadro de Dados Estendido

(RTR = "dominante") ou Quadro Remoto (RTR = "recessivo"). No formato estendido (fig. 2.5) é formado por um identificador de 29 bits, do bit SRR ("recessivo", substitui o bit RTR do formato padrão), bit IDE (identifica se o quadro é padrão, IDE = "dominante", ou estendido, IDE = "recessivo") e pelo bit RTR.

- **Campo de controle**: no formato padrão é constituído do DLC (*Data Length Code*), que indica o número de bytes no campo de dados, e dos bits IDE e r0 (reservado). No formato estendido é formado pelo DLC e pelos bits reservados r1 e r0.
- **Campo de dados**: consiste dos dados a serem transmitidos pelo Quadro de Dados.
- **Campo CRC**: contém a seqüência de CRC (*Cyclic Redundancy Code*) de 15 bits seguida por um delimitador (1 bit).
- **Campo ACK**: é composto pelo ACK *Slot* (1 bit) e Delimitador de ACK (1 bit). É utilizado pelos nós receptores para indicar o correto recebimento de uma mensagem.
- **Fim do quadro**: é formado por uma seqüência de sete bits "recessivos" e serve para delimitar o quadro.

2.3.3.2 Quadro Remoto

O Quadro Remoto, mostrado na figura 2.6, é utilizado para requisitar dados. É formado pelos campos: Início do Quadro, Campo de Arbitragem, Campo de Controle, Campo de CRC, Campo de ACK e Fim do Quadro. No Quadro Remoto o bit RTR é "recessivo" e não há campo de dados. Este tipo de quadro é utilizado quando um nó deseja solicitar dados a outro nó. O nó que possui a informação responde, então, através de um Quadro de Dados com o mesmo identificador da solicitação.

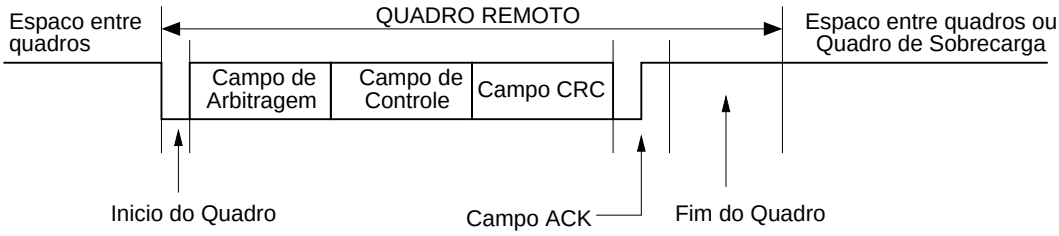

Figura 2.6 — Quadro Remoto

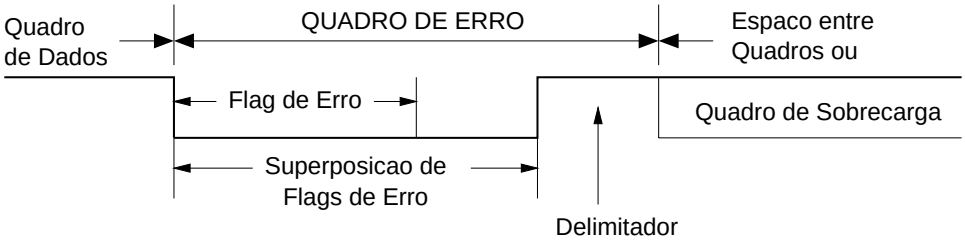

Figura 2.7 — Quadro de Erro

2.3.3.3 Quadro de Erro

Um Quadro de Erro é transmitido por um nó sempre que for detectado algum tipo de erro no barramento. Se um nó detectar um problema com um Quadro de Dados, por exemplo, isto será imediatamente indicado por um Quadro de Erro. O Quadro de Erro, como mostra a figura 2.7, é composto por dois campos:

- **Flags de Erro**: é formado pela superposição de Flags de Erro provenientes de diferentes nós. O comprimento total pode variar de seis a doze bits. Existem dois tipos de Flags de Erro: Flags de Erro Ativos, que consistem de seis bits "dominantes" consecutivos, e Flags de Erro Passivos, constituídos por seis bits "recessivos" consecutivos.
- **Delimitador de Erro**: consiste de oito bits "recessivos".

2.3.3.4 Quadro de Sobrecarga

Existem duas condições que levam à transmissão de um Quadro de Sobrecarga:

- As condições internas de um receptor, o que requer um atraso até o próximo Quadro de Dados ou Quadro Remoto.

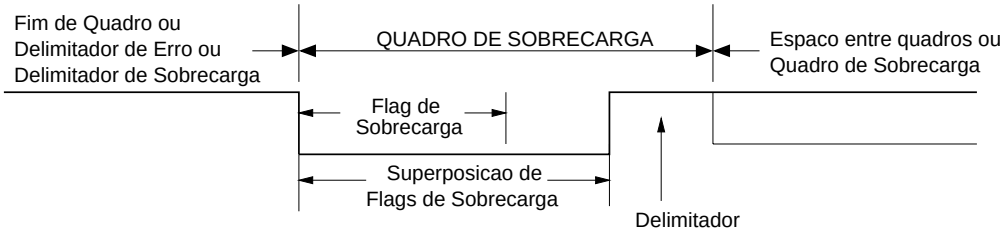

Figura 2.8 — Quadro de Sobrecarga

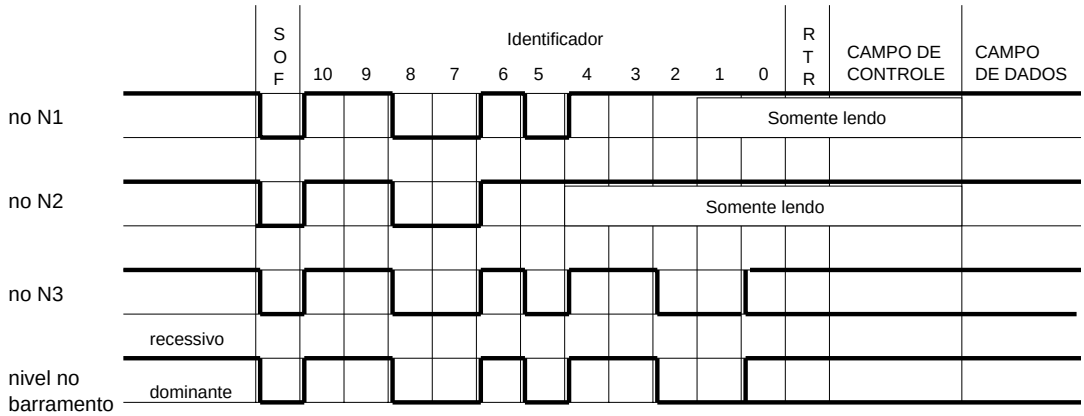

Figura 2.9 — Exemplo do processo arbitragem no barramento CAN

- Detecção de um bit "dominante" no espaço entre dois quadros.

No máximo dois Quadros de Sobrecarga podem ser gerados para atrasar o próximo Quadro de Dados ou Quadro Remoto. O Quadro de Sobrecarga, mostrado na figura 2.8, é formado pelos campos:

- **Flag de Sobrecarga**: consiste de seis bits "dominantes".
- **Delimitador de Sobrecarga**: consiste de oito bits "recessivos".

2.3.4 Arbitração do Barramento

O processo de arbitragem ocorre sempre que dois ou mais nós iniciam uma transmissão exatamente ao mesmo tempo, após verificar que o barramento está desocupado. Um exemplo deste processo, onde se tem três estações N1, N2 e N3 concectadas a um barramento, pode ser visto na figura 2.9. Neste exemplo vê-se que nada acontece até o bit cinco, onde os nós N1 e N3 transmitem bits "dominantes" ("0") e o nó N2 tenta transmitir um bit "recessivo". Por ter escrito um bit "recessivo" e lido um bit "dominante" no barramento o nó N2 perde a arbitragem e passa somente a ler o barramento, emitindo bits "recessivos". Finalmente no bit dois o nó N1 perde a arbitragem para o nó N3 que possui um identificador com valor menor e, por isso, com maior prioridade do que os nós N1 e N2.

O resultado deste processo de arbitragem é que a mensagem com a prioridade mais alta sempre consegue ser transmitida sem atrasos. Os nós N1 e N2, cujas mensagens perderam o processo de arbitragem, terão suas mensagens transmitidas depois que o nó três finalizar a sua transmissão.

2.3.5 Detecção e Sinalização de Erros

São utilizados dois mecanismos de detecção de erros ao nível de bits: monitoração e inserção de bits (*bit stuffing*). Já no nível de mensagens são usados o *Cyclic Redundancy Check* (CRC) e checagem dos quadros (*Frame check*). A sinalização de qualquer erro detectado é feita através de um Quadro de Erro.

A monitoração simultânea, por todos os nós, das mensagens que circulam no barramento permite a detecção de mensagens corrompidas. Desta maneira os

campos das mensagens são constantemente verificados de acordo com o tipo da mensagem, assim como o seu CRC. Caso algum nó detecte problemas com uma mensagem ele indicará isso através de um Quadro de Erro. Tão logo seja possível a mensagem será retransmitida garantindo assim que todas as mensagens serão corretamente recebidas por todos os nós.

Há também mecanismos de proteção contra nós com algum tipo de falha. Dessa maneira, se um dos nós estiver constantemente indicando o recebimento de mensagens corrompidas ele estará perturbando o barramento com sucessivas retransmissões de mensagens. Assim, um contador interno de erros existente em cada controlador CAN força o nó a se desconectar quando um certo número de erros é atingido.

2.3.6 Filtragem

Como dito anteriormente, o protocolo CAN é usualmente implementado na forma de um controlador que usualmente também se comunica com um microcontrolador. Assim, a maioria dos controladores de protocolo oferece um serviço de filtragem de mensagens que faz com que somente as mensagens com o padrão de identificação pré-programado sejam armazenadas e sinalizadas ao microcontrolador. Isso possibilita uma economia de tempo de leitura e processamento das mensagens recebidas, liberando o microprocessador para tarefas mais importantes, contanto que a filtragem seja corretamente configurada. Essa operação normalmente envolve a configuração de duas máscaras para o identificador das mensagens de forma a selecionar as mensagens ou grupo de mensagens desejadas e descartar as não desejadas.

2.3.7 Interfaceamento

2.3.7.1 Interface com microcontrolador

Um nó é geralmente conectado a um barramento constituído de dois fios terminados em ambas as pontas por dois resistores. O controlador CAN pode estar diretamente conectado ao microcontrolador ou como dito anteriormente, o microcontrolador pode possuir um controlador CAN internamente. A ligação do nó com o meio físico normalmente se dá através de um *transceiver,* o qual se comunica serialmente com o controlador através do pino de saída Tx e do pino de entrada Rx. A função do *transceiver*, no caso do uso de par trançado, é transformar os bits que entram e saem do controlador em tensão diferencial a ser aplicada ao barramento.

O modo de transmissão diferencial é utilizado para proporcionar imunidade a interferências eletromagnéticas ao barramento. A alimentação do *transceiver* é feita com cinco volts e o nível "recessivo" corresponde à uma diferença de tensão menor do que 0,5V entre CAN_H e CAN_L, com a tensão em CAN_H sendo normalmente maior do que CAN_L. Já o nível "dominante" é detectado quando a diferença de tensão for de no mínimo 0,9V sendo que a tensão nominal neste estado é de 3,5V para CAN_H e 1,5V para CAN_L. Convém lembrar que os sinais dos pinos Tx e Rx possuem direções definidas enquanto que os sinais CAN_H e CAN_L não. O esquema deste tipo de conexão é mostrado na figura 2.10.

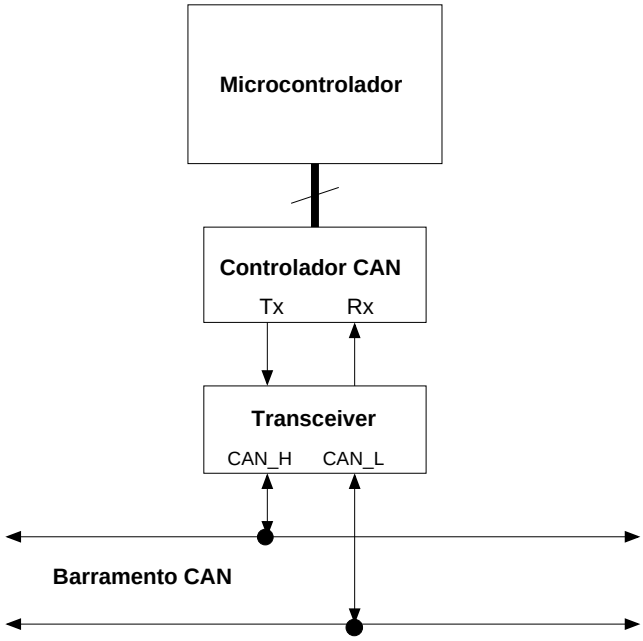

Figura 2.10 — Esquema para a interface do microcontrolador com o barramento CAN

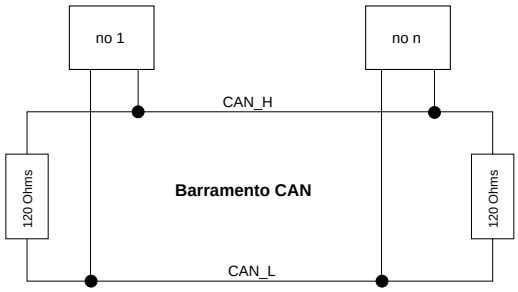

Figura 2.11 — Forma recomendada de conexão ao barramento

Uma das variantes possíveis é a utilização de acoplamento óptico, o que proporciona um desacoplamento elétrico entre o barramento e os nós.

2.3.7.2 Conexão ao Barramento

A forma recomendada de conexão ao barramento, mostrada na figura 2.11, é feita através de dois fios: CAN_H e CAN_L. Além disso recomenda-se utilizar dois terminadores compostos de resistores de aproximadamente 120Ω.

A CiA também recomenda no standard DS-102 [CIA 94a] a utilização de um conector padronizado de 9 pinos para a conexão dos nós ao barramento (tabela 2.3)

2.3.8 Protocolos de Alto Nível

Considerando-se que o protocolo CAN oferece serviços somente para transferir e requisitar dados, praticamente qualquer aplicação distribuída necessita especificar o uso do identificador e dos bytes de dados além de fornecer funcionalidade adicional

Pino	Sinal	Descrição
1	-	Reservado
2	CAN_L	Linha "baixa" do barramento
3	CAN_GND	Terra para CAN
4	-	Reservado
5	(CAN_SHLD)	Blindagem opcional para CAN
6	GND	Terra opcional
7	CAN_H	Linha "alta" do barramento
8	-	Reservado
9	(CAN_V+)	Fonte externa opcional para CAN

Tabela 2.3 — Conector recomendado pela CiA

para, por exemplo, inicialização dos nós, estabelecimento da comunicação, transmissão de pacotes de dados com mais de 8 bytes, controle de fluxo e endereçamento dos nós.

Os tópicos citados são cobertos pelos chamados HLP (*High Layer Protocols*), o qual é um termo derivado do modelo OSI de sete camadas. Alguns desses protocolos de alto nível são: CanOpen [CIA 94b], Device Net [NOO 94], Smart Distributed Systems [HON 96], CAN Kingdom [FRE 97] e J1939 [DAT 95]. Esses protocolos, apesar de desenvolvidos por empresas e/ou grupos de empresas, são abertos e permitem que qualquer um os utilize e desenvolva novos produtos sem ônus. Além disso, muitas ferramentas de *software* para programação, monitoração, configuração e módulos de *hardware* foram desenvolvidos para os protocolos citados e estão disponíveis comercialmente.

2.4 Modelagem Orientada a Objetos

A necessidade crescente de sistemas tempo-real cada vez maiores e mais dinâmicos requer *software* e *hardware* embarcados complexos, os quais tendem a ser difíceis de entender, manter e modificar. A complexidade de tais sistemas tende a aumentar com a necessidade de comportamento altamente dinâmico e adaptativo, além de restrições temporais complexas. Assim, a modelagem de um sistema com todas estas características requer métodos e técnicas que economizem tempo e facilitem o seu desenvolvimento. De acordo com [PER 98], a utilização de modelos orientados a objeto pode ser considerada a mais adequada para o desenvolvimento de sistemas tempo-real distribuídos. As características de OO (Orientação a Objetos) que apoiam isso são:

- **Compreensibilidade**: os modelos orientados a objeto permitem que os objetos do modelo representem e se comportem da mesma maneira que seus elementos correspondentes no mundo real.
- **Modularidade**: a identificação de objetos e classes proporciona uma divisão lógica do problema em partes que podem ser examinadas separadamente ou

no contexto do sistema. Quando são necessárias modificações pode-se alterar somente os objetos envolvidos deixando os demais como estão.

- **Concorrência**: em plantas reais os processos ocorrem normalmente de forma concorrente. Assim, os objetos modelados se tornam as unidades naturais para execução concorrente em tempo-real.

- **Distribuição**: em um modelo orientado a objeto o sistema tempo-real é organizado como uma coleção de objetos que cooperam entre si, os quais podem interagir passando mensagens uns para os outros. Desta forma pode-se ter arquiteturas de *hardware* e *software* implementados de forma distribuída e paralela.

- **Melhor gerenciamento da complexidade**: a orientação a objeto proporcionam conceitos abstratos poderosos para se tratar com problemas complexos. Os diferentes tipos de hierarquias facilitam a apresentação e o gerenciamento dos vários níveis de detalhes.

- **Reutilização**: as classes criadas podem ser reaproveitadas através dos conceitos de instanceamento, que permite a replicação dos atributos e comportamento para todas as instâncias de uma mesma classe, e através da herança, que pressupõe a existência de uma classe que define o comportamento geral e de subclasses que especificam serviços especializados.

2.5 Objetos Distribuídos

Uma arquitetura tempo-real distribuída modelando sensores, atuadores e nós de controle como objetos comunicantes suporta de muitas maneiras as necessidades de controle de processos com respeito à extensibilidade, confiabilidade e custo. Isto leva à um sistema de arquitetura modular no qual pequenos objetos autônomos cooperam de maneira a realizar atividades de controle. Um objeto é caracterizado por um nome e uma lista de métodos que podem ser invocados. Como o objeto é ativo ele pode também exportar informações, o que é equivalente a enviar uma invocação para outro objeto ou outro grupo de objetos sem uma requisição prévia.

A forma convencional de interação entre objetos está baseada em invocação de objetos síncrona. No caso de objetos ativos, o modelo mantém a semântica de chamada de procedimento, na qual um serviço específico é requisitado de outro objeto através de transferência de controle e troca de parâmetros via memória compartilhada, em vez de comunicação explícita. Mesmo quando se estende orientação a objetos à uma planta distribuída este modelo de invocação de objetos é preservado. Em resumo, é uma forma síncrona ponto-a-ponto coordenando dois objetos bem conhecidos (por endereço).

