Universidade Federal de Santa Catarina – UFSC
Departamento De Informática e Estatística – INE
Bacharelado em Ciências Da Computação

Suporte a redes CAN para Aplicações Embarcadas

Alessandro Barreiros Maurici

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Banca Examinadora:
Arliones Stevert Hoeller Junior
Lucas Francisco Wanner

Florianópolis
2005

Suporte a redes CAN para Aplicações Embarcadas

Alessandro Barreiros Maurici

Este Trabalho de Conclusão de Curso foi julgado adequado para a obtenção do título de Bacharel em Ciências da Computação, e aprovado em sua forma final pela Coordenadoria do Curso de Bacharelado em Ciências da Computação.

Prof. Dr. José Mazzucco Junior

Banca Examinadora

Prof. Dr. Antônio Augusto Medeiros Fröhlich

B. Sc. Arliones Stevert Hoeller Junior

B. Sc. Lucas Francisco Wanner

Dedico este trabalho a minha família, amigos e professores.

Resumo

A utilização de redes de comunicação para sistemas embutidos deixou de ser uma ferramenta opcional para tornar-se uma necessidade, estas redes são denominadas *Fieldbus.*

CAN, Controller Area Network, é uma especificação de interconexão e protocolo para comunicação. Redes CAN são relativamente antigas, porém, com a miniaturização e queda dos custos dos circuitos eletrônicos, o uso deste tipo de rede vem crescendo muito. Entretanto ainda enfrentamos o problema da implementação de software para sistemas embutidos, por exemplo, o pequeno ou nenhum reaproveitamento de código é um dos problemas enfrentados.

Um projeto de Sistema Operacional Orientado à Aplicação, como o EPOS, para sistemas deste gênero, minimiza o tempo e o custo para um desenvolvedor implementar um software, o qual deverá ou poderá rodar em diferentes plataformas.

Neste trabalho será exibida uma visão sobre CAN e do sistema operacional EPOS, também será dada uma introdução sobre a arquitetura básica do microcontrolador AVR, utilizado na implementação de um mediador, que possibilitará o uso de redes CAN no AVR utilizando o sistema EPOS.

Palavras-chave: Controller Area Network, CAN, Fieldbus, Sistemas Embutidos, AVR, EPOS, Sistema Operacional Embutido.

Abstract

Utilization of communication networks for embedded systems, gone from optional to a necessity, these networks type is called Fieldbus.

CAN, Controller Area Network, is a specification of interconnection and protocol for communication. CAN networks are relatively old, but, with the miniaturization and falling prices of electronics circuits, this kind of networks became growing higher. However we still face software implementation problems, as an example, the low or none reuse of code is one of the problems.

A project of Application-Oriented Operating System, like EPOS, for these kind of system, minimizes time and costs for a developer do implement a software, which may or have to run in mixed platforms.

This work will show an introduction of CAN and EPOS, also some explanation about AVR microcontroller basic architecture, this microcontroller is utilized in the EPOS mediator implementation, which will provide CAN access into AVR running EPOS system.

Keywords: Controller Area Network, CAN, Fieldbus, Embedded Systems, AVR, EPOS, Embedded Operational System.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

A necessidade, da comunicação entre dispositivos eletrônicos, gera uma área de estudos cujo campo para pesquisa é vasto. Com os avanços tecnológicos obtidos nos sistemas digitais, conseguimos dispositivos cada vez menores, a necessidade de comunicar estes pequenos dispositivos com outros da mesma categoria ou até com sistemas maiores é indispensável. Exemplos de sistemas assim podem estar em qualquer lugar, no carro, no avião, em casa.

Este trabalho visa amparar o programador de software para plataformas embutidas, dando suporte ao nível de sistema operacional para utilização de dispositivos embutidos com funcionalidade de conexão a redes CAN, uma estrutura de rede a qual provê várias opções desejáveis ou até indispensáveis para redes embutidas.

1.1 Objetivos Gerais

Este trabalho visa os seguintes objetivos:

- Implementação de um mediador de hardware para o EPOS*, utilizando microcontrolador da família AVR, está como principal objetivo deste trabalho. O mediador atuará entre o sistema operacional e o hardware, e através de uma abstração, atuará no software aplicativo, disponibilizando uma interface definida aos programadores de aplicativos para sistemas embutidos que desejam utilizar a tecnologia CAN para interconexão de dispositivos.

- Pesquisar a implementação de um protocolo que possa ser utilizado entre redes heterogêneas, utilizando canais de eventos [7], fazendo uso do mediador criado na etapa anterior.