Um exemplo bastante simples de uma classe pode ser dado por uma válvula inteligente (fig. 2.12). Os objetos desta classe podem ser definidos como compostos por um motor e um sensor de posição da válvula, o que permite que a válvula se auto-ajuste de forma inteligente. Dessa maneira, um outro objeto pode simplesmente pedir que ela abra 50% ou 75% sem se preocupar com o controle efetivo da tarefa. Pode-se também definir uma interface para esta válvula constituída de métodos e

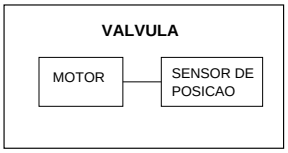

Figura 2.12 — Diagrama de classes da válvula inteligente

Métodos	**Atributos**
abrir_x	PID_motor
configurar_PID	abertura_valvula
informar_posicao	

Tabela 2.4 — Interface da válvula inteligente

atributos (tabela 2.4). O método *abrir_x* permite ajustar a sua saída em um valor entre 0 e 100%, o método *informar_posicao* fornece a posição atual da válvula e o método *configurar_PID*, por sua vez, permite que se ajuste os parâmetros internos do PID de controle. Os atributos internos do objeto são o PID do motor, que pode ser configurado externamente, e a abertura da válvula atualizada pelo sensor de posição. Este tipo de interface executa um encapsulamento que simplifica o projeto de sistemas grandes e complexos na medida em que esconde os detalhes de funcionamento que não são necessários. Utilizando objetos inteligentes como o anterior pode-se construir objetos maiores e mais complexos tal como um tanque de água composto de uma bomba, um sensor de nível e uma válvula.

3 Análise de Requisitos e Arquitetura Proposta

3.1 *Hardware*

Antes da formulação de uma proposta mais detalhada de arquitetura para sistemas automação baseados em objetos distribuídos e no barramento CAN faz-se necessário uma análise dos requisitos de *software* e *hardware* que devem ser atendidos. Assim, num primeiro momento, é necessário definir exatamente que tipo de sistema se deseja desenvolver e para que tipo de aplicações ele se destina. Em virtude de seu caráter distribuído, os sistemas de automação industrial que se propõe desenvolver podem ser inicialmente definidos como embarcados. Esta característica permite agregar inteligência a qualquer dispositivo sensor, atuador ou controlador em qualquer local de uma planta restringindo, contudo, o tamanho máximo do sistema (*hardware*).

Sistemas embarcados geralmente tem requisitos diferentes daqueles apresentados por computadores *desktop*, tais como longos ciclos de vida, confiabilidade, capacidade de tempo-real e alta dependência de custo. Além disso, esses sistemas são projetados para uma aplicação específica em lugar de proporcionarem uma grande capacidade de processamento para uso geral, como ocorre nos computadores *desktop*. Em alguns casos especiais, contudo, um sistema embarcado pode necessitar um grande capacidade de processamento, por exemplo, para processamento de imagens.

Baseando-se na divisão informal proposta por [KOO 96] para sistemas embarcados, pode-se classificar a arquitetura proposta como do tipo distribuído e possuindo características tais como: velocidade de processamento de 1 a 10MIPS, velocidade de transferência de I/O de 100Kb/s, memória de 1 a 16Mb e vida útil de 2 a 4 anos.

A arquitetura que se propõe desenvolver diferencia-se consideravelmente daquelas baseadas no barramento CAN e utilizadas em veículos automotores, as quais possuem uma velocidade de processamento bem menor e que podem ser classificadas como sistemas embarcados pequenos. A arquitetura proposta diferencia-se também de sistemas para processamento de sinais, os quais necessitam de uma grande velocidade de processamento e quantidades de memória bem maiores. A partir dessas considerações básicas pode-se analisar e definir a arquitetura em maiores detalhes com respeito ao sistema operacional, suporte para o desenvolvimento de aplicações, suporte para comunicação entre os objetos e *hardware*.

Considerando-se que grande parte dos sistemas de automação atuais são formados não só por unidades de controle (sensores, atuadores) mas também por sistemas de supervisão, decidiu-se por adotar uma arquitetura alvo contendo dois tipos de unidades de processamento (vide fig. 3.1):

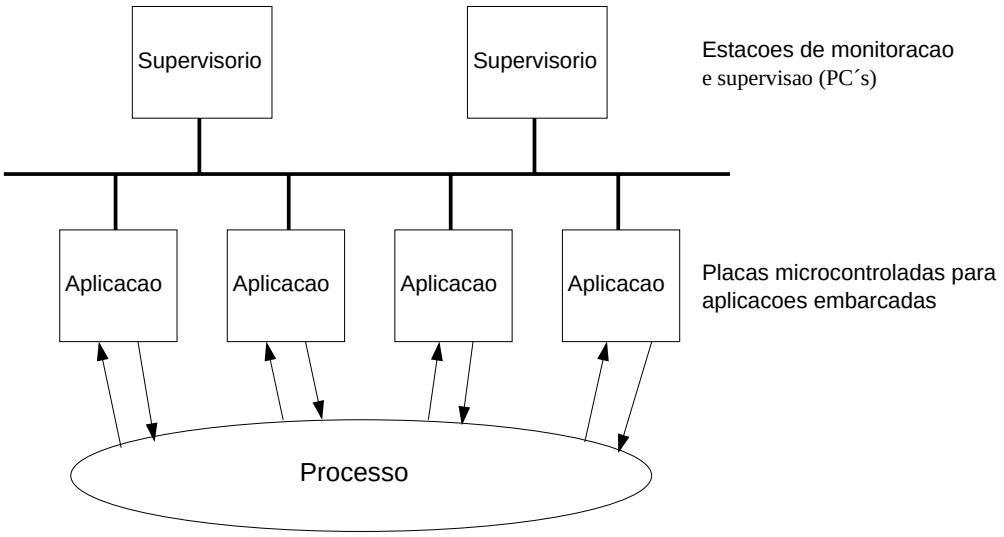

Figura 3.1 — Arquitetura básica do *hardware*

- **Placas microcontrolada**s de baixo custo, as quais conteriam interfaces para conexão com processos técnicos, a fim de permitir a aquisição de variáveis obtidas por sensores e a atuação no processo;
- **PC´s** onde seriam executados os programas de supervisão e monitoração.

3.1.1 Placas Microcontroladas

As diversas arquiteturas atualmente baseadas no barramento CAN para sistemas de automação podem ser, a grosso modo, divididas em arquiteturas simples com microcontroladores de 8 ou 16 bits e arquiteturas mais complexas, baseadas em microcontroladores de 32 bits. As arquiteturas mais simples geralmente não utilizam sistemas operacionais e o acesso ao barramento CAN é realizado através de rotinas de bibliotecas. Além disso, esses sistemas são geralmente programados em linguagem *assembler* ou C padrão, não utilizando conceitos de orientação a objetos. Sistemas desse tipo geralmente utilizam microcontroladores de 8 ou 16 bits das famílias Intel 8051, Siemens C166 e Motorola 68HC11.

Uma arquitetura deste tipo, concebida para o sistema de controle de um robô móvel autônomo, pode ser vista em [BOU 96]. Constitui-se de um sistema descentralizado formado por um nó com um PC industrial e uma série de nós compostos de uma placa microcontrolada com uma placa anexa. A placa microcontrolada é baseada em um microcontrolador 80C31 de 8 bits com 128Kb de RAM, 256Kb ROM e um controlador CAN 82C200. A placa anexa executa tarefas tais como gerenciamento de sensores e controle do motor, dependendo do nó.

As arquiteturas baseadas em microprocessadores de 32 bits, por sua vez, freqüentemente utilizam-se de algum tipo de sistema operacional e, por isso, possibilitam o desenvolvimento de sistemas de automação mais complexos. Um exemplo deste tipo de arquitetura é a adotada em [KIR 96] para uma fábrica de latas. É baseada em uma série de nós conectados a um barramento CAN constituídos de uma placa básica com um microcontrolador 68332 e uma ou mais placas adicionais,

as quais suportam sinais digitais e/ou analógicos de entrada e/ou saída. O *software*, por sua vez, é composto de uma parte fixa que manipula o barramento CAN, fornece meios para monitoração, *download* e diagnóstico e que possui um escalonador tempo-real simples.

A comparação dos dois tipos de arquiteturas citadas reforça a adoção de uma arquitetura baseada num microprocessador de 32 bits, o qual fornece capacidade de processamento suficiente para a utilização de um sistema operacional com suporte para objetos ativos. Além disso, as arquiteturas analisadas utilizam um *hardware* básico composto de placas com microcontroladores e controlador CAN e disponibilizam conectores para a instalação de placas adicionais com sensores, atuadores e outros dispositivos. Assim, definiu-se uma arquitetura de *hardware* básica composta por um microcontrolador ou microprocessador de 32 bits, memórias RAM e ROM (FLASH-ROM) e um controlador de protocolo CAN.

Dentre uma série de *kits* de *hardware* disponíveis, com quantidades de memória variáveis e diferentes características adicionais, optou-se por adquirir uma placa MEGA332 da empresa alemã MCT[1] a qual tem custo adequado e características que suportam tanto um sistema operacional embarcado quanto o barramento CAN. Suas características principais são:

- Microcontrolador MC68332
- Controlador de barramento CAN SJA1000 da Philips
- 1Mb de FLASH-ROM
- 2Mb de RAM
- UART 68681
- Conversor A/D TLC2543
- Relógio de tempo-real RTC72421

Esta placa possui importantes características como o microcontrolador MC68332 e as memórias disponíveis na placa que são suficientes para a utilização de um sistema operacional embarcado e a execução de programas baseados em objetos distribuídos, além do controlador CAN SJA1000 que suporta o protocolo CAN com identificadores de 11 e 29 bits.

Características adicionais do *kit* são as ferramentas para compilação e *software* para *download* e monitoração. A existência de um "bootloader" na FLASH-ROM da placa permite realizar *downloads* de código utilizando-se os programas DOS disponíveis para PC. As ferramentas de compilação foram utilizadas no início do trabalho somente para fins de teste. Os programas de *download*, por sua vez, foram usados durante todas as fases de desenvolvimento.

É importante lembrar que qualquer placa com características funcionais semelhantes seria capaz de suportar o arquitetura proposta. Quantidades maiores de memória podem permitir o desenvolvimento de aplicações mais complexas, com um

[1] www.mct.de

número maior de objetos distribuídos. Além disso, a utilização de um microcontrolador 68376 com controlador CAN embutido, por exemplo, pode trazer algumas vantagens como maior velocidade e simplificação do acesso ao barramento e redução da área da placa.

3.1.2 PC´s

Como dito antes, os PC´s podem ser usados como estações de supervisão e monitoração na arquitetura proposta, assim como também são usados em sistemas baseados nos barramentos Fieldbus e Profibus. Existem diferentes placas de interface para PC´s e o barramento CAN: via os barramentos internos ISA ou PCI ou via as interfaces seriais ou paralelas (centronics, RS232 ou PCMCIA). Assim, usando-se programas de interfaceamento apropriados, diversos programas de monitoração do barramento e supervisão remota da aplicação podem ser usados. Dessa maneira, optou-se pela utilização de PC´s com placas CAN-ISA, as quais apresentam excelente relação custo-benefício em função da velocidade de acesso, disponibilidade dos conectores ISA na maioria dos PC´s e custo relativamente acessível.

3.1.2.1 Placa CAN-ISA

A arquitetura típica de uma placa contém os circuitos de decodificação dos sinais e endereço do barramento ISA, um ou mais controladores CAN, transceivers para o barramento CAN e eventuais circuitos de opto-acoplamento para isolamento elétrico. A comunicação com a placa pode ser realizada através de um programa de interfaceamento (*device driver)* que forneça a funcionalidade básica para a sua operação. *Drivers* desse tipo existem tanto para o Sistema Operacional Windows quanto para o Linux e são amplamente utilizados por diversos programas comerciais em funções de teste, configuração, supervisão e outras. Além disso, existem também esquemas e *layouts* de placas CAN-ISA e seus respectivos *device drivers* para o Sistema Operacional Linux.

3.2 Sistema Operacional

A utilização do conceito de objetos ativos distribuídos e que podem executar de forma concorrente implica diretamente na utilização de um sistema operacional multitarefa que permita ter um certo número de objetos em cada uma das placas microcontroladas da rede. Tal sistema operacional deve ser, portanto, do tipo embarcado e/ou permitir o seu porte para o *hardware* escolhido, suportando também a implementação dos objetos como processos concorrentes. Além disso, um sistema operacional geralmente traz vantagens tais como suporte para sistemas de arquivos (*filesystems*), *device drivers* (comunicação serial, unidades de disco, LCD, teclado), dispositivos de rede (TCP/IP, RPC), *graphic drivers*, gerenciadores de entrada e saída, interfaces gráficas e outros módulos e dispositivos que facilitam o desenvolvimento e operação de um sistema de automação.

Atualmente existe uma série de sistemas operacionais que podem ser utilizados em sistemas embarcados. Alguns deles são versões embarcadas de sistemas operacionais conhecidos da Microsoft [MIC 00] como o Windows CE e Windows

NT Embedded, o QNX Neutrino da QNX [LTD 00] e versões embarcadas do Linux e outros são sistemas operacionais específicos para microcontroladores como os desenvolvidos pela empresa Keil [SOF 00]. A maioria deles fornece suporte para tempo-real, com diferentes níveis e metodologias de gerenciamento.

Os sistemas operacionais embarcados fornecidos pela Keil, tais como o RTX51, RTX251 e RTX166, são desenvolvidos para os microcontroladores 8051, 80251 e C166, respectivamente. São sistemas operacionais do tipo tempo-real e tem a vantagem de oferecer suporte para barramento CAN. Contudo, são desenvolvidos para microcontroladores de 8 (8051) ou 16 bits (80251 e C166) que, apesar de possuirem capacidade suficiente para rodar objetos autônomos, não permitem a utilização de um sistema operacional com suporte para sistemas de arquivos, TCP/IP e outros servicos que os sistemas operacionais para arquiteturas de 32 bits oferecem.

A QNX Software Systems[2], por sua vez, desenvolveu um sistema operacional tempo-real chamado QNX Neutrino para sistemas embarcados baseados em processadores x86 (386 a Pentium III), PowerPC e MIPS. É baseado em um *microkernel* que fornece serviços com núcleo de tempo-real para aplicações embarcadas, incluindo comunicação por mensagens, serviços para *threads* POSIX, semáforos, sinais e escalonamento. Além disso, ele pode ser estendido para suportar filas de mensagens POSIX, sistemas de arquivos, suporte para rede e outros módulos com suporte para diversos outros serviços.

A Microsoft oferece duas versões do Windows para sistemas embarcados: Windows CE, com suporte para processadores ARM, MIPS, PowerPC, SH e x86, e Windows NT Embedded. O Windows CE é um sistema operacional compacto e modular, desenvolvido para plataformas de 32 bits, que promete combinar a flexibilidade e confiabilidade do Windows com o suporte para processamento tempo-real. Além disso oferece suporte para serviços de rede, processos e *threads*, TCP/IP, sistemas de arquivos e suporte para memória *flash*.

Uma alternativa aos sistemas comerciais disponíveis consiste do sistema operacional Linux, reconhecido geralmente como sofisticado, confiável, portável e de baixo custo, apesar de não ser apropriadamente tempo-real. Mesmo não tendo recursos tão sofisticadas quanto os dos sistemas operacionais tempo-real (ex: QNX) as vantagens do Linux se sobressaem às desvantagens para muitas das aplicações embarcadas. Alguns argumentos que reforçam a utilização do Linux são: possui praticamente todas as características e ferramentas de desenvolvimento que os sistemas UNIX possuem, fornece serviços de TCP/IP, interfaces gráficas, possui compiladores para diversas linguagens tais como C, C++ e Java e, um dos mais importantes, é um sistema aberto.

Algumas versões do Linux para sistemas embarcados podem ser encontradas na forma de distribuições comerciais tais como BlueCat [WOR 00], Embedix [LIN 00a], ET-Linux [SRL 00] e Hard Hat [VIS 00]. Apesar de fornecerem versões compactas capazes de rodar em pequenas quantidades de memória, a maioria deles necessita no mínimo de um processador 386 e uma unidade de disco. Assim, sua utilização em sistemas de automação embarcados fica bastante restrita já que, normalmente, é

[2] O Sistema Operacional QNX consiste basicamente de um Sistema Operacional UNIX-RT de alta performance.

desejável rodá-los a partir de ROM e, muitas vezes, em processadores de capacidade de processamento inferior a um 386.

Existem também alguns projetos específicos baseados no Linux voltados para determinados microprocessadores ou dispositivos como: Linux CE [LIN 00b](para Handheld's), ELKS - THe Embedded Linux Kernel Subset Project [ELK 00] (para 8086) e μClinux [UCL 00] (para sistemas sem MMU-*Memory Management Unit*). Estes sistemas tem a vantagem de utilizar processadores de custo menor e de não necessitar de unidades de disco para o núcleo (*kernel*) e aplicações. Em razão da utilização de processadores diferentes da família x86 esses sistemas geralmente utilizam bibliotecas C modificadas que às vezes não possuem todos os recursos das bibliotecas padrão do Linux. Além disso, freqüentemente, é necessário desenvolver programas de interfaceamento específicos para o *hardware* utilizado apesar de que grande parte dos programas de interfaceamento disponíveis geralmente pode ser aproveitada ou portada para tais sistemas.

Fazendo-se uma comparação entre os sistemas operacionais embarcados citados (ver tabela 3.1) pode-se encontrar o mais adequado para a utilização na arquitetura proposta para sistemas de automação. Sistemas operacionais para microprocessadores de 8 ou 16 bits oferecem recursos somente para escalonamento e gerenciamento de processos concorrentes, dificultando a implementação de sistemas de controle mais complexos que necessitem de maior suporte. Os sistemas operacionais comerciais para microprocessadores de 32 bits, apesar do grande suporte para as aplicações e capacidade de tempo-real, possuem desvantagens como a necessidade de um microprocessador de grande capacidade e alto custo, além de serem sistemas fechados e não portáveis. Os sistemas operacionais embarcados baseados no Linux para microprocessadores de 32 bits, por sua vez, tem a vantagem de serem abertos e permitir o porte para diferentes microprocessadores. As respectivas versões comerciais tem a desvantagem de serem relativamente grandes e necessitarem unidades de disco, o que não ocorre com os projetos de *software* livre atualmente em andamento.

A partir desta comparação foi escolhido o Sistema Operacional μClinux, o qual é um porte de domínio público do Sistema Operacional Linux para processadores sem MMU, principalmente em razão do *hardware* necessário, baixo custo, possibilidade de ampliação e serviços disponíveis. O μClinux possui algumas outras vantagens tais como: é um *software* livre com código aberto, tem potencialmente todas as características do Linux padrão e é suportado por uma rede de desenvolvedores que se apoiam mutuamente. Além disso, o μClinux atende os requisitos necessários à arquitetura proposta, principalmente pela capacidade multi-tarefa, suporte para comunicação entre processos distribuídos, disponibilidade de *device drivers* diversos e ferramentas de compilação, além do custo virtualmente zero. No item que se segue são descritas as características gerais deste sistema operacional bem como as partes básicas que o constituem.

3.2.1 Sistema Operacional μClinux

O projeto Linux/Microcontroller é um porte do Linux 2.0 para sistemas sem MMU (Memory Management Unit). Inicialmente o sistema foi portado para microcontroladores Motorola da família 68k (68328, 68EZ328, 68332) utilizados em controle embarcado e em aplicações PDA tais como o Palm Pilot da 3Com. Re-

Sistema Oper.	**Processador**	***Boot***	**Requerimentos**	***Kernel***
RTX166	C166	ROM	2-3 Kb RAM	6-35 Kb
QNX Neutrino	x86, PowerPC e MIPS	ROM	não disponível	n. d.
Windows CE	x86, PowerPC, MIPS, ARM, SH	ROM	400 Kb ROM/RAM	n. d.
ELKS	8086	disco	não disponível	200 Kb
ET-Linux	386SX ou melhor	disco	2 Mb DRAM, 2 Mb disco	n. d.
μClinux	68XXX, Coldfire, ARM	RAM/ROM	1 Mb ROM, 1 Mb RAM	450 Kb

Tabela 3.1 — Comparação entre alguns sistemas operacionais embarcados

centemente foi realizado o porte do μClinux para microprocessadores Coldfire da Motorola e i960 da Intel e alguns outros ainda estão em desenvolvimento. Como todos os portes do Linux, o μClinux é *software* de distribuição livre sob Licença Pública GPL. O μClinux, baseado inicialmente no núcleo 2.0.33 e agora no 2.0.38, é considerado estável da mesma maneira que o compilador e as bibliotecas. Os sistemas a que o μClinux é destinado usualmente não possuem unidades de disco. Esta diferença faz com que seja necessário retirar todo o código do núcleo relativo a utilização de memória virtual. Além disso, o núcleo usualmente é colocado em FLASH-ROM o que significa que devem ser feitas algumas alterações no processo de *boot*.

As diferenças do μClinux em relação a um Linux padrão são mínimas visto que a maior parte das mudanças ocorreram em relação ao gerenciamento de memória (MMU) que na maior parte encontra-se em módulos e arquivos separados das outras funções do sistema. Todas as principais funções do núcleo são providas tais como: temporizadores (*timers*), interrupções, programas de interfaceamento, *sockets* e outros. Para facilitar a compreensão os arquivos do μClinux foram divididos de acordo com o microcontrolador e placa utilizada pelo sistema. Assim, tem-se alguns arquivos específicos para a inicialização, configuração do sistema e tratamento de interrupções, entre outros, que são diferentes para cada microcontrolador e/ou placa.

Existem dois programas compiladores, atualmente baseados no compilador GCC 2.7.2.3, no ambiente μClinux: um para o núcleo e outro para as aplicações. O núcleo é compilado com um programa compilador que gera programas no formato COFF, já que o mesmo roda diretamente na FLASH-ROM, utilizando a RAM somente para dados. As aplicações em C padrão são compiladas por um compilador C e as aplicações C++ por um compilador específico para C++. No caso das aplicações, ambos os compiladores C e C++ geram código em formato COFF que depois é transformado em formato FLAT por um programa específico de conversão. Com o código neste formato é montado um sistema de arquivos em ROM com todas as aplicações a serem utilizadas pelo μClinux. Este sistema de arquivos permite economizar espaço de memória ROM, já que é um sistema de arquivos mais compacto, permitindo também economizar RAM já que as aplicações rodam diretamente em ROM.

Apesar de estável, o compilador possui algumas limitações com relação ao tamanho máximo do código compilado da aplicação, oferece somente um suporte

básico para C++ e não suporta números do tipo *float*. Devido à utilização do formato PIC (*Position Independent Code*) nas aplicações o segmento de texto destas fica limitado a 32Kb (na verdade limitado a saltos de +-32Kb dentro do código) e o de dados a 64Kb, já que todos os saltos dentro do programa são realizados de forma relativa com base no registrador *A4* de 16 bits. Isso elimina a necessidade de relocar os programas em RAM a cada vez que são utilizados e possibilita que os mesmos sejam executados diretamente em ROM.

O μClinux é constituído pelo seguinte conjunto de ferramentas básicas disponível na página oficial do projeto que pode ser resumido por: *gcc* (compilador), *binutils* (ferramentas auxiliares para compilação), *linux 2.0.38* (núcleo), *pilot* (programas do ambiente de usuário), *genromfs* (gerador de sistema de arquivos em ROM), *coff2flt* (conversor de código em formato COFF para FLAT), *uC-libc* (biblioteca C) e *uC-libm* (biblioteca matemática).

3.3 Suporte para Desenvolvimento de Aplicações

O desenvolvimento de sistemas baseados em objetos distribuídos requer naturalmente uma programação em linguagem orientada a objetos tal como C++ ou Java. Assim, são necessárias ferramentas de compilação para a linguagem escolhida e bibliotecas de funções adequadas ao sistema operacional e ao *hardware* utilizados. Além disso, é desejável a utilização de alguma ferramenta que facilite a criação de tais sistemas e sua implementação em uma série de placas microcontroladas. Sendo assim, foram aproveitados trabalhos anteriores que buscavam dar suporte para a implementação de objetos ativos, caso da linguagem AO/C++, e para a modelagem, simulação e geração de código, caso do SIMOO-RT.

3.3.1 A Ferramenta SIMOO-RT

O SIMOO-RT [BEC 99a][BEC 99b] é um ambiente integrado para modelagem orientada a objeto, simulação e implementação de sistemas tempo-real distribuídos, especialmente aqueles concebidos para automação industrial. Esta ferramenta é parte de um trabalho de pesquisa em andamento na Universidade Federal do Rio Grande do Sul, desenvolvido no contexto do projeto ADOORATA, uma cooperação Brasil-Alemanha. Atualmente o ambiente SIMOO-RT roda no sistema operacional MS-Windows e possui um gerador de código para a linguagem AO/C++, uma extensão do C++ para suporte à criação de objetos concorrentes e distribuídos, além de requisitos temporais (como métodos cíclicos, tempo máximo de execução, etc...). Ambientes de execução para programas AO/C++ existem para os sistemas operacionais QNX e Linux.

De acordo com a metodologia do SIMOO-RT, o primeiro passo no processo de desenvolvimento é a definição de um modelo orientado a objeto do problema sob análise. Além disso, o usuário pode impor restrições temporais como, por exemplo, o período de amostragem de um sensor, e também tempo máximo de execução (*deadline*) para operações mais críticas, oferecendo tratamento de exceção para aquelas que não forem atendidas. O ambiente permite também o uso de máquinas de estado

para descrever o comportamento do modelo o que permite uma melhor descrição do funcionamento do sistema.

3.3.2 Linguagem AO/C++

A linguagem AO/C++ (*Active Objects* / C++) [PER 94] baseia-se fundamentalmente numa união das propriedades de orientação a objetos do C++ padrão com os benefícios dos sistemas UNIX-RT. A idéia principal do AO/C++ é mapear o modelo "logicamente distribuído" das linguagens orientadas a objetos, caso de C++, com o modelo "fisicamente distribuído" de um sistema operacional UNIX-RT orientado a processos, como o Sistema Operacional QNX. O AO/C++ adiciona algumas primitivas ao C++ de maneira a permitir a definição de objetos ativos, métodos disparados por tempo, especificações de restrições temporais (ex.: tempo máximo de execução e seu tratamento de exceção), etc. A comunicação inter-objeto do C++ é transparentemente mapeada para a comunicação inter-processos fornecida pelos sistemas UNIX-RT, de maneira que os programas distribuídos gerados são muito similares a programas C++ escritos para aplicações seqüenciais para somente uma máquina. Tanto comunicações síncronas quanto assíncronas são suportadas pelo AO/C++. Além disso, todas as características de orientação a objetos existentes suportadas por compiladores C++, especialmente múltipla herança, polimorfismo e *overloading* são suportadas.

As classes do AO/C++ são criadas em dois arquivos separados (exclusivos para cada classe): a estrutura da classe é declarada em um arquivo de cabeçalho (<Classe>.ph) e a definição dos seus métodos é feita em um segundo arquivo (<Classe>.pC). A partir destes dois arquivos são criadas uma classe ativa, a qual executa as instruções da instância, e uma classe passiva que é responsável por mapear as chamadas de funções C++ para mensagens destinadas ao processo relacionado e também por empacotar/desempacotar os argumentos passados e os resultados retornados. Estas duas classes corresponderiam respectivamente, dentro do padrão RT-CORBA recentemente definido [OMG 99], aos conceitos "servant" e "stub".

Este mapeamento, realizado pelo pré-processador, torna possível ativar um método de um objeto a partir de um outro objeto ativo que pode estar localizado tanto no mesmo processador quanto em um processador diferente, podendo inclusive estar em um computador (nó) diferente. Esta funcionalidade é usualmente conhecida como "chamada remota de métodos" (em inglês, RMI, "remote method invocation"). Devido a esta característica, quando um objeto ativo chama uma função de outro, a chamada é convertida, por exemplo, no Sistema Operacional QNX, para uma mensagem transmitida através da função *send()*, que realiza o envio de mensagens de forma síncrona. No receptor a mensagem é então decodificada e a função correspondente é ativada. No servidor a função de mapeamento da mensagem para a chamada de método é realizada por um módulo, também gerado automaticamente, similar a um "object adapter", OA, de CORBA.

O mapeamento automático de uma chamada de métodos é feito de maneira diferenciada caso um valor de retorno definido seja esperado ou não. Caso o método a ser executado não tenha valor de retorno, ou seja, a função é do tipo "void <função> (parâmetros)" a chamada remota é mapeada como assíncrona. Se houver tipo definido, a comunicação é mapeada síncrona. No caso de comunicação síncrona,

```
int Sensor::Amostra(cycle_t)
{
begin_cycle
    nova_temperatura = Le_temperatura(); // leitura da temperatura do sensor
    Controle->Nova_medicao(nova_temperatura);
    Supervisao->Novo_valor(nova_temperatura);
end_cycle
}
```

Figura 3.2 — Exemplo de código AO/C++

o objeto cliente fica bloqueado até que o objeto servidor processe a mensagem e retorne os dados resultantes.

Construtores especiais, semelhantes aos utilizados em linguagens de tempo real de alto nível (como PEARL), também foram adicionados ao AO/C++ para dar mais poder de expressão à linguagem. Estes construtores permitem a definição de processos com ativação periódica, garantindo os seguintes recursos: ativação cíclica de funções dos objetos; definição de tempo máximo de execução e tempo máximo de espera (tanto no lado cliente como no lado servidor), associados a blocos de instruções.

Na figura 3.2 podemos ver um exemplo de função periódica, que corresponde ao método *Amostra()*, pertencente a uma classe *Sensor*. O período de execução do método é definido pelo valor *cycle_t* e o código a ser executado ciclicamente está localizado entre as declarações *begin_cycle* e *end_cycle*. Assim, a cada ciclo o método é ativado e é feita uma nova leitura da temperatura. Esta, então, é informada aos objetos *Controle* e *Supervisao* através de chamadas de método apropriadas.

O presente trabalho faz uso de uma versão do AO/C++ para Linux desenvolvida no Laboratório de Automação. Ela é baseada na utilização de *sockets* para comunicação entre os processos e está implementada na forma de uma biblioteca de funções que suporta as funções utilizadas na versão original para QNX (servidores de nome, criação remota de processos, comunicação via chamada remota de procedimentos). O ambiente de execução contém dois processos que atuam como servidor de nomes dos objetos e servidor de nomes de *proxys* os quais tornam possível a comunicação entre objetos distribuídos. O processo de criação, geração de código e compilação das aplicações para Linux pode ser visto na figura 3.3.

Os recursos do AO/C++ utilizados neste trabalho são basicamente o suporte para a criação de instâncias e para a comunicação entre os objetos do mesmo nó. Além disso, também é aproveitada a facilidade para a criação de rotinas cíclicas e definição de tempo máximo de execução que são recursos importantes para o desenvolvimento de sistemas de automação industrial. Contudo, devido ao fato de que o Linux não é um sistema operacional tempo-real, não existem garantias de que a aplicação gerada atenderá aos requisitos temporais impostos. Esta falta de capacidade de tempo-real do Linux tem motivado o desenvolvimento de alguns métodos que tem o objetivo de dar garantias e/ou melhorar o desempenho do mesmo. A análise e implementação desses métodos, contudo, foge aos objetivos desta dissertação de mestrado e ficam como sugestão de trabalhos futuros.

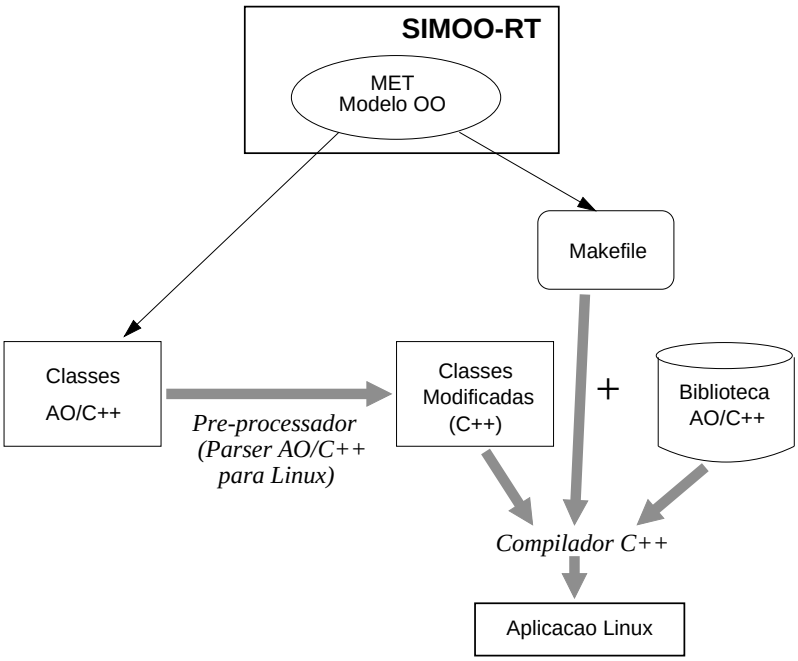

Figura 3.3 — Visão geral da geração de código e processo de compilação

3.4 Suporte para Comunicação entre os Objetos

Além dos requisitos já mencionados anteriormente, o desenvolvimento de aplicações tempo-real baseadas em objetos distribuídos exige também uma camada de infra-estrutura que suporte a comunicação entre os objetos. De acordo com [RUF 99] é necessário que essa camada de infra-estrutura forneça serviços de comunicação e gerenciamento necessários para apoiar a camada de orientação a objetos tais como comunicação de grupos, detecção de falhas, *membership*, sincronização de relógio e serviços de gerenciamento de grupo. A definição do modelo de comunicação a ser adotado pode ser melhor realizada através de uma análise dos modelos de comunicação adotados por alguns dos protocolos de alto nível existentes para o barramento CAN. Assim, segue-se uma rápida analise dos protocolos CANOpen e Smart Distributed Systems quanto ao tipo de comunicação utilizada e forma de alocação dos identificadores CAN.

3.4.1 CANOpen

O protocolo CANOpen utiliza-se de perfis de dispositivos padronizados que definem a funcionalidade básica dos dispositivos ao mesmo tempo que permitem a implementação pelo fabricante de características específicas. Utiliza o protocolo CAN 2.0A com identificadores de 11 bits e define um esquema de alocação obrigatório para eles.

Os identificadores (fig. 3.4) são previamente divididos em:

- **Código da função (4 bits)**: tipo de serviço requisitado.
- **Identificador de módulo (7 bits)**: identificador do nó

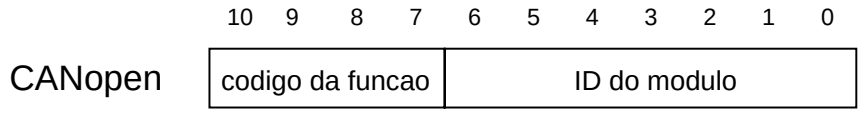

Figura 3.4 — Composição do identificador no CANOpen

O código da função identifica o tipo de serviço solicitado que pode ser NMT (*Network Management*), SYNC (*Synchronization*), TIME STAMP, EMERGENCY, PDO (*Process Data Object*) ou SDO (*Service Data Objetct*). Este esquema de alocação dos identificadores permite uma comunicação ponto a ponto entre um único mestre e até 127 escravos além de alguns serviços em modo de *broadcast*.

3.4.2 Smart Distributed Systems (SDS)

Este protocolo foi originalmente desenvolvido pela Honeywell Inc. e é apoiado pela CiA (CAN in Automation). O protocolo SDS é baseado no padrão CAN 2.0A (com identificadores de 11 bits) e cobre a camada física (*physical layer*) e a camada de aplicação (*application layer*). As mensagens de dados no SDS, chamadas de APDU (*Application Layer Protocol Data Units*), podem ser do tipo curta ou longa. Na forma curta (fig. 3.5), usada em dispositivos com somente um objeto embarcado, as mensagens não possuem dados (*Data Length Code* = 0) o identificador CAN é dividido em:

- **Dir/Pri (1 bit)**: identifica a direção de transmissão com respeito ao conteúdo do endereço lógico.
- **Endereço Lógico (7 bits)**: identifica o dispositivo transmissor/receptor.
- **Serviço (3 bits)**: identifica o serviço (ex.: *Change of State ON*, *Write ON State*).

O formato longo, por sua vez, é usado para transmitir mensagens com maior número de informações e também para assessar dispositivos com mais de um objeto embarcado. Neste caso algumas informações adicionais, ocupando dois ou mais bytes, são colocadas no campo dos dados da mensagem CAN:

- **Especificadores do Serviço (3 bits)**: possui um campo para identificar se é uma requisição ou resposta e um indicador de fragmentação.
- **Identificador do Objeto Embarcado (5 bits)**: identifica o objeto dentro do dispositivo.
- **Parâmetros do Serviço (8 bits)**: parâmetros adicionais dependentes do serviço.

No caso de mensagens fragmentadas, utilizadas para a transmissão de grande quantidade de dados divididas em partes, dois bytes adicionais são usados para identificar o número do fragmento e o número total de fragmentos.

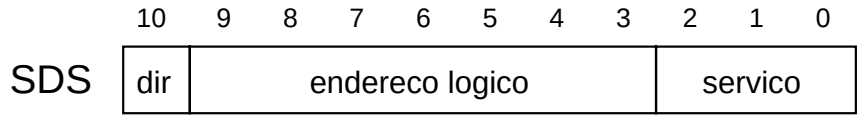

Figura 3.5 — Composição do identificador no SDS

3.4.3 Comparação entre os Protocolos de Alto Nível

Como pode-se ver nos protocolos de alto nível analisados, a comunicação geralmente utilizada no barramento CAN é baseada num endereçamento direto dos dispositivos. Desta maneira a característica de *broadcast* do CAN não é devidamente aproveitada, ficando as comunicações restritas à um padrão ponto a ponto com endereçamento de nós ou eventualmente de objetos embarcados, como ocorre no SDS. Além disso, o uso de identificadores de 11 bits limita a quantidade de nós e objetos que podem ser endereçados, dificultando sua utilização num sistema baseado em objetos distribuídos.

A utilização de um protocolo que aproveitasse a característica de *broadcast* poderia trazer algumas vantagens para sistemas baseados em objetos autônomos. Utilizando o exemplo mostrado na figura 3.6 para comparação, vê-se que, em um esquema de comunicação ponto a ponto, é necessário que o controlador envie uma mensagem, endereçada ao sensor, requisitando dados e que fique aguardando a resposta. O mesmo ocorre com o supervisório que poderia estar localizado em um PC, por exemplo. Assim, num esquema desse tipo, faz-se necessário a transmissão de quatro mensagens, no total, apesar de um único dado ter sido realmente transmitido. Caso tivesse sido utilizado o recurso de *broadcast* do CAN, por exemplo através de um protocolo do tipo "publisher/subscriber", o sensor poderia ser programado para publicar periodicamente seus dados de forma que tanto o controlador quanto o supervisório os receberiam ao mesmo tempo (fig. 3.7). Tem-se assim uma grande economia de tempo de ocupação do barramento, já que somente uma mensagem foi transmitida, em comparação com as quatro do esquema anterior.

Existem basicamente dois padrões de comunicação que utilizam a característica de *broadcast* os quais são chamados de produtor/consumidor e "publisher/subscriber" (algo como publicador/assinante). No primeiro, cada estação do barramento recebe todas as mensagens transmitidas. Cabe então as estações receptoras, e aos possíveis objetos existentes nelas, decidir se a mensagem (informação) lhes interessa. No caso de um protocolo do tipo "publisher/subscriber" é necessário que cada objeto faça uma assinatura prévia do(s) tipo(s) de informação que deseja receber. Assim, não é necessário que cada mensagem recebida seja analisada por cada objeto quanto ao conteúdo para que se possa determinar se a mesma é util.