* *Embedded Parallel Operating System*

1.2 Objetivos Específicos

São considerados os seguintes objetivos:

- O estudo da arquitetura de rede CAN, bem como de outras arquiteturas, como Profibus, indispensável para realizar o trabalho
- Aprofundamento nas funcionalidades do sistema operacional
- Estudo detalhado da família do microcontrolador utilizado para implementação, visando melhor utilização das funcionalidades disponibilizadas.

1.3 Motivação e Justificativa

CAN, apesar de ter sido criado em 1983, é uma tecnologia que está ganhando muita atenção para o desenvolvimento dos sistemas embutidos. Podemos hoje aplicar este tipo de rede em sistemas automobilísticos, industriais, comercias ou até mesmo aeroespaciais.

Vendas anuais de unidades com suporte a CAN de acordo com [9]:
1996 - ~11 milhões de unidades
1997 - ~24 milhões de unidades
1998 - ~97 milhões de unidades
1999 - ~123 milhões de unidades
2000 - ~137 milhões de unidades (projeção)

Para o programador, lidar diretamente com o hardware, pode significar um tempo muito grande perdido, já que o tempo de estudo e aprendizado das funcionalidades de uma dada plataforma pode ser grande, desta forma, uma provável mudança da mesma[8], necessitaria a realização de novo estudo e geração de uma novas implementações. Assim, utilizando um sistema operacional orientado a aplicação como o EPOS, uma interface padronizada e independente de plataforma pode ser gerada. Com esta interface, mudanças no software aplicativo tornam-se desnecessárias caso haja uma mudança de plataforma.

Como notado por Kaiser [7], a heterogeneidade entre redes utilizadas em sistemas embutidos representa um fator a ser considerado. A interação entre estas redes, muitas vezes indispensável, pode tornar-se um grande problema devido aos protocolos usados atualmente, problemas como: modo de endereçamento, propriedades da

transmissão, utilização eficiente dos recursos da rede. Estes problemas levam a produzir um protocolo independente de rede, porém, sem a perda das características necessárias para redes embutidas, como a interoperabilidade, eficiência, entre outras.

2 Controller Area Network

Com o desenvolvimento iniciado em 1983, na empresa BOSCH, para ter uma solução de uma interna para automóveis, “Controller Area Network”(CAN), foi anunciada oficialmente em 1986 pela BOSCH na Alemanha. Inicialmente para uso em unidades de controle eletrônico nos carros produzidos pela Mercedes. Em 1987 surgiram os primeiros circuitos integrados para CAN, fabricados pela Intel e pela Philips.

2.1 Características Gerais

CAN é um barramento serial para interligar dispositivos em rede. Como citado anteriormente, foi criada inicialmente para uso em sistemas de automóveis, mas logo teve o uso estendido para aplicações industriais. Hoje este barramento é usado primariamente em sistemas embutidos. Como detalhado por Livani, Kaiser e Jia [10], CAN possui facilidades que são muito desejadas na área da computação embutida, como tolerância a EMI*, prioridade de mensagens, recuperação de falhas, entre outras.

Uma rede CAN pode interligar até 2032 dispositivos, sendo que o limite prático é de aproximadamente 110 dispositivos [5], cada um destes é tratado como um nó da rede. No nível físico, o link serial mais usado é composto de dois fios, o sinal tem característica diferencial, é capaz de operar até 1 Mbps, tendo restrições de velocidade em virtude da distância entre os nós. Para uma rede, com extensão 1 km, a velocidade por ser reduzida até 50Kbps. Cada nó ligado a este link serial é capaz de ouvir, simultaneamente a outros nós, os dados transmitidos na rede. A escrita, porem, é uma operação permitida somente para um dispositivo por vez.

O protocolo CAN 2.0A tornou-se o padrão ISO 11898-1 em 1993. A última versão do protocolo é a 2.0B. A maior diferença entre as duas versões é a quantidade de bits no identificador, 11 bits para o 2.0A e 29 bits para o 2.0B.

Os padrões ISO para CAN, ou com origem em redes do deste tipo, são os seguintes: ISO 11898-2 (alta velocidade), ISO 11898-3 (tolerante a falhas / baixa

* Eletromagnetic Interference – Interferência Eletromagnética

velocidade), ISO 11992-1 e SAE J2411 (meio de transmissão utiliza uma única linha para transmissão e recepção).