Com as evidentes vantagens que um protocolo que aproveita a característica de broadcast do CAN pode trazer para um sistema de automação e considerando-se a característica natural de *broadcast* do barramento CAN foi então definida a utilização de um protocolo "publisher/subscriber", desenvolvido originalmente no Departamento de Estruturas Computacionais da Universidade de Ulm na Alemanha e atualmente parte do projeto ADOORATA, uma cooperação Brasil-Alemanha.

O próximo ítem descreve com mais detalhes as características e o funcionamento deste protocolo.

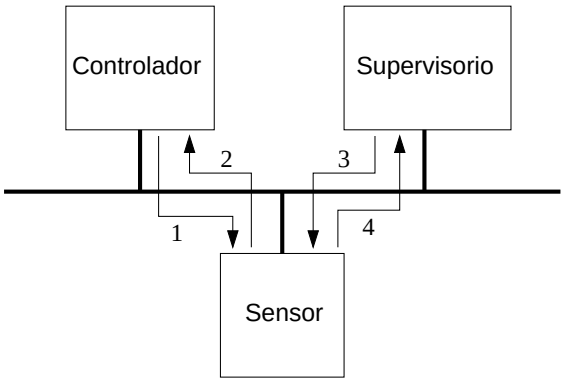

Figura 3.6 — Exemplo de sistema de automação com comunicação ponto a ponto

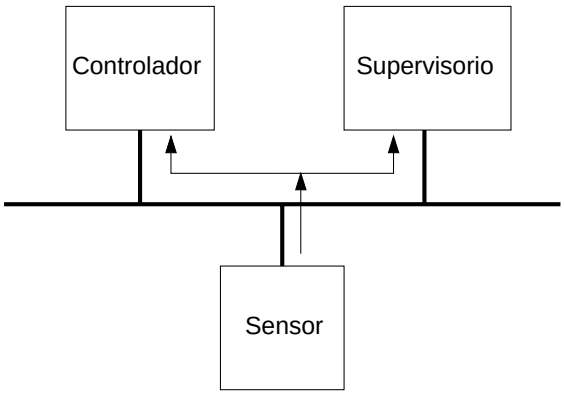

Figura 3.7 — Exemplo de sistema de automação com comunicação por *broadcast*

3.4.4 Protocolo "publisher/subscriber" para CAN

Em um ambiente distribuído cooperativo também é importante que os objetos, baseados no encapsulamento de informações (*information hiding*) com componentes reutilizáveis, preservem sua autonomia com respeito ao controle. A coordenação entre objetos é baseada em compartilhamento de informações em vez de transferência de controle motivando um padrão de interação "publisher/subscriber" entre objetos distribuídos em vez do modelo cliente/servidor convencional.

O padrão "publisher/subscriber" não somente suporta autonomia de controle mas também reflete as necessidades de sistemas de controle distribuídos com respeito à distribuição de informação. Enquanto o modelo cliente/servidor define uma relação "um para um", em um sistema constituído de sensores e atuadores inteligentes, modelados como objetos autônomos, usualmente uma relação de comunicação "um para muitos" do tipo "publisher/subscriber" é mais adequada. Neste modelo de comunicação um processo se registra para o recebimento de certa mensagem, num procedimento semelhante ao assinante de uma revista. Um publicador, por sua vez, publica instâncias deste tipo de informação no respectivo canal.

O modelo de comunicação escolhido é uma implementação do protocolo "publisher/subscriber" para barramento CAN descrito em [KAI 99]. De acordo com este protocolo, o barramento CAN é particularmente adequado à implementação de um protocolo "publisher/subscriber" altamente eficiente já que sua característica de *broadcast* suporta um esquema de endereçamento baseado no conteúdo da mensagem. O identificador da mensagem CAN pode ser convenientemente usado para especificar um certo canal de informações em vez de uma fonte ou endereço de destino. Como em um sistema de controle a informação publicada em um canal em muitos casos está relacionada a um evento gerado pela planta, o termo canal de eventos (*event channel*) é utilizado. Considerando que cada mensagem é disseminada (*broadcasted*) no barramento CAN, todos os nós conectados ao barramento recebem a mensagem e são capazes de determinar o canal de eventos a partir do identificador da mensagem. Um objeto pode assinar determinados canais de eventos e usar o identificador CAN para estabelecer um filtro local que selecione aquelas mensagens que são publicadas naquele canal. A arquitetura geral do método é mostrada na figura 3.8.

Existem dois elementos que suportam o protocolo "publisher/subscriber": o ECH (*Event Channel Handler*), que é instanciado em cada nó, e o ECB (*Event Channel Broker*) que é único para toda a rede. O ECH fornece a abstração dos canais de evento para os objetos da aplicação, além de manipular todas as tarefas de baixo nível do protocolo CAN. Assim, os objetos podem publicar (*publish*) em um canal de eventos e assinar/cancelar (*subscribe/unsubscribe*) a assinatura de um canal de eventos. Adicionalmente, o ECH configura os filtros de mensagens do controlador CAN e sinaliza a chegada de eventos à respectiva aplicação. O ECB, por sua vez, trata das tarefas de configuração dos nós e ligação dinâmica dos identificadores das mensagens com tipos de eventos.

Para facilitar a programação e isolar o programador de aplicações das questões de comunicação de baixo nível, os tipos de eventos são representados por códigos chamados de UID's. Um objeto autônomo encapsulado possui um conjunto de UID's representando todas as relações de comunicação necessárias. Ao nível da linguagem

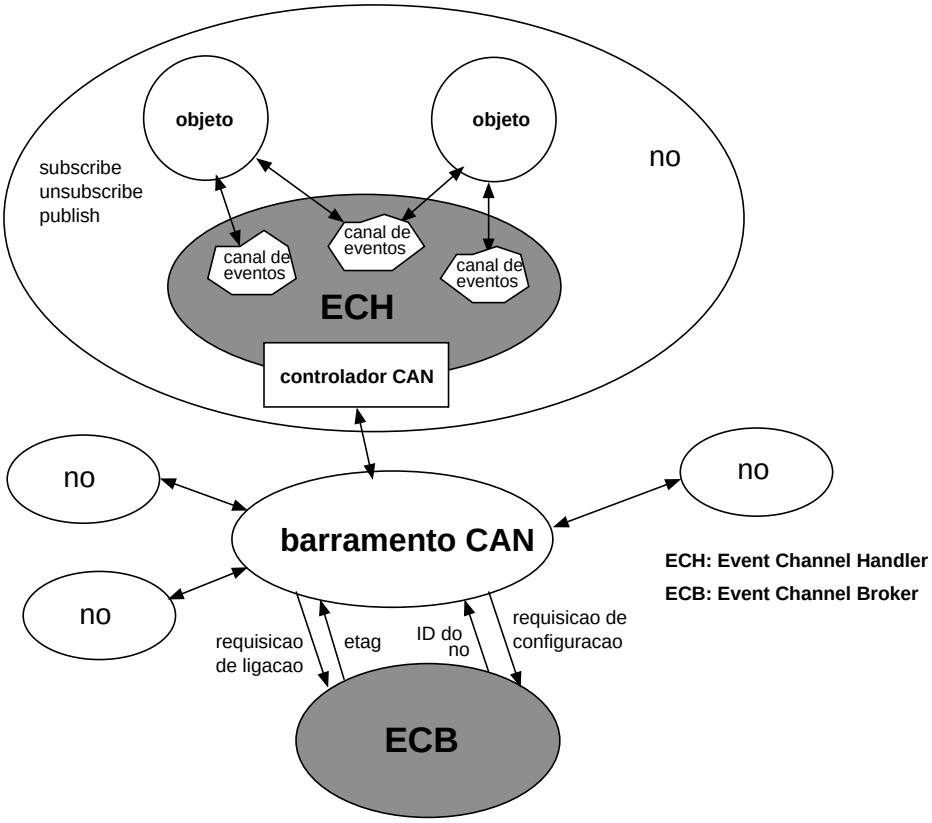

Figura 3.8 — Esquema do protocolo "publisher/subscriber"

de programação podem existir nomes específicos que devem ser mapeados para estes UID's usando um diretório de UID's. Durante a configuração do sistema ou quando um dispositivo é conectado dinamicamente, estes UID's devem ser mapeados para identificadores CAN de baixo nível, tarefa que é realizada pelo ECB. Assim, o ECB é responsável por resolver o problema de assinalar estaticamente identificadores CAN a canais de eventos, o que poderia restringir severamente a capacidade de configuração dinâmica e reconfiguração. Sempre que um objeto da aplicação deseja publicar um evento, ele usa o UID do respectivo canal de eventos. Se o CAN-ID do canal de eventos já está registrado no ECH local, este pode realizar a ligação localmente e conduzir a mensagem para publicação. Se não, o ECH requisita ao ECB, o qual tem o registro de ligação do canal que inclui o UID em seu registro e retorna o CAN-ID. Um canal específico é reservado para este protocolo. A assinatura de um canal funciona de maneira similar.

Os eventos publicados e as configurações requisitadas pelo ECH bem como as respostas dadas pelo ECB são realizadas por mensagens com formatos específicos. Assim, a configuração de um nó (*configuration request*) é requisitada através do evento de número 1 (fig. 3.9) fornecendo-se o identificador longo do nó. A resposta é dada pelo ECB através do evento número 2 (fig. 3.10) com o identificador curto (*TxNode*) transmitido no campo de dados da mensagem. A configuração dos eventos (*binding request*) é solicitada pelo evento número 3 (fig. 3.11), onde é transmitido o identificador longo do evento, e é respondida através do evento 4 (fig. 3.12), onde o identificador curto (*etag*) é fornecido pelo ECB. Além disso os eventos são

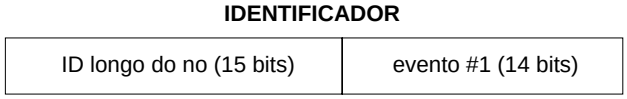

Figura 3.9 — Formato da mensagem de *Configuration request*

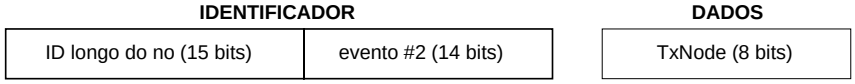

Figura 3.10 — Formato da mensagem de *Configuration reply*

publicados pelos objetos por uma mensagem com um formato específico (ver fig. 3.13) que divide o identificador CAN em três campos:

- *Priority* (8 bits): associa uma prioridade à mensagem.
- *TxNode* (7 bits): código associado ao nó.
- *etag* (14 bits): código associado ao evento.

O protocolo, contudo, não especifica a forma de utilização dos dados, deixando em aberto a quantidade de bytes a ser utilizada e que tipo será usado (bytes, inteiros, etc...).

3.5 Visão Geral da Arquitetura Proposta

A arquitetura definida a partir do detalhamento realizado nos ítens anteriores pode ser resumida no seguinte conjunto de ferramentas e metodologias (ver fig. 3.14):

1. Modelagem orientada a objetos usando objetos ativos.
2. Rede de placas microcontroladas para a aplicação e PC's para supervisão.
3. Sistema Operacional μClinux para as placas microcontroladas e Linux para os PC´s.
4. Comunicação por barramento CAN utilizando protocolo "publisher/subscriber".

A idéia geral consiste em criar sistemas de automação usando modelos OO com o auxílio do SIMOO-RT e gerar o código da aplicação em AO/C++ para Linux. Com este código pode-se então fazer a distribuição da aplicação para uma série de placas microcontroladas interconectadas por um barramento CAN. O Sistema Operacional μClinux utilizado nas placas, sendo orientado a processos, oferece o suporte para a implementação dos objetos na forma de processos concorrentes. A comunicação

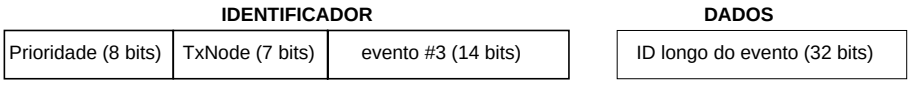

Figura 3.11 — Formato da mensagem de *Bind request*

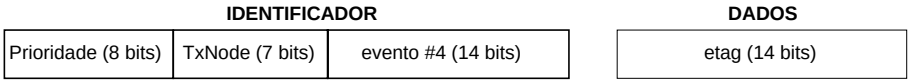

Figura 3.12 — Formato da mensagem de *Bind reply*

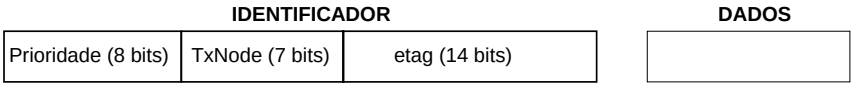

Figura 3.13 — Formato de uma mensagem de evento

entre os objetos, por sua vez, é suportada por um protocolo "publisher/subscriber" desenvolvido para aproveitar a característica de *broadcast* do barramento CAN.

Devido às limitações de desempenho do 68332, aplicações que demandem maior poder de processamento tem a possibilidade de utilizar uma placa microcontrolada semelhante baseada num processador Motorola Coldfire ou Intel i960 que também permitem a utilização do μClinux. Dessa maneira, é possível portar todo o *software* desenvolvido para diferentes plataformas rodando μClinux permitindo, assim, a utilização de uma rede heterogênea com placas rodando o mesmo sistema em processadores diferentes.

O SIMOO-RT foi escolhido como ferramenta para modelagem e geração de código principalmente porque sua interface gráfica facilita enormente a compreensao e gerenciamento das aplicações. Além, disso o recurso de geração de código em AO/C++ pode ser plenamente aproveitado para a arquitetura proposta. Contudo, algumas alterações mínimas no código gerado são necessárias já que o protocolo de comunicação a ser utilizado é o "publisher/subscriber" e não ponto a ponto, como no AO/C++. Atualmente é necessário que essas alterações sejam feitas manualmente, nada impedindo, contudo, que no futuro seja adicionada uma opção ao SIMOO-RT para a geração automática de código para a presente arquitetura.

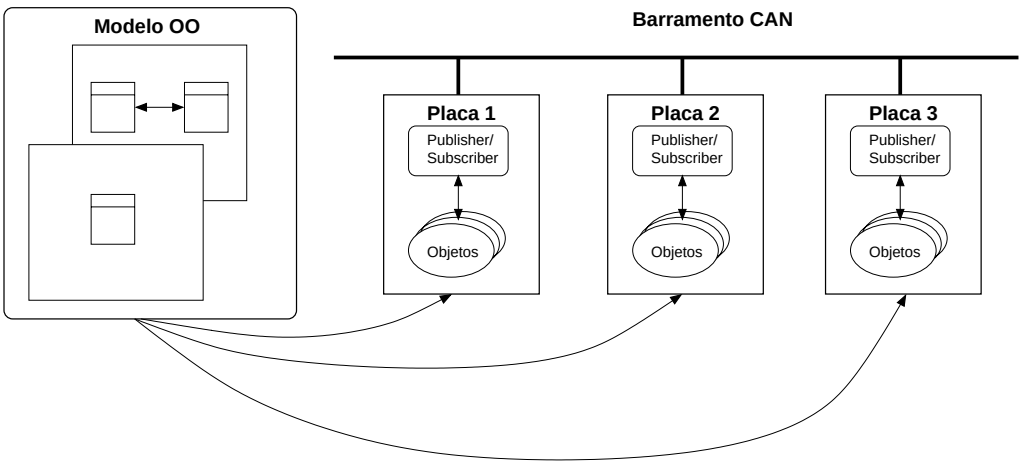

Figura 3.14 — Visão geral da arquitetura proposta

4 Arquitetura Desenvolvida

Seguindo-se à fase de concepção da arquitetura proposta para objetos distribuídos baseada no barramento CAN, passou-se ao desenvolvimento do suporte computacional necessário. Esta fase constituiu-se, assim, do porte do sistema operacional μClinux para uma placa microcontrolada, desenvolvimento de *device driver* para o controlador CAN SJA1000, suporte para utilização de C++ e finalmente o desenvolvimento do suporte para a comunicação "publisher/subscriber". As várias partes desta fase de implementação são descritas com maiores detalhes nas seções que se seguem e um esquema geral da arquitetura é mostrado na figura 4.1.

4.1 Versão do μClinux para a Placa MEGA332

A versão do Sistema Operacional μClinux desenvolvida para a placa MEGA332 constituiu-se de modificações em *scripts* de configuração e *makefiles*, na definição de parâmetros relativos ao mapeamento da memória da placa e modificações nas rotinas de tratamento de interrupções. Inicialmente foi usado o μClinux baseado no núcleo 2.0.33 para a criação da versão para a MEGA332 a qual, mais tarde, foi substituída por uma versão do μClinux com núcleo 2.0.38. Esta versão mais recente foi adotada já que apresentava algumas vantagens como o conceito de separação do código fonte por microcontrolador e por placa, facilitando o entendimento e manipulação do código. Além disso, alguns problemas (*bugs*) com a versão anterior foram corrigidos pelos desenvolvedores tornando o sistema mais estável e confiável.

Para o desenvolvimento de uma versão do μClinux para a MEGA332 foram utilizados os arquivos-fonte fornecidos na página oficial do projeto os quais são formados por um grande número de arquivos C, cabeçalhos, *scripts* e outros. Assim, a partir dessa instalação básica do μClinux foi criada uma versão para a MEGA332 através da modificação e/ou criação de uma série de arquivos, tais como: inicialização do núcleo (*boot*), *scripts* de configuração, cabeçalhos e *layout* da memória da placa (RAM e ROM). Uma das alterações realizadas foi a adição da opção para o suporte para a MEGA332 e para o *device driver* do controlador CAN SJA1000, facilitando a configuração do núcleo e geração do código para a placa utilizada. O resultado final das modificações realizadas pode ser visto no *boot* mostrado na figura 4.2.

Uma outra alteração feita foi com relação às funções de requisição e tratamento das interrupções do sistema, as quais foram reformuladas de maneira a utilizar a forma padrão do Linux que consiste, basicamente, das funções *request_irq()* e *free_irq()*. No código original para o MC68332 era necessário adicionar manualmente novas interrupções na rotina de tratamento do núcleo, tornando impossível requisitar novas interrupções em tempo de execução. Assim, a rotina desenvolvi-

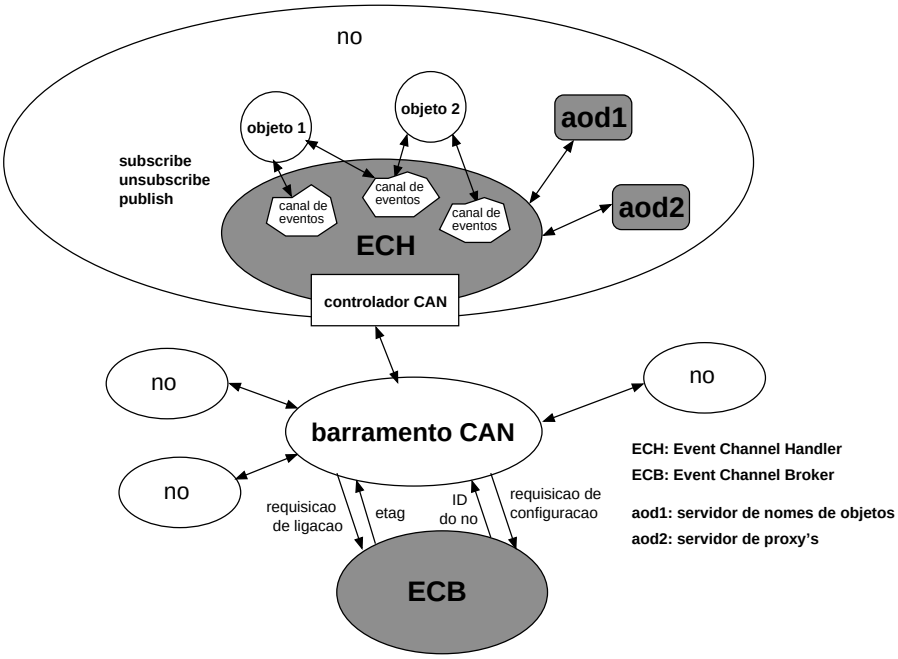

Figura 4.1 — Esquema da arquitetura desenvolvida

da para o MC68328 foi adaptada de maneira que as interrupções requisitadas são habilitadas através da habilitação dos pinos da porta F do MC68332. Foi utilizado este recurso para a solução do problema já que o MC68332, ao contrário do MC68328, não possui uma variável interna responsável por habilitar e desabilitar as interrupções individualmente.

Com o objetivo de reduzir o tamanho do núcleo ao mínimo necessário este foi reconfigurado de maneira que somente os módulos e programas de interfaceamento realmente necessários fossem compilados. Assim, dos 896Kb disponíveis para código na FLASH-ROM da MEGA332, em torno de 485Kb são ocupados pelo núcleo, pelo interpretador de comandos (*shell*) e seus relativos arquivos executáveis. O arquivo inteiro que é carregado na placa ocupa algo em torno de 614Kb deixando somente 282Kb para as aplicações. Dessa maneira, o desenvolvimento de aplicações maiores necessitaria de uma quantidade adicional de FLASH-ROM, principalmente considerando-se aplicações orientadas a objeto com grande número de classes e instâncias.

Algumas adaptações também foram feitas no ambiente de usuário e aplicações. O programa que inicializa o sistema após o *boot* montando diretórios e iniciando algumas aplicações, entre outras tarefas, sofreu alterações. Alguns parâmetros da rotina de configuração da comunicação serial com o PC foram modificados e, além disso, alguns programas que eram ativados automaticamente foram retirados de forma a reduzir o tamanho do sistema ao mínimo necessário.

```
ABCDEFGHI
MEGA332 support (C) 1999 Cristiano Brudna (GCAR-UFRGS)
Flat model support (C) 1998,1999 Kenneth Albanowski, D. Jeff Dionne
Calibrating delay loop.. ok - 2.35 BogoMIPS
Memory available: 1844k/1981k RAM, 576k/896k ROM (140k kernel data, 320k code)
Swansea University Computer Society NET3.035 for Linux 2.0
NET3: Unix domain sockets 0.13 for Linux NET3.035.
Swansea University Computer Society TCP/IP for NET3.034
IP Protocols: ICMP, UDP, TCP
uClinux version 2.0.38.0 (brudna@hendrix) (gcc version 2.7.2.3) #215 Thu May 4 16:40:53 BRT
2000
MC68332 serial driver version 1.00
ttyS0 is a builtin MC68332 UART
MC68332 CANbus-SJA1000 driver version 1.2
MC68332 RTC72421 driver version 1.0
RTC72421: 05/06/00 14:06:38
Ramdisk driver initialized : 16 ramdisks of 4096K size
Blkmem copyright 1998,1999 D. Jeff Dionne
Blkmem copyright 1998 Kenneth Albanowski
Blkmem 1 disk images:
0: 278960-2C795F (RO)
loop: registered device at major 7
VFS: Mounted root (romfs filesystem) readonly.

================================================================

                Welcome to uClinux/Pilot!

uClinux release 2.0.38.0, build #215 Thu May 4 16:40:53 BRT 2000
uClinux/Pilot release 1.0.0, build #198 Thu May 4 16:40:34 BRT 2000

================================================================

Mounting proc on /proc
Expanding initial ramdisk image into /dev/ram0
Mounting /dev/ram0 on /var
Making /var/tmp
Attaching loopback device
Starting task manager
uClinux command shell (version 1.1)
/>
```

Figura 4.2 — Boot do μClinux na placa MEGA332

4.2 Ambiente de Desenvolvimento

4.2.1 *Download* do Código

O *kit* adquirido possui um programa de monitoração para DOS que se comunica via porta serial com o *bootloader* da placa MEGA332. Tendo em vista a utilização do Sistema Operacional Linux como ambiente de desenvolvimento, tornou-se inviável a utilização de um segundo computador rodando DOS para a realização de *download* do código gerado. Assim, optou-se por utilizar um emulador de DOS para Linux chamado DOSEMU, o que permitiu utilizar o programa de *download* do *kit* dentro do Linux. Desta maneira, pode-se modificar e compilar o μClinux e após fazer o seu *download* para a placa utilizando-se somente um computador com Sistema Operacional Linux.

4.2.2 Terminal Serial para a MEGA332

Dentre as várias formas possíveis de comunicação com a placa optou-se pela criação de um programa terminal para PC baseado numa comunicação direta via porta serial. Isto permite que inicialmente se capture as mensagens do *boot* do sistema e depois se envie comandos de *shell* para o μClinux, acompanhando-se também a execução dos programas. O programa terminal é capaz de detectar o símbolo de *prompt* ($/>$) do μClinux, permitindo assim ao usuário digitar comandos que serão enviados à placa quando se tecla ENTER.

4.3 *Device Driver* para o SJA1000

O *device driver* do SJA1000 foi desenvolvido com o objetivo de facilitar a transmissão e recepção de mensagens do barramento CAN de acordo com a interface padrão do Linux. O código do *device driver* foi desenvolvido com base em [RUB 98] e as funções implementadas são: *open()*, *close()*, *read()*, *write()* e *ioctl()*. São utilizadas interrupções para o recebimento de mensagens e *buffers* de recepção e transmissão. As limitações existentes são que somente uma aplicação pode acessar o *device driver* do SJA1000 de cada vez e somente uma mensagem pode ser escrita ou lida de cada vez.

O acesso ao *device driver* do SJA1000 é provido por uma "entrada" no diretório de dispositivos do μClinux (*/dev*) que deve ser aberto inicialmente por um comando *open()* para possibilitar as operações subseqüentes. A função *open()* é responsável por configurar inicialmente o SJA1000 em modo TouCAN, o qual permite o uso de quadros estendidos, com taxa de transmissão de 100 Kbit/s e é habilitada a geração de interrupções de recepção e de transmissão. A taxa de transmissão pode ser reconfigurada depois pelo usuário através da função *ioctl()* do *device driver* para as taxas de 100, 250, 500 ou 1000Kbit/s as estão pré-programadas no device driver. No lado do microcontrolador 68332, é requisitada a interrupção *IRQ5* a qual está conectada ao pino de geração de interrupção do SJA1000.

Uma operação de leitura é executada através da função *read()* e pode ser solicitado o modo de bloqueio ou não-bloqueio. Caso seja feita uma leitura do dispositivo no modo de bloqueio e não exista nenhuma mensagem no *buffer*, o programa do

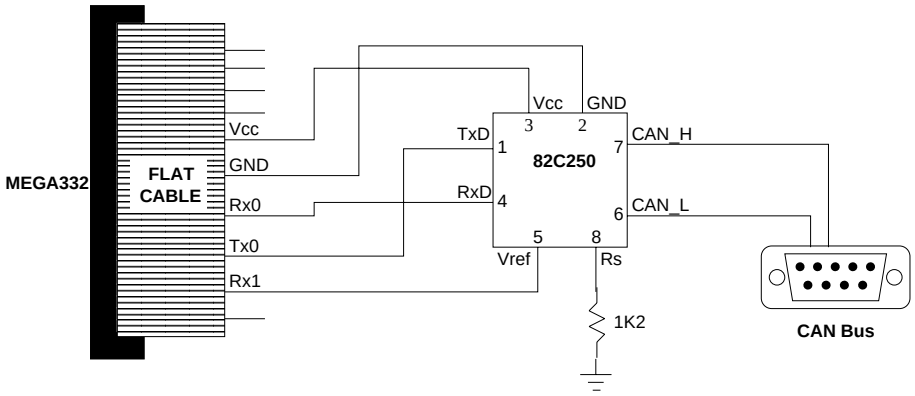

Figura 4.3 — Circuito com transceiver para a MEGA332

usuário e o *device driver* são colocados em um estado de espera. Quando uma interrupção de recepção gerada pelo SJA1000 é recebida pelo *device driver*, este lê a mensagem recebida, coloca-a no *buffer* de recepção e desbloqueia a aplicação transferindo também a mensagem a ela. Caso seja feita uma leitura no modo de não-bloqueio e não exista nenhuma mensagem no *buffer* do *device driver* este retorna um valor -EAGAIN, indicando à aplicação que deve tentar de novo. A função *read()* retorna 13 bytes contendo o *dlc* (*data length code*), o identificador da mensagem e os dados.

A função *write()*, por sua vez, faz com que o *device driver* armazene a mensagem no *buffer* de transmissão e tente transmiti-la. Caso não seja possível no momento o *device driver* aguarda a próxima interrupção, de recepção ou de transmissão, e então tenta novamente. Algumas funções adicionais tais como configuração da taxa de transmissão e das máscaras de filtragem podem ser acessadas através da função *ioctl()*.

Os testes de comunicação foram inicialmente realizados com a ajuda de uma placa composta de um microcontrolador 8031 e de um controlador CAN 82C200 capaz de manipular somente mensagens no formato *standard* (identificadores de 11 bits). Posteriormente também foram feitos testes de comunicação entre a MEGA332 e um PC com uma placa CAN-ISA com a utilização de quadros estendidos de 29 bits. Além disso, foi montada uma pequena placa (fig. 4.3) com um *transceiver* 82C250 para conexão do controlador CAN SJA1000 da MEGA332 com o barramento CAN.

4.4 Adaptações para Compilar Código C++

A utilização de aplicações modeladas com técnicas de orientação a objetos e implementadas em código AO/C++ levou à necessidade de um compilador C++ para o μClinux.

Assim, foi modificada a opção de geração do compilador, existente em um arquivo de *makefile* do mesmo, para que fossem gerados os programas executáveis que compilam tanto C quanto C++. Além disso, a biblioteca C padrão do μClinux teve de ser modificada a fim de manter a compatibilidade com o formato C++. O resultado final das alterações realizadas foi a criação de um novo *kit* para o compilador, capaz de gerar automaticamente o compilador C++ necessário, e de uma

nova biblioteca C compatível com C++, que visam facilitar a tarefa de instalação do sistema.

Um pequeno programa em C++ também foi criado para testar o compilador gerado. Este pequeno programa exemplo serviu para configurar adequadamente o sistema e criar um *Makefile* adequado para os programas C++. O novo compilador gerado, por sua vez, é capaz de compilar todas as características da linguagem C++ com algumas pequenas restrições, já que sua versão não é a mais recente. Além disso, é importante ressaltar que a biblioteca existente é de somente funções C padrão e não de C++.

4.5 AO/C++ para μClinux

A utilização da linguagem AO/C++ para o desenvolvimento dos sistemas de automação e a futura utilização do SIMOO-RT para a modelagem e geração de código nesta mesma linguagem fez com que fosse necessária uma biblioteca AO/C++ para o μClinux. Para isto, a biblioteca AO/C++ existente para Linux teve de sofrer algumas pequenas alterações para que funcionasse no ambiente proposto. Dessa maneira, foram necessárias algumas adaptações no código e substituição de funções assim como algumas modificações na biblioteca C do μClinux.

Para testar esta versão do AO/C++ foram criadas duas aplicações: um sistema simples com somente uma classe pai e duas classes filha e um sistema de aquecimento para o teste de funções periódicas. O primeiro sistema permitiu testar a biblioteca criada assim como a comunicação entre os objetos e o correto funcionamento dos *daemons aod1* e *aod2*. A segunda aplicação, como dito antes, teve como objetivo testar as funções periódicas que são suportadas pelo *timer* existente na biblioteca AO/C++ para Linux/μClinux.

4.6 Suporte para "publisher/subscriber"

Com o objetivo de aproveitar os recursos já fornecidos pelo AO/C++ foi adotada uma estratégia de integração do protocolo *publisher/subscriber* ao primeiro. Assim, o suporte para a comunicação "publisher/subscriber" foi basicamente desenvolvido através da adição das funções *subscribe()*, *unsubscribe()* e *publish()* à biblioteca AO/C++ e do desenvolvimento do ECH e do ECB na forma de *daemons*.

No AO/C++ original a comunicação entre os objetos é sempre realizada num padrão ponto-a-ponto utilizando-se para isso uma chamada remota de método. Esta chamada remota (ex.: *motor->Ligar()*) é sempre traduzida, de forma transparente para o programador, para uma mensagem a ser enviada ao objeto de destino. Assim, conhecendo-se o nome do objeto de destino, *motor* por exemplo, obtêm-se o identificador de seu processo (PID) e no caso do Linux, também a porta de comunicação a ele associada, através de uma requisição ao servidor de nomes de objetos *(aod1)*. Após isso, o nome do método e seus eventuais argumentos (dados) são empacotados em uma mensagem que é enviada para o objeto de destino através de *sockets*, na versão para Linux, ou através da função *send()*, na versão para QNX. O objeto de destino, por sua vez, desempacota a mensagem e ativa o método indicado passando também os dados, caso existirem.

Na nova versão do AO/C++ desenvolvida, a qual possui suporte para "publisher/subscriber", não é necessária nenhuma informação com relação ao objeto de destino, já que não há uma comunicação direta entre ele e o objeto de origem. Em lugar da necessidade de se conhecer o nome do objeto e do método que se deseja invocar, somente é necessário conhecer-se o nome do evento e o formato dos dados eventualmente associados a ele (ex.: "LIGAR_MOTOR"). Assim, todas as operações de comunicação são realizadas por intermédio de uma solicitação de serviço ao ECH o qual é responsável pelo mapeamento das funções solicitadas em operações de baixo nível.

Para a comunicação dentro dos nós, ou seja, entre os objetos e o ECH, foi utilizada uma comunicação baseada em *sockets* do tipo *internet* de forma a manter a compatibilidade com a forma de comunicação do AO/C++ para Linux. Os *sockets* do tipo internet permitem a comunicação entre dois processos diferentes, os quais podem estar em máquinas diferentes, através da especificação de uma porta de comunicação e de um endereço TCP/IP apropriados associados ao *socket*[BLO 92]. Com base nisso, cada uma das três funções criadas é capaz de transmitir por *sockets* uma mensagem ao ECH solicitando o serviço correspondente (*subscribe*, *unsubscribe* ou *publish*) e contendo os parâmetros fornecidos pelo objeto. As funções adicionadas à biblioteca do AO/C++ tem os seguintes argumentos:

- *int* subscribe(*char* *event_name, *char* event_uid, *char* *object_name)
- *int* unsubscribe(*char* *event_name, *char* event_uid, *char* *object_name)
- *int* publish (*char* *event_name, *char* event_uid)
- *int* publish (*char* *event_name, *char* event_uid, *int* data1)
- *int* publish (*char* *event_name, *char* event_uid, *int* data1, *int* data2)

Os eventos a serem recebidos por um objeto são assinados através da função *subscribe(),* onde o parâmetro *event_name* corresponde ao nome do evento, o *event_uid* corresponde ao código numérico do evento (gerado em tempo de compilação com base na lista de eventos) e o *object_name* corresponde ao nome alfanumérico do objeto "assinante". O parâmetro *event_name*, apesar de não ter utilidade para o ECH, é utilizado para facilitar a compreensão e a depuração de erros por parte do programador. Um exemplo de assinatura de evento poderia ser:

subscribe("TEMPERATURA", TEMPERATURA, "controlador");

Da mesma maneira, a função *unsubscribe()*, cujos argumentos são idênticos aos da função *subscribe()*, pode ser utilizada para remover a assinatura de uma mensagem a qualquer momento durante a execução do código. A publicação de eventos, por sua vez, é realizada através da função *publish()* onde, como antes, o parâmetro *event_name* corresponde ao nome do evento e o *event_uid* corresponde ao código numérico do evento. Além disso, foi aproveitado um recurso do C++ que permite distinguir funções de nome igual mas com argumentos de número e/ou tipo diferentes, de maneira que pode-se publicar uma mensagem com nenhum, um ou dois dados utilizando-se a mesma função *publish()*. Um exemplo de publicação de eventos poderia ser:

publish("TEMPERATURA", TEMPERATURA, temp);

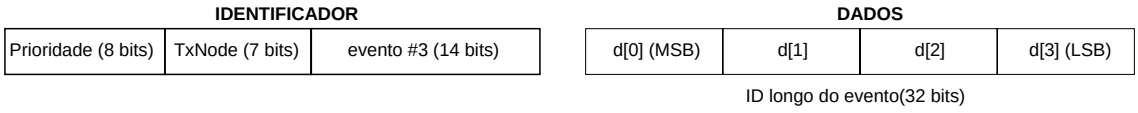

Figura 4.4 — Formato do *bind_ request* implementado

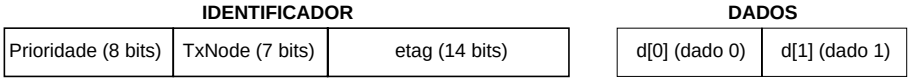

Figura 4.5 — Formato da mensagem de evento implementada

4.6.1 Event Chanell Handler (ECH)

Dentre as diversas tarefas executadas pelo ECH, a primeira delas consiste em inicializar e configurar o barramento CAN além de abrir um *socket* para o recebimento de mensagens dos objetos do nó. Após isso, é feita a configuração do nó que consiste na solicitação ao ECB de um identificador CAN para o nó (*TxNode*), o qual será utilizado nas próximas mensagens a serem transmitidas. A implementação do ECH foi feita de forma que seu laço principal consiste em verificar primeiro se foram recebidas mensagens pelo barramento e, em seguida, verificar mensagens recebidas por *sockets*. Ambas as verificações são realizadas em modo de não-bloqueio, de forma que, se não existirem mensagens no *buffer* do respectivo *device driver* ou *socket*, o ECH não fica trancado aguardando.

Os identificadores CAN são tratados no ECH e no ECB e não nos objetos, onde a utilização de constantes com os nomes dos eventos (ex.: FECHAR_PORTA, ABRIR_PORTA) facilita a leitura e compreensão do código da aplicação. O ECH mantém duas listas para uso interno: a primeira é uma lista de assinaturas onde cada entrada relaciona um objeto com o evento assinado e a segunda é uma lista contendo identificadores (UID's) dos eventos e seus respectivos identificadores CAN (*etags*).

Como dito anteriormente, os serviços solicitados pelos objetos do mesmo nó através de *sockets* e executados pelo ECH são *subscribe*, *unsubscribe* e *publish*. A cada vez que um pedido de assinatura (*subscribe)* chega ao ECH este adiciona o nome do objeto e o respectivo evento à lista de assinaturas. Feito isso, é verificado na lista de identificadores CAN de eventos (*etags)* se já existe um *etag* associado ao evento e, caso não exista, o ECH solicita ao ECB um novo identificador e então armazena-o na sua lista. A solicitação é feita através de uma requisição de ligação (fig. 4.4) com o identificador longo do evento sendo passado como dado, onde o byte mais significativo é o dado 0 e o menos significativo é o dado 3. A operação *unsubscribe*, por sua vez, consiste simplesmente em remover a entrada correspondente da lista de assinaturas.

Sempre que um pedido de publicação (*publish*) é recebido o ECH consulta sua lista de identificadores CAN de eventos e, caso o mesmo não exista, é solicitado ao ECB que crie e envie um como resposta. Existindo um identificador, o ECH prepara a mensagem para ser transmitida ao barramento CAN (fig. 4.5) formatando os campos do identificador e adicionando os dados fornecidos pelo objeto. Na

implementação realizada foi escolhida a utilização de dois dados, cada um com um byte, dos oito *bytes* possíveis numa mensagem de dados do protocolo CAN.

A chegada de uma mensagem pelo barramento CAN, por sua vez, resulta inicialmente em uma análise e decomposição dos campos da mesma, onde o ECH extrai a prioridade, *TxNode*, *etag* e os dois possíveis dados da mensagem. Em seguida, se não for uma mensagem de configuração, o UID que corresponde ao identificador CAN (*etag)* da mensagem recebida é procurado na lista de eventos do ECH. Caso o UID não exista isso significa que o evento não é utilizado no nó e a operação é encerrada. Caso ele exista é feita uma varredura na lista de assinaturas para se verificar se algum objeto assinou o respectivo evento. Existindo uma assinatura, a mensagem é transmitida via *sockets* para o objeto assinante e, após, caso não tenha atingido o final da lista, o ECH passa à verificação da próxima entrada da lista de assinaturas.

Apesar da utilização de programas de interfaceamento, os quais possuem interfaces padronizadas com respeito à biblioteca C e ao núcleo do Linux, a utilização de controladores diferentes pode requerer procedimentos de configuração e operação diferentes. Assim, foi usada uma programação baseado em uma biblioteca de "classes de barramento" que fornecem o suporte para diferentes controladores CAN. Foram então desenvolvidas classes para o controlador CAN SJA1000 da MEGA332 e para uma placa CAN-ISA com dois controladores 82527. Além disso, o desenvolvimento e teste do protocolo "publisher/subscriber" foi facilitado pela criação de uma classe *Ethernet* que permite a comunicação por *sockets* entre o ECB e o ECH. A troca da classe depende, desse modo, de uma simples modificação em uma linha de código do ECH a qual define o barramento utilizado (ex.: *Ethernet bus*, *CAN_SJA1000 bus*).

Os métodos a serem suportados por cada classe são:

- **init**: abre o *device driver* e configura o controlador CAN.
- **end**: fecha o *device driver.*
- **config_request**: codifica uma requisição de configuração e transmite pelo barramento. Após, decodifica a resposta do ECB e devolve o *TxNode.*
- **bind_request**: codifica uma requisição de *etag* e transmite pelo barramento. Após, decodifica a resposta do ECB e devolve o *etag.*
- **read_message**: verifica se há mensagens no *device driver*, decodifica de acordo com o formato e retorna a prioridade, o *TxNode*, o *etag* e os dois dados.
- **write_message**: codifica uma mensagem para o respectivo barramento e transmite.

A utilização de classes diferentes para cada controlador ou placa permite que a formatação das mensagem CAN seja feita de maneira transparente para o ECH, ou seja, tanto os campos do identificador quanto os dados são formatados exclusivamente pelos métodos da classe sem conhecimento do ECH.

```
ECB daemon
Monitoring CAN network

0000C001 [ ] 9587082
ecb: CONFIG REQUEST
ecb: long_node_id=3  TxNode(short)=7
Enviando resposta: 0000C002 [ 07 ] -1073744632


0001C003 [ 00 00 00 04 ] 9589395
ecb: BIND REQUEST
ecb: long_id=4
ecb: new event  long_id=4 etag=10
Enviando resposta: 0001C004 [ 00 0A ] -1073744632

0001C003 [ 00 00 00 05 ] 9589403
ecb: BIND REQUEST
ecb: long_id=5
ecb: new event  long_id=5 etag=11
Enviando resposta: 0001C004 [ 00 0B ] -1073744632
```

Figura 4.6 — Operações do ECB

4.6.2 Event Channel Broker (ECB)

O ECB é responsável por fazer a configuração dos nós e fornecer identificadores CAN de eventos quando solicitado. Este programa ("daemon"*)* recebe os requisições de configuração de nó e de ligação e responde fornecendo identificadores CAN para o nó (*TxNode*) e para os eventos (*etag*), respectivamente. Os identificadores CAN e códigos dos eventos (*UID's*) são armazenados em uma lista geral de toda a aplicação.

Os pedidos de configuração de nó (fig. 3.9) são respondidos com um valor de *TxNode* gerado por um contador interno. Não há manutenção de uma lista interna de nós que indique o *TxNode* correspondente. Com relação aos pedidos de configuração de eventos, cada vez que um deles é recebido o ECB verifica se já existe um identificador CAN designado para o evento e, em caso negativo, é criado um novo identificador e o mesmo é armazenado na lista. Após isso, o identificador do evento é transmitido ao barramento CAN numa mensagem apropriada (fig. 3.12), de acordo com o protocolo "publisher/subscriber", e recebido pelo ECH solicitante. Uma amostra do ECB em funcionamento pode ser vista na figura 4.6.

4.6.3 Código dos Objetos

Tendo-se em vista a utilização da arquitetura proposta foi necessária uma reestruturação e algumas modificações no código das aplicações geradas pelo SIMOO-RT. Estas modificações estão relacionadas com a utilização do protocolo de comunicação "publisher/subscriber", com o suporte para a criação de instâncias e com a distribuição da aplicação. O suporte fornecido pelo AO/C++ para Linux permite a criação remota de processos em qualquer computador conectado a uma rede *ethernet* através da utilização de primitivas e serviços baseados em TCP/IP. Na arquitetura proposta isto não é possível, já que o sistema operacional μClinux não suporta a criação remota de processos no barramento CAN.

```
#ifndef _Door_
#define _Door_
#include <stdio.h>
#include <conio.h>
active class Door
{
    int door_pos;
    public:
    Door();
    ~Door();
    void Config(int, int, int, int);
    void Start(char*45);
    void End();
    void Open_Door();
    void Close_Door();
};
#endif
```

Figura 4.7 — Exemplo de cabeçalho AO/C++ para "publisher/subscriber" (*Door.ph*)

4.6.3.1 Código em AO/C++

Um código de uma aplicação em AO/C++ que utilize o protocolo "publisher/subscriber" deve ter um formato significativamente diferente daquele gerado pelo SIMOO-RT baseado em chamadas remotas de método. Um código em AO/C++ corresponde ao arquivo *<Classe>.pC* onde está descrita a funcionalidade da classe com os respectivos métodos e ao arquivo *<Classe>.ph* estão as declarações de dados e métodos.

De acordo com este modelo, uma classe *Door* (retirada do estudo de caso) teria um código AO/C++ como o da figura 4.7 para o cabeçalho e o da figura 4.8 para o código de execução. Assim, pode-se ver que no cabeçalho a classe está declarada como classe ativa e que os métodos declarados nela são:

- construtor da classe: configura os eventos com um valor *default;*
- destrutor da classe;
- *Config()*: configura os eventos;
- *Start()*: assinatura dos eventos e código de inicialização do usuário;
- *End()*: finalizador da classe;
- métodos do usuário: definem a funcionalidade da classe.

Inicialmente, quando a instância é criada, os eventos a serem assinados e publicados recebem um valor *default* que está declarado no construtor da classe. O método *Config()*, presente em cada classe e chamado logo após a criação de uma instância, foi especialmente criado para suportar ao mesmo tempo a criação de instâncias e o protocolo "publisher/subscriber". Este método é responsável por, antes do objeto executar algum método do usuário, configurar cada evento assinado e publicado pela instância de modo que os eventos corretos sejam transmitidos (ex.: "PORTA_FECHADA_1" para a *porta_1* e "PORTA_FECHADA_2" para a *porta_2*) e recebidos (ex.: "FECHAR_PORTA_1" para o *porta_1* e "FECHAR_PORTA_2" para a *porta_2*).

```
#include "Door.h"
Door::Door( )
{
        OPEN_DOOR = 4;
        CLOSE_DOOR = 5;
        DOOR_OPENED = 6;
        DOOR_CLOSED = 7;
}
Door::~Door( )
{
}
void DoorProc::Config(int event1, int event2, int event3, int event4)
{
        // subscribed events
        OPEN_DOOR = event1;
        CLOSE_DOOR = event2;
        // published events
        DOOR_OPENED = event3;
        DOOR_CLOSED = event4;
}
void DoorProc::Start(char *name)
{
        subscribe("OPEN_DOOR", OPEN_DOOR, name);
        subscribe("CLOSE_DOOR", CLOSE_DOOR, name);
        //**** User Code ********************************
        door_pos=0;
}
//-----------------------------------------
void DoorProc::End()
{
}
//-----------------------------------------
void DoorProc::Open_Door()
{
    while (door_pos>0) door_pos--;
    publish("DOOR_OPENED", DOOR_OPENED);
    cout << "door: porta aberta" << endl;
}
//-----------------------------------------
void DoorProc::Close_Door()
{
    while (door_pos<10) door_pos++;
    publish("DOOR_CLOSED", DOOR_CLOSED);
    cout << "door: porta fechada" << endl;
}
```

Figura 4.8 — Exemplo de código AO/C++ para “publisher/subscriber” (*Door.pC*)

A chamada do método *Config()* e a conseqüente configuração da instância é feita pelo disparador do nó que configura os eventos assinados e os publicados por cada objeto. Além disso, os eventos do tipo "<classe>_CONFIG", "<classe>_START" e "<classe>_END" recebem valores fixos para todas as instâncias de todos objetos já que estes não são assinados mas sim ativados diretamente pelo disparador do nó.

Este esquema de configuração permite que em casos mais complexos como, por exemplo, em um sistema com dois controladores de temperatura, cada um com dois sensores, os eventos assinados e publicados sejam corretamente associados. Dessa maneira, o controlador 1 pode ser configurado para receber as temperaturas dos sensores 1 e 2 e controlador 2 as temperaturas dos sensores 3 e 4. Sendo feita de maneira externa à cada classe, essa configuração preserva a generalidade do código de cada classe excluindo, assim, a necessidade de um código especial para cada instância.

A assinatura dos eventos, por sua vez, é realizada no método *Start()* de cada classe e consiste basicamente em assinar os eventos previamente configurados através da chamada do método *Config()*. Por último, de acordo com exemplo dado, tem-se o código de execução da classe composto pelos métodos *Open_Door()* e *Close_Door()*. O primeiro método, *Open_Door()*, é responsável por "abrir" a porta controlada pelo objeto e em seguida publicar um evento dizendo que a porta está aberta ("DOOR_CLOSED"). De forma semelhante, o segundo método *Close_Door()* "fecha" a porta e publica um evento dizendo que ela está fechada ("DOOR_OPENED").

Como pode-se ver, a maior parte do trabalho é realizada pela infra-estrutura de comunicação através do uso das três funções criadas. Isso permite que o programador se preocupe somente com a funcionalidade da classe e com os eventos que devem ser publicados e assinados, e não de que maneira eles devem ser transmitidos e recebidos. Adicionalmente, não há preocupação em determinar quem irá receber as mensagens com os eventos através de, por exemplo, uma indicação do nome do objeto de destino e do nó em que ele se encontra.

4.6.3.2 Disparo da Aplicação

O disparo da aplicação consiste do processo de criar e configurar os objetos que compõe uma aplicação em seus respectivos nós. O disparador único para toda a aplicação que é atualmente criado pelo gerador de código do SIMOO-RT teve de ser abandonado já que o ambiente *runtime* do AO/C++ para Linux utiliza-se de funções baseadas em TCP/IP para criar objetos remotamente. De acordo com esse método o código da aplicação é instalado somente em uma máquina e, durante o disparo da mesma, os objetos são criados em diferentes nós, de acordo com uma configuração prévia.

No caso da arquitetura desenvolvida este procedimento não é mais possível pois as funções do ambiente de execução do AO/C++ utilizadas para instanciamento não funcionam no barramento CAN. Além disso, é desejável que cada nó da aplicação funcione de maneira independente e autônoma com relação à criação e gerenciamento dos objetos. Assim, caso durante a inicialização de uma aplicação um dos nós esteja com defeito os demais não são afetados, pelo menos com relação à criação dos objetos.

```
#define EVENTS_LIST CONFIG,START,END,\
        OPEN_DOOR_1,CLOSE_DOOR_1,DOOR_OPENED_1,DOOR_CLOSED_1,\
        OPEN_DOOR_2,CLOSE_DOOR_2,DOOR_OPENED_2,DOOR_CLOSED_2
```

Figura 4.9 — Exemplo de lista de eventos

```
//   File Name: Node_1.C
#include "Door.h"
#include "Heading.h"
enum{EVENTS_LIST};
void main(void)
{
        Door door_1, door_2;
        char *name;
        name = "/door_1";
        door_1.create(name);
        cout << endl << "Objeto " << name << " criado!" << endl;
        door_1.Config(OPEN_DOOR_1, CLOSE_DOOR_1, DOOR_OPENED_1, DOOR_CLOSED_1);
        cout << endl << "Objeto " << name << " configurado!" << endl;
        name = "/door_2";
        door_2.create(name);
        cout << endl << "Objeto " << name << " criado!" << endl;
        door_2.Config(OPEN_DOOR_2, CLOSE_DOOR_2, DOOR_OPENED_2, DOOR_CLOSED_2);
        cout << endl << "Objeto " << name << " configurado!" << endl;
        // Start objects
        name = "/door_1";
        cout << getpid() << ": Chamando door_1.Start()" << endl;
        door_1.Start(name);
        name = "/door_2";
        cout << getpid() << ": Chamando door_2.Start()" << endl;
        door_2.Start(name);
        while(getch() != 27)
                delay(500);
        // Stop objects
        door_1.End( );
        door_2.End( );
        door_1.stop( );
        door_2.stop( );
}
```

Figura 4.10 — Exemplo de disparador de aplicação

A forma de disparo da aplicação desenvolvida para a presente arquitetura foi utilização de um processo chamado *Node_ <X>* (ex.: figura 4.10) criado na forma de um para cada nó da aplicação (ex.: *Node_ 1*, *Node_ 2*) que é responsável por criar e inicializar localmente todos os objetos relativos ao nó, além de realizar a configuração dos eventos. Além disso, para suportar a configuração das instâncias criou-se um arquivo de cabeçalho contendo uma lista de eventos de toda a aplicação. Esta lista é necessária para que cada disparador possa configurar cada uma das instância com os códigos corretos dos eventos. O motivo que levou à criação de uma lista global de eventos de toda a aplicação foi a necessidade de designar um código identificador único para cada evento ainda em tempo de compilação, ou seja, antes da execução da aplicação. Esta mesma lista poderia ser gerada automaticamente pelo gerador de código do SIMOO-RT assim como os disparadores dos nós que poderiam ser gerados com a ajuda do diagrama de instâncias e do "deployment diagram".

Como pode-se ver na lista de eventos da figura 4.9, tem-se uma lista de eventos de uma aplicação com duas instâncias da classe *door*. Estas duas instâncias são criadas e configuradas pelo disparador do nó 1 que, através da chamada do método *Config()*, informa a cada uma delas o conjunto de eventos que deve ser utilizado.

```
.SUFFIXES: .ph .pC .C .cc
AO = ../../..
INC = $(AO)/include
LIB = $(AO)/lib
CFLAGS = -c -O2 -I$(INC)
LFLAGS = -L$(LIB) -laocc
CC = g++
LD = g++
LIBNETWORK = $(AO)/network/libnet.a
LIBCANISA = $(AO)/CanBus_Kaiser/lib/canlib.o
.C.o:
	$(CC) $(CFLAGS)  $<
.cc.o:
	$(CC) $(CFLAGS) -I$(AO)/network -I$(AO)/CanBus_Kaiser/include $? -o $@
all: Node_1 ech_1 DoorMain
##### ECH Node 1 #####
ech_1: ech.o $(LIBNETWORK) $(LIBCANISA)
	$(LD) $? $(LFLAGS) -o $@
##### Node 1 starting process #####
Node_1: AO.o Node_1.o Elevator.o Keybd.o Door.o Motor.o
	$(LD) $? $(LFLAGS) -o $@
##### Dependencies class: DOOR #####
DOOR_OBJ = AO.o DoorMain.o DoorProc.o Door.o
DoorMain: $(DOOR_OBJ)
	$(LD) $? $(LFLAGS) -o $@
clean:
	rm -f *.o *Main core
```

Figura 4.11 — Exemplo de *Makefile*

4.6.3.3 Makefile

Para facilitar o processo de compilação da aplicação para a arquitetura desenvolvida é utilizado um *Makefile* bastante diferente do normalmente gerado pelo SIMOO-RT (ver fig. 4.11). A utilização de um compilador específico para o Motorola 68332, a forma de geração do arquivo para o *download* na placa e a estrutura diferenciada do protocolo "publisher/subscriber" implica em se escrever um *Makefile* específico para cada um dos nós de uma aplicação que gere automaticamente todos os arquivos executáveis necessários. Para criar um novo *Makefile* é necessário que se siga o modelo mostrado nas aplicações de exemplo criadas neste trabalho.

4.6.4 Protocolo "publisher/subscriber" para PC

Após o desenvolvimento do suporte "publisher/subscriber" para μClinux os *daemons* e as funções que o compõe foram adicionadas à biblioteca AO/C++ padrão do Linux criando-se, assim, uma biblioteca específica para uso em PC's. Isto tornou possível a criação de aplicações com comunicação "publisher/subscriber" para computadores *desktop* assim como a utilização de sistemas mistos baseados em Linux e μClinux. O suporte para "publisher/subscriber" para Linux foi desenvolvido para uma placa CAN-ISA, a qual possui dois controladores 82527 da Intel, disponível no Laboratório de Automação. Para isso os *device drivers* da CAN-ISA disponíveis foram atualizados do núcleo 2.0.X para o 2.2.X e configurados para endereços e para um IRQ apropriado. Além disso, uma classe com rotinas específicas que configuram adequadamente a CAN-ISA foram criadas para o ECH.

5 Estudo de Caso

5.1 Sistema de Controle Multi-Elevador

O estudo de caso escolhido para testar a arquitetura desenvolvida foi um sistema de controle multi-elevador baseado em um trabalho previamente realizado no Laboratório de Automação deste departamento [BRU 98]. Este sistema foi escolhido por adequar-se bem à uma arquitetura de baixo custo baseada em uma rede de placas embarcadas e no conceito de objetos distribuídos. O número de placas a ser utilizado pode ser, por exemplo, de uma para cada elevador existente ou mais apropriadamente de acordo com o número de objetos existentes, capacidade de processamento requerida e custo total.

5.1.1 Modelo do Sistema

O sistema de controle multi-elevador foi inicialmente criado com o auxílio do SIMOO-RT onde foi criada uma estrutura de classes e instâncias visando um sistema capaz de gerenciar um sistema com mais de um elevador. Num primeiro nível (vide fig. 5.1), de acordo com a metodologia do SIMOO-RT, o sistema de elevadores foi definido por uma única classe *Multielev* que agrega todas as demais.

O segundo nível (fig. 5.2) apresenta um detalhamento do anterior, com o diagrama de classes sendo composto pelo processador dos botões de chamada (classe *Button*), um gerenciador (classe *Manager*) e um elevador (classe *Elevator*). O diagrama de instâncias, correspondente a esse nível, é mostrado ao lado como sendo formado por um gerenciador, um processador de botões e dois elevadores. Um número maior de elevadores poderia ser adicionado, no SIMOO-RT, simplesmente adicionando-se instâncias no respectivo diagrama. O terceiro nível (fig. 5.3), por sua vez, detalha a classe elevador através de um diagrama de classes composto de um motor (classe *Motor*), de um teclado interno (classe *Keybd*) e de uma porta (classe *Door*) com uma instância de cada no diagrama de instâncias.

5.1.1.1 Gerenciador (classe *Manager*)

Esta classe, mostrada na figura 5.2, é responsável por decidir qual elevador irá atender cada nova requisição feita pelos botões. O gerenciador recebe mensagens dos elevadores, através de uma assinatura prévia, contendo o estado atual de cada um deles, incluindo o andar atual e direção de deslocamento, e armazena-os em uma lista interna. A partir desta lista com o estado de cada elevador o gerenciador aplica uma determinada estratégia de decisão e transmite ao elevador selecionado o andar que deve ser atingido.

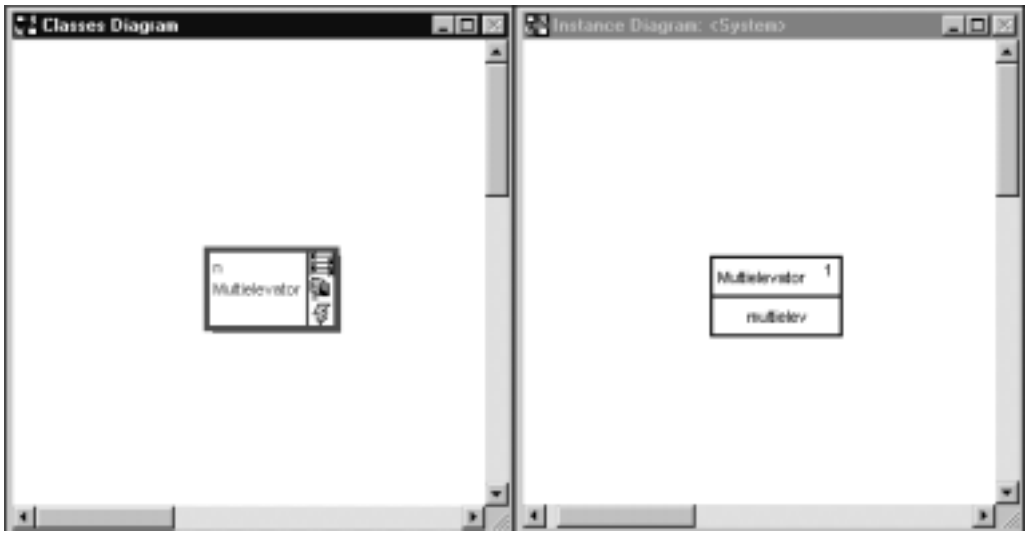

Figura 5.1 — Primeiro nível do modelo do elevador

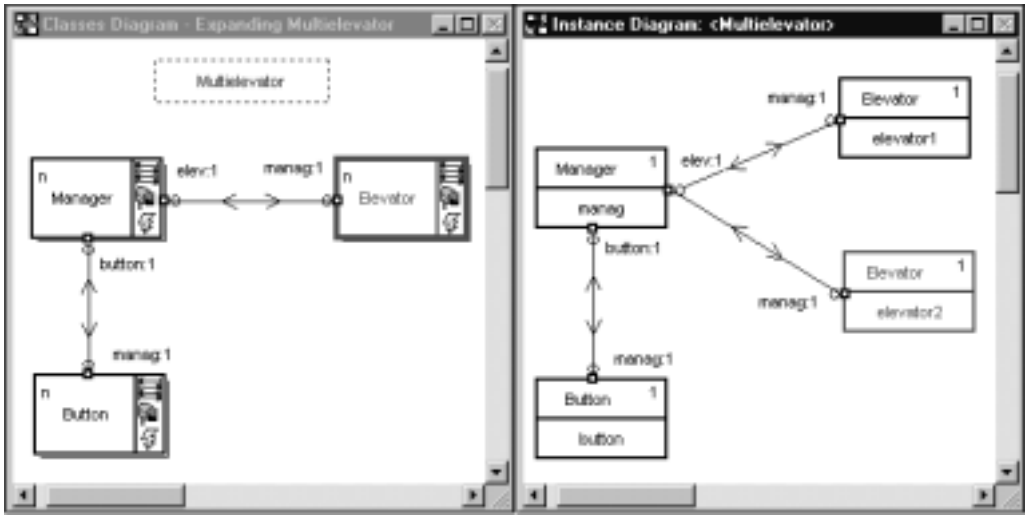

Figura 5.2 — Segundo nível do modelo do elevador

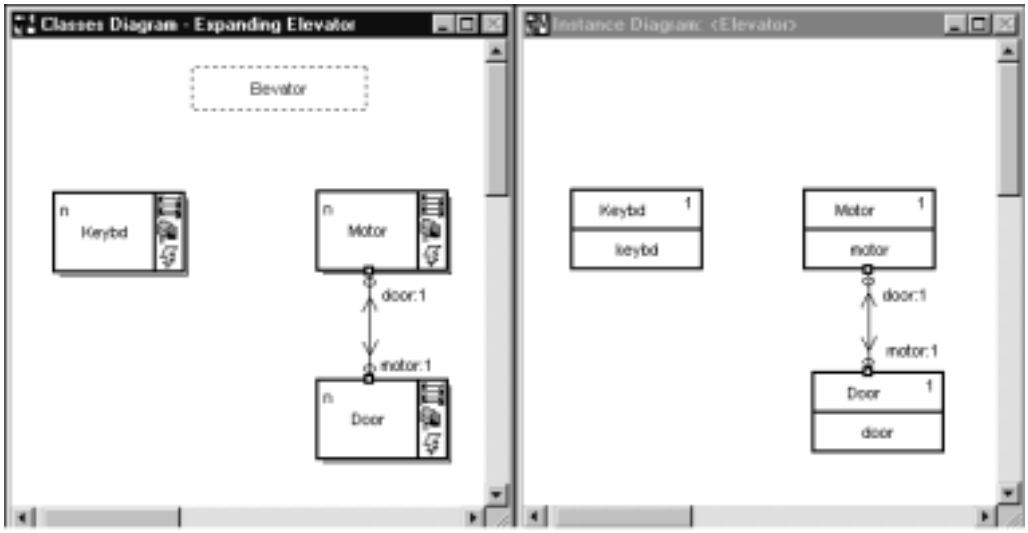

Figura 5.3 — Terceiro nível do modelo do elevador

O gerenciador, na verdade, faz uma centralização do despacho das requisições dos botões, o que simplifica a criação do sistema de controle. Uma forma alternativa possível seria fazer com que os próprios elevadores negociassem entre si e decidissem qual deles iria atender a cada requisição recebida. Dessa maneira, todos eles receberiam as requisições e informariam se poderiam atendê-las, talvez dizendo quanto tempo precisariam para atender. Assim, o elevador com o menor tempo de atendimento seria automaticamente escolhido. Um problema desta estratégia é como garantir que sempre um dos elevadores se responsabilizará por atender à solicitação e que, de preferência, somente um irá atendê-la.

5.1.1.2 Botão (classe *Button*)

Esta classe (fig. 5.2) controla os botões externos de chamada do elevador que ficam no corredor dos prédios. As chamadas são publicadas na forma de uma mensagem indicando o andar do botão e a direção solicitada que pode ser para cima ou para baixo.

5.1.1.3 Elevador (classe *Elevator*)

A classe *Elevator* (fig. 5.2) agrega um conjunto de funções que permite controlar os objetos constituintes de cada elevador individualmente. Recebe requisições do teclado e do gerenciador e mantém uma lista interna de requisições a serem atendidas. Além disso, seleciona qual das requisições existentes na lista será atendida primeiro e remove da lista as requisições já atendidas. Mantém também um registro com o estado atual do elevador formado a partir das informações transmitidas pela porta e pelo motor.

Pode-se ver na figura 5.4 uma relação de métodos no campo "Methods", as respectivas mensagens de ativação no campo "Activating Messages" e, no campo "Method Code" o código do método *Select_Floor()*. As mensagens de ativação foram

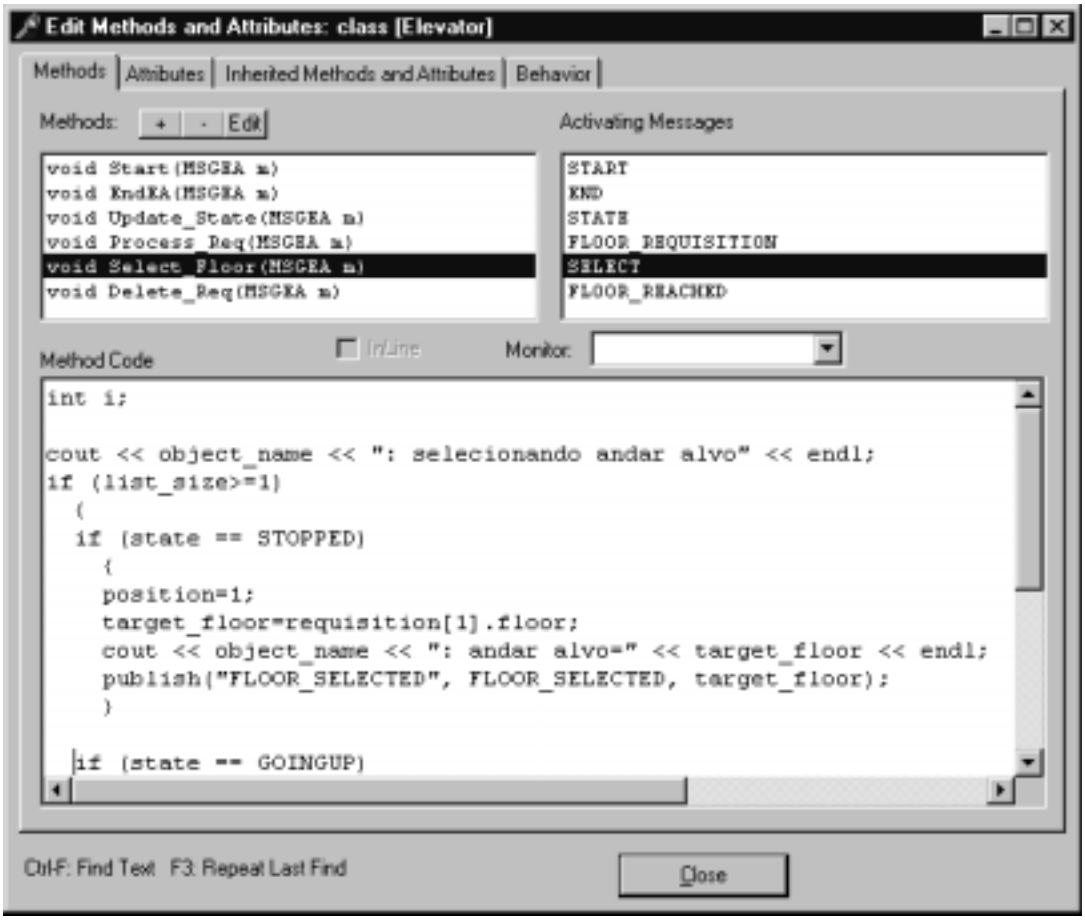

Figura 5.4 — Janela de edição de métodos da classe *Elevator*

definidas de acordo com o conceito de eventos do protocolo "publisher/subscriber" facilitando a implementação da aplicação numa fase posterior.

5.1.1.4 Motor (classe *Motor*)

Esta classe (fig. 5.3) é responsável pelo controle do motor do elevador e pela execução e sincronização da seqüência de operações necessárias para a movimentação e parada do elevador. É também responsável por enviar os comandos para a abertura e fechamento da porta constituindo-se basicamente em uma máquina de estados que sincroniza a porta e o motor.

5.1.1.5 Porta (classe *Door*)

A classe porta (fig. 5.3) executa basicamente as operações de abertura e fechamento da porta publicando uma mensagem sempre que uma destas operações é finalizada.

5.1.1.6 Teclado (classe *Keybd*)

Esta classe, mostrada na fig. 5.3, corresponde ao teclado interno do elevador que contém os números de todos os andares. Cada vez que um andar é selecionado uma requisição é enviada diretamente para a classe *Elevator*.

5.1.1.7 Mensagens de Ativação das Classes

De acordo com a metodologia do SIMOO-RT para cada classe existe uma lista de métodos com a respectiva mensagem de ativação (ver fig. 5.1). Como o protocolo de simulação utilizado é ponto-a-ponto, não há necessidade de distinção entre as mensagens de ativação de cada instância. Assim, como no protocolo "publisher/subscriber" não são utilizadas mensagens de ativação mas sim eventos, os métodos e as mensagens de ativação definidas no SIMOO-RT foram utilizadas somente como base para a implementação feita a seguir.

5.1.2 Implementação

A fase posterior à criação do modelo orientado a objetos constituiu-se da geração do código da aplicação para o Linux, de acordo com o procedimento mostrado na figura 3.3. A isso seguiu-se o processo de porte para o μClinux e para o protocolo "publisher/subscriber" baseado nas modificação descritas no item 4.6.3. Todas as modificações necessárias no código dos objetos citadas anteriormente foram efetuadas manualmente em cada um dos arquivos constituintes da aplicação, e em seguida a aplicação foi compilada e testada na forma de duas diferentes versões do sistema, as quais são descritas a seguir.

A primeira versão desenvolvida (ver fig. 5.5) constitui-se de único elevador rodando na placa MEGA332 e de um ECB rodando num PC com Sistema Operacional Linux. Este sistema, por tratar-se de um teste inicial, era constituído de somente uma instância de cada objeto. A execução desta aplicação assim como de qualquer outra que use a arquitetura desenvolvida, pode ser dividida em três fases:

1. instanciação e inicialização dos objetos;
2. "assinatura" dos eventos;
3. interação entre os objetos.

Durante a fase de instanciação dos objetos os mesmos são criados pelo processo disparador do nó e recebem nomes tais como: *elevator_1* e *door_1*. Durante a criação destes objetos os eventos utilizados por cada um deles são inicializados com valores *default* os quais, logo em seguida, são adequadamente configurados pelo disparador do nó. Isto é feito através da invocação do método de configuração *Config()*, o qual foi adicionado durante a fase de implementação do sistema com um elevador. A última operação realizada pelo disparador consiste em invocar o método *Start()* de cada objeto que é responsável pela assinatura dos eventos, de acordo com a configuração previamente realizada.

Como vê-se na tabela 5.2, os eventos assinados pelo objeto *elevator_1*, por exemplo, receberam a adição de um índice que tem a função de diferenciar o conjunto

Classe	**Método**	**Mensagem de ativação**
Manager		
	Start	MANAGER_START
	End	MANAGER_END
	Process_Button	BUTTON_REQ
	Update_State	STATE
	Attach_to_list	ATTACH
Button		
	Start	BUTTON_START
	End	BUTTON_END
	Send_Req	BUTTON_SEND
Elevator		
	Start	ELEVATOR_START
	End	ELEVATOR_END
	Update_State	STATE
	Process_Req	KEYBD_REQ
	Select_Floor	SELECT
	Delete_Req	FLOOR_REACHED
Motor		
	Start	MOTOR_START
	End	MOTOR_END
	Go	DOOR_CLOSED
	Update_Floor	PASSING_FLOOR
	Activate	FLOOR_SELECTED
	Up	MOTOR_UP
	Down	MOTOR_DOWN
	Stop	MOTOR_STOP
	Move	MOTOR_MOVE
	Deactivate	DOOR_OPENED
Door		
	Start	DOOR_START
	End	START_END
	Open_Door	OPEN_DOOR
	Close_Door	CLOSE_DOOR
Keybd		
	Start	KEYBD_START
	End	KEYBD_END
	Send_Req	KEYBD_SEND

Tabela 5.1 — Lista de eventos do Sistema de Controle Multi-Elevador

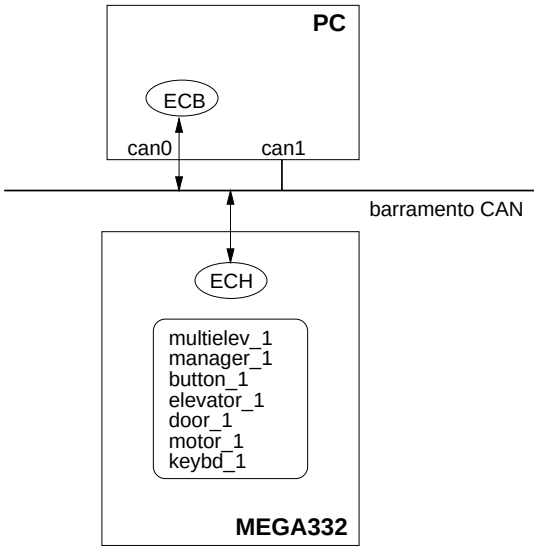

Figura 5.5 — Primeira implementação do elevador

Método	Evento assinado no *elevator_1*
Update_State	STATE_1
Process_Req	KEYBD_REQ_1
Select_Floor	SELECT_1
Delete_Req	FLOOR_REACHED_1

Tabela 5.2 — Relação de eventos e métodos ativados do objeto *elevator_1*

de eventos assinados por cada instância. Este mesmo artifício também é utilizado para os eventos publicados, de modo que praticamente qualquer tipo de relação entre instâncias pode ser convenientemente configurada. Por último, vê-se que os métodos *Start()*, *End()* e *Config()* não constam da tabela, já que eles são diretamente invocados pelo disparador do nó e, portanto, não precisam ser assinados.

Utilizando-se o diagrama de seqüências da figura 5.6 pode-se visualizar melhor a interação entre os objetos durante a execução da aplicação. Assim, vê-se que a seleção de um andar por um usuário dentro do elevador faz com que o teclado (*keybd_1*) publique um evento dizendo que uma requisição foi feita. O elevador (*elevator_1*) recebe então este evento e, verificando que em sua lista de requisições não há mais nenhuma requisição, publica um evento dizendo que o andar selecionado deve ser atingido. O motor (*motor_1*), por sua vez, recebe esta mensagem e inicia a seqüência de movimentação publicando um comando para que a porta se feche e, feito isso, inicia o deslocamento. Quando o andar alvo é atingido, o próprio motor (*motor_1*) inicia a parada e após publica um comando para que a porta se abra. Finalmente, após receber um evento dizendo que a porta está aberta, o motor (*motor_1*) publica um evento dizendo que o andar foi alcançado e então a seqüência é encerrada.

Na segunda versão do sistema de controle multi-elevador implementada foi utilizada uma distribuição dos objetos da aplicação de maneira que o teclado e os botões do elevador foram colocados no PC e os demais objetos na MEGA332 (vide fig. 5.7). O PC, possuindo uma placa CAN-ISA com dois controladores CAN foi

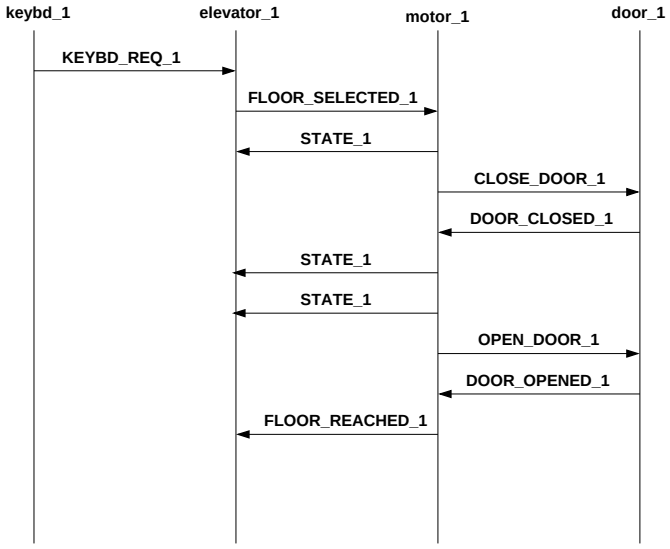

Figura 5.6 — Diagrama de seqüências

utilizado de maneira que o ECB pode se comunicar através do controlador número zero e o ECH através do número um. Além disso, no PC foram utilizadas versões do ECH e ECB para Linux enquanto que na MEGA332 foram, naturalmente, utilizadas as versões para μClinux.

A distribuição do código da aplicação para esta versão consistiu basicamente em modificar o processo disparador do nó da MEGA332, através da retirada das instruções que criavam as instâncias do teclado e dos botões e da modificação do *makefile* do nó de maneira que eles não fossem compilados. Além disso, foi criado um processo disparador para o PC, através da inclusão de instruções para a criação das instâncias, e um *makefile* apropriado para o nó. Apesar disso, não foi necessária nenhuma modificação no código das classes que compõem a aplicação já que a criação dos objetos é controlada "externamente" em relação aos objetos, pelo disparador do nó.

A disponibilidade de somente uma placa MEGA332 e uma CAN-ISA para PC impossibilitou a implementação de um sistema com maior número de elevadores. No entanto, a distribuição da aplicação em um número maior de nós é tarefa relativamente fácil já que consiste somente em se modificar e/ou criar novos disparadores para os nós e os respectivos *makefiles*. Adicionalmente, a criação de mais instâncias requer também que se adicione os nomes dos novos eventos na lista geral de eventos.

No caso hipotético de um sistema com dois elevadores, tería-se um esquema como o da tabela 5.3, onde o elevador 1 estaria associado ao teclado 1, ao motor 1 e à porta 1. Da mesma maneira, o elevador 2 estaria associado ao teclado 2, ao motor 2 e à porta 2. Além disso, os dois elevadores se comunicariam com o gerenciador que, neste caso, seria chamado de gerenciador 1. O gerenciador, por sua vez, assinaria os eventos "STATE_1 " relativo ao elevador 1 e "STATE_2" relativo ao elevador 2 recebendo-os ao mesmo tempo que os objetos *elevator_1* e *elevator_2*.

Já no caso de um protocolo de comunicação do tipo ponto a ponto seriam necessárias quatro mensagens: o evento "STATE_1" precisaria ser transmitido

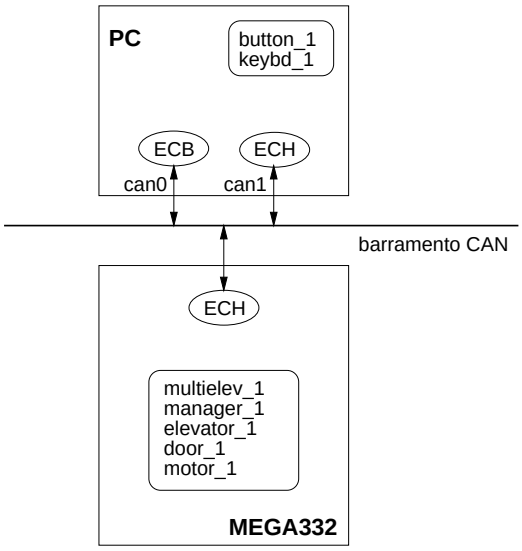

Figura 5.7 — Implementação do elevador com distribuição de objetos

Método	Evento no *elevator_1*	Evento no *elevator_2*
Update_State	STATE_1	STATE_2
Process_Req	KEYBD_REQ_1	KEYBD_REQ_2
Select_Floor	SELECT_1	SELECT_2
Delete_Req	FLOOR_REACHED_1	FLOOR_REACHED_2

Tabela 5.3 — Eventos assinados e métodos nas instâncias da classe *Elevator*

uma vez ao *elevador_1* e outra ao gerenciador e, de maneira semelhante, o evento "STATE_2" teria de ser transmitido uma vez ao *elevador_2* e outra ao gerenciador.

Com este pequeno exemplo pode-se ver claramente que a utilização do protocolo "publisher/subscriber" pode trazer vantagens, entre elas a economia de tempo, para sistemas onde o mesmo dado precisa ser transmitido a uma série de objetos. Esta característica também pode ser aproveitada por sistemas de monitoração e supervisão, os quais podem assinar os eventos que precisam ser monitorados sem, contudo, interferir na operação do sistema.

Com respeito à distribuição deste sistema, poderia ser utilizada uma série de placas microcontroladas, como mostra o exemplo da figura 5.8. Neste exemplo tem-se dois elevadores, um na placa 1 e outro na placa 2, onde estão os objetos que compõe cada um deles. O objeto *multielevator*, o gerenciador e os botões foram colocados na placa 3 e, além disso, o ECB localiza-se na placa 4. Dessa maneira, em um sistema multi-elevador real, podería-se ter uma placa embarcada para cada um dos elevadores do conjunto, uma placa para o ECB e pelo menos uma para os botões e os demais objetos.

Como se pode ver na placa 4, o ECB é o único elemento existente já que, de acordo com a implementação da arquitetura proposta realizada, é necessário que o ECB possua um controlador CAN exclusivo para ele. De outra maneira, o compartilhamento de um controlador CAN com um ECH, por exemplo, poderia causar atrasos nas respostas do ECB às requisições de configuração e ligação já que este seria responsável por todos os nós do barramento CAN.

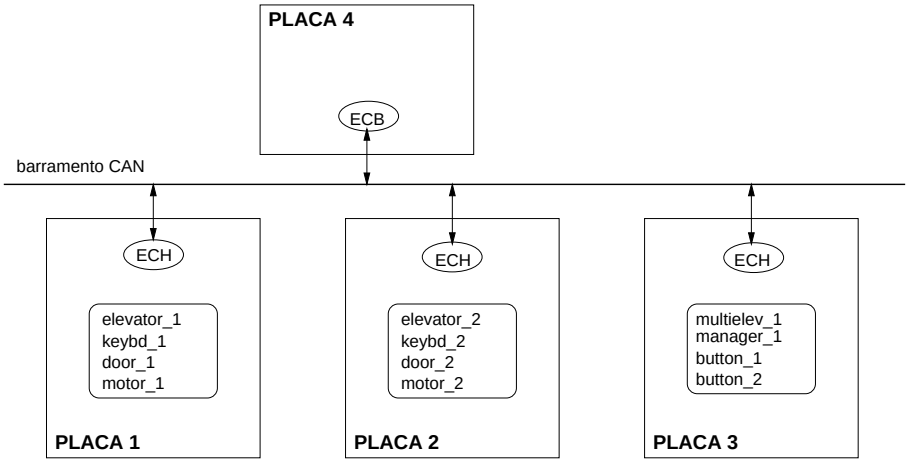

Figura 5.8 — Configuração sugerida para o sistema multi-elevador

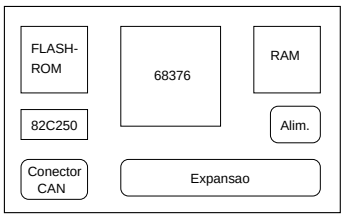

Figura 5.9 — Placa sugerida

5.2 Experiências Obtidas

A partir da experiência adquirida no estudo de caso com a placa MEGA332 chegou-se à definição de uma placa mais apropriada para o desenvolvimento de sistemas de automação baseados em objetos distribuídos e no barramento CAN. A placa sugerida (fig. 5.9) é composta basicamente por:

- Microcontrolador Motorola 68376
- 2 Mb RAM
- 2Mb FLASH-ROM
- *Transceiver*
- Conectores para: serial, CAN e expansão

O microcontrolador 68376 foi escolhido porque já possui um controlador CAN embutido e porque sendo semelhante ao 68332 possibilita o utilização do μClinux. Os 2Mb de RAM e a FLASH-ROM permitem a criação de aplicações maiores e mais complexas e também a utilização de maior suporte do sistema operacional tal como programas de interfaceamento e módulos. Eventualmente, com a redução de custos das memórias pode-se utilizar uma quantidade ainda maior, dependendo também das necessidades.

6 Conclusões

6.1 Conclusões

De modo geral o foco do trabalho foi na integração dos elementos necessários para a criação e implementação de sistemas de automação baseados em objetos distribuídos e no barramento CAN. Esses elementos, de acordo com a arquitetura proposta, constituem-se de placas microcontroladas para a aplicação e computadores (PC's) para supervisão, sistema operacional, barramento industrial para comunicações e orientação a objetos para modelagem e criação dos sistemas de controle. O trabalho concentrou-se basicamente no preenchimento das lacunas existentes entre os elementos, principalmente através do desenvolvimento de *software*. Os elementos propriamente desenvolvidos constituiram-se da versão do Sistema Operacional μClinux para a placa microcontrolada MEGA332, do programa de interface (*device driver*) para o controlador CAN SJA1000, de adaptações na biblioteca C do μClinux para compatibilização com C++, do porte do AO/C++ do Linux para o μClinux e, finalmente, do suporte para a comunicação entre os objetos através de protocolo "publisher/subscriber".

A utilização de objetos distribuídos para o desenvolvimento de aplicações em automação mostrou ser uma alternativa viável e eficiente. A infra-estrutura criada suporta a distribuição dos objetos em tantos nós quanto forem necessários e fornece também os meios para a comunicação entre eles, tanto dentro do mesmo nó quanto entre nós diferentes, através do barramento CAN. Além disso, é possível criar diversas instâncias dos objetos e, através de uma configuração adequada, diferenciar os eventos publicados e assinados por cada uma delas a fim de manter a compatibilidade com o protocolo "publisher/subscriber". Com isso, a arquitetura proposta mostrou-se adequada ao desenvolvimento de sistemas de automação além de ser suficientemente flexível para suportar tanto a utilização de objetos em placas microcontroladas quanto em PC's.

O abandono do uso de RMI (Remote Method Invocation), suportado pelo AO/C++, trouxe algumas vantagens e desvantagens para a arquitetura desenvolvida. Em primeiro lugar, não é mais possível a um objeto ativar diretamente um método de outro objeto, tanto local quanto remotamente. Em lugar disso, os objetos publicam mensagens com eventos que podem ser assinados por outros objetos para recepção. Dessa forma, a comunicação é sempre anônima e assíncrona, o que significa também que o modelo cliente/servidor, geralmente utilizado pelos programas em AO/C++, não é mais suportado. Isso se reflete no momento da criação de um sistema de automação, já que o mesmo deve ser concebido desde o início para a utilização de comunicação anônima e assíncrona. Cabe então ao projetista decidir quais os eventos que devem ser publicados e quais os eventos que precisam se assi-

nados por cada objeto. Dessa maneira, quando mais de um objeto deseja receber o mesmo evento só é necessário se modificar os objetos "receptores" o que não ocorre quando se usa RMI.

Com relação ao Sistema Operacional μClinux pode-se constatar que o mesmo é um sistema operacional que oferece muitos recursos e ao mesmo tempo é suficientemente simples para utilização em automação embarcada e outras aplicações. Seu processo de *boot* é bastante rápido em comparação com plataformas baseadas em disco e é capaz de gerenciar uma quantidade de processos concorrentes em número suficiente de forma segura e eficiente. Uma outra vantagem importante do μClinux, em relação aos sistemas operacionais embarcados comerciais, é a sua característica de sistema aberto, que permite a realização de adaptações e pequenas melhorias de acordo com as necessidades do usuário.

Como dito anteriormente, o sistema operacional μClinux também pode ser utilizado em diferentes microcontroladores, tais como o Motorola Coldfire e o i960 da Intel, que são de maior capacidade de processamento, além daqueles da família Motorola 683XX. Assim, a solução desenvolvida tem a vantagem de ser portável, permitindo o uso de sistemas heterogêneos compostos de diferentes placas, cada uma adequada a um tipo de tarefa.

As aplicações obtidas pela modificação do código gerado pelo SIMOO-RT mostraram-se eficientes do ponto de vista do funcionamento. A utilização das funções *subscribe*, *unsubscribe* e *publish* é relativamente simples e facilita bastante o trabalho do programador, já que não há necessidade nenhuma de identificar o objeto de origem e nem o de destino. A publicação de eventos, periódica ou não, utilizada no protocolo "publisher/subscriber", evita a ocupação desnecessária do barramento com solicitações de informações e suas respectivas respostas que é a forma normalmente usada em sistemas mestre/escravo. Assim, o modelo de comunicação "publisher/subscriber" permite um uso mais efetivo do barramento quando comparado ao modelo mestre-escravo.

6.2 Trabalhos Futuros

A inexistência de suporte para aplicações tempo-real é uma deficiência que poderia ser atacada em trabalhos futuros, já que Sistema Operacional μClinux não oferece este tipo de suporte que muitas das aplicações de automação industrial e embarcadas apresentam. Apesar de possuir pontos positivos tais como a inexistência de *swap* e tamanho reduzido do núcleo, o sistema operacional utilizado não oferece garantias de tempo-real. Assim, uma melhoria do desempenho tempo-real do μClinux poderia ser conseguida com a utilização de técnicas tais como as apresentadas nos projetos RED-Linux [WAN 99] ou RT-Linux [RTL 00].

Como segunda sugestão de trabalho fica o desenvolvimento de maior suporte para a criação das aplicações. Como dito antes, a modificação do código gerado pelo SIMOO-RT para a arquitetura desenvolvida constitui-se em um trabalho demorado e sujeito à erros. Da mesma maneira, a criação manual de uma aplicação a partir do zero é uma tarefa complexa e sem a agilidade necessária. A sugestão apresentada é a da criação de um gerador de código específico para o SIMOO-RT que leve em consideração a arquitetura proposta bem como a criação de maior suporte para

a distribuição da aplicação. Um recurso desejável seria também a simulação da comunicação por *broadcast,* de forma semelhante aquela que ocorre no barramento CAN, dentro do ambiente SIMOO-RT, permitindo testar e refinar a aplicação ainda na fase de criação.

Referências Bibliográficas

[BEC 99a] BECKER, L. B. Ambiente de modelagem e implementação de sistemas tempo real usando o paradigma de orientação a objetos. Porto Alegre:CPGCC da UFRGS, 1999. Dissertação de Mestrado.

[BEC 99b] BECKER, L. B. Simoo-rt: An integrated object-oriented environment for the development of distributed real-time systems. In: 15TH ISPE/IEE INTERNATIONAL CONFERENCE ON CAD/CAM, ROBOTICS, AND FACTORIES OF THE FUTURE, CARS AND FOF'99, 1999. **Proceedings...** Campinas, SP:[s.n.], 1999. p.13–18.

[BLO 92] BLOOMER, J. **Power Programming With RPC**. O'Reilly & Associates, Inc., 1992.

[BOS 91] BOSCH, R. Can specification version 2.0. Published by Robert Bosh GmbH, 1991.

[BOU 96] BOURDON, G.; RAUX, P. Can for autonomous mobile robot. In: 3RD INTERNATIONAL CAN CONFERENCE, 1996. **Proceedings...** Paris, France:[s.n.], 1996.

[BRU 98] BRUDNA, C. Sistema para controle de elevador em tempo-real. In: II SEMINáRIO INTERNO DO DEPARTAMENTO DE ENGENHARIA ELéTRICA, 1998. **Proceedings...** Porto Alegre, Brasil:[s.n.], 1998. p.I21–24.

[CIA 94a] CIA. Can physical layer for industrial applications - cia ds102-1. Published by CAN in Automation, April, 1994.

[CIA 94b] CIA. Canopen communication profile for industrial systems. cia draft standard 301 version 3.0. Published by CAN in Automation, 1994.

[DAT 95] DATTOLO, J. J. Society of automotive engineers (sae) implementation of can for heavy duty truck and bus market - specification j1939. In: 2ND INTERNATIONAL CAN CONFERENCE, 1995. **Proceedings...** London, United Kingdom:[s.n.], 1995.

[ELK 00] ELKS. The embeddable linux kernel subset. URL: www.elks.soton.ac.uk, 2000.

[FRE 97] FREDRIKSON, L. A can kingdom (rev. 3.01). Published by KVASER AB, Box 4076, S-51104, Kinnault, Sweden, 1997.

[HON 96] HONEYWELL. Smart distributed systems, application layer protocol version 2, 1996. Micro Switch Specification GS 052 103 Issue 3.

[KAI 99] KAISER, J. Implementing the real-time publisher/subscriber model on the controller area network (can). In: 2ND INT. SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING (ISORC99), 1999. **Proceedings...** Saint Malo, France:[s.n.], 1999.

[KIR 96] KIRK, B.; FREI, M. The unicontrol hardware, software and system concept. In: 3RD INTERNATIONAL CAN CONFERENCE, 1996. **Proceedings...** Paris, France:[s.n.], 1996.

[KOO 96] KOOPMAN, P. Embedded system design issues (the rest of the story). In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN, 1996. [s.n.], 1996.

[LIN 00a] LINEO. Embedix sdk, 2000. URL: www.lineo.com.

[LIN 00b] LINUXCE. Linux ce. URL: www.linuxce.org, 2000.

[LTD 00] LTDA, Q. S. S. Qnx neutrino. URL: www.qnx.com, 2000.

[MIC 00] MICROSOFT. Windows ce. URL: www.microsoft.com, 2000.

[NOO 94] NOONEN, D.; SIEGEL, S. Devicenet applicattion protocol. In: 1ST. INTERNATIONAL CAN CONFERENCE, 1994. **Proceedings...** Mainz, Germany:[s.n.], 1994.

[OMG 99] OMG. Real time corba. Published by OMG, march, 1999. OMG Document orbos /99-02-12.

[PER 94] PEREIRA, C. E. Real-time active objects in c++/real-time unix. In: ACM SIGPLAN WORKSHOP ON LANGUAGES, COMPILER AND TOOLS SUPPORT FOR REAL-TIME SYSTEMS, 1994. **Proceedings...** Orlando, USA:[s.n.], 1994.

[PER 98] PEREIRA, C. E. On the suitability of object orientation for real-time distributed systems development. In: WORKSHOP ON ARCHITECTURES AND ALGORITHMS FOR REAL-TIME CONTROL, 1998. **Proceedings...** Cancun, Mexico:[s.n.], 1998.

[POP 97] POPP, M. The rapid way to profibus-dp. PROFIBUS Nutzerorganization - PNO, 1997. Karslhue, Germany.

[RTL 00] RTLINUX. Real-time linux. URL: www.rtlinux.org, 2000.

[RUB 98] RUBINI, A. **Linux Device Drivers**. O'Reilly & Associates, Inc., first edition, 1998.

[RUF 97] RUFINO, J. An overview of the Controller Area Network. In: CIA CAN FORUM FOR NEWCOMERS, 1997. **Proceedings...** Braga, Portugal:[s.n.], 1997.

[RUF 99] RUFINO, J.; VERíSSIMO, P. Embedded platforms for distributed real-time computing: Challenges and results. In: 2ND INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING, 1999. **Proceedings...** Saint Malo, France:[s.n.], 1999. p.147–152.

[SOF 00] SOFTWARE, K. Rtx51, rtx251 and rtx166 operating systems. URL: www.keil.com, 2000.

[SRL 00] SRL, P. Etlinux. URL: www.prosa.it/etlinux, 2000.

[TAN 89] TANENBAUM, A. **Computer Networks**. Englewood Cliffs:Prentice-Hall, 1989.

[UCL 00] UCLINUX. The linux/microcontroller project. URL: www.uClinux.org, 2000.

[VIS 00] VISTA, M. Hard hat linux. URL: www.mvista.com, 2000.

[WAN 99] WANG, Y.-C. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In: 20TH REAL-TIME SYSTEMS SYMPOSIUM, 1999. **Proceedings...** Phoenix, Arizona, USA:[s.n.], 1999.

[WIL 99] WILD, R. A tool for validating timing requirements of industrial applications based on the foundation fieldbus protocol. In: 24TH IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, 1999. **Proceedings...** Schloss Dagstuhl, Germany:[s.n.], 1999.

[WOR 00] WORKS, L. The blue cat linux. URL: www.lynx.com, 2000.