2.2 Arquitetura

A rede é multi-master, ou seja, pode ter mais de um nó controlador, o que facilita a criação de um sistema redundante. Para isto, usa como protocolo de acesso o CSMA/CD+AMP (Carrier Sense Multiple Access/Collision Detection + Arbitration on Message Priority), desta forma, CAN trabalha de modo semelhante a ethernet comum, mas ao invés de corrigir colisões de transmissão fazendo com que os dois nós em conflito parem de transmitir, a rede CAN usa um arbítrio de comparação binária (figura 2.1) para definir a prioridade das mensagens e decide qual será enviada. Quanto menor o valor associado maior a prioridade. Os bits que trafegam na rede recebem uma denominação de dominante e recessivo, um bit dominante representa o valor lógico 0 e o recessivo, o valor lógico 1 [1](figura 2.2).

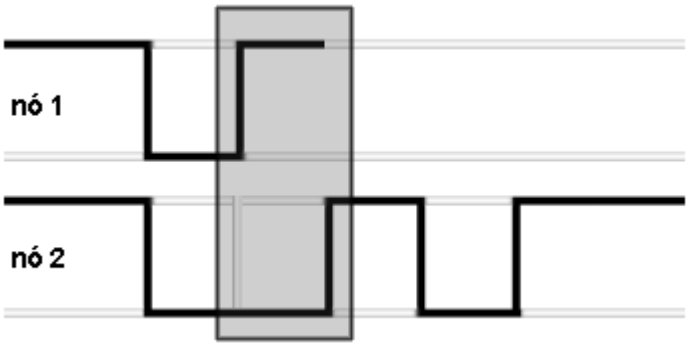


Figura 2.1: Exemplo do arbítrio sendo aplicado durante uma transmissão entre dois nós

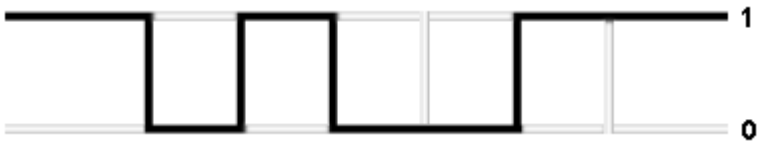


Figura 2.2: Estados lógicos do barramento em uma transmissão

As mensagens, transmitidas no barramento, não contem endereços de transmissor ou receptor, elas podem possuir um identificador único de acordo com o conteúdo, assim

cada receptor pode testar este identificador, portanto, se ele identificar um conteúdo relevante, a mensagem é processada.

A especificação de CAN para as camadas de rede, de acordo com a referência ISO/OSI se classifica da seguinte forma [1]:

Camada de Enlace

- LLC: responsável pela filtragem de mensagens, notificação de sobrecarga e controle de recuperação.
- MAC: encapsula/desencapsula dados, realiza codificação dos quadros(*Bit Stuffing*: caso aconteça 5 bits consecutivos apresentando o mesmo nível, insere um bit com valor inverso), controle de acesso ao meio, detecção e sinalização de erros

Estas duas subcamadas são responsáveis ainda pelo confinamento de falhas, ou seja, um nó que estiver com muitos erros de transmissão ou recepção poderá ser automaticamente desligado da rede. O controlador CAN é responsável por lidar automaticamente com estes serviços de forma transparente ao software.

Camada Física

Realiza codificação e decodificação dos bits utilizando NRZ (*Non Return to Zero*) para que o valor médio de ocorrência de bits recessivos e dominantes seja equilibrado, temporização e sincronização do sinal. As características desta camada não são definidas pela especificação da BOSCH, porém, a norma ISO define as características padrões para um transceiver [6]. Ela também é responsável pelo confinamento de falhas (juntamente com a camada de enlace) e tratamento de falhas provenientes do barramento.

Camada de Aplicação

É definida em nível de usuário, e não consta na especificação. Hoje existem algumas camadas especificadas: NMEA2000, CANopen, CANaerospace, DeviceNet, a CAN Kingdom, entre outras [2] [4].

Outras Camadas

Em geral as camadas intermediárias entre enlace e aplicação são parcialmente implementadas por protocolos de alto nível.

2.2.1 Tipos de Quadros

CAN utiliza variados tipos de quadros para envio de dados, requisição de dados, propagação de erros, e mensagens de notificação de sobrecarga.

2.2.1.1 Quadro de Dados

O quadro de dados (como pode ser visto nas figuras 2.3 e 2.4) é diferente para os padrões 2.0A e 2.0B. A partir destas figuras, podemos definir os campos da seguinte maneira [1]:

O Início como Start of Frame(SOF) consiste de um bit dominante, neste bit é realizada a primeira sincronização entre os nós.

Dentro dos campos seguintes temos os bits:

- RTR (*remote transmission reques*t, requisição de transmissão remota): requisição de transmissão remota, indica se o quadro é de dados ou de requisição remota, dominante para dados.
- IDE (*identifier extension,* identificador de extensão): indica se o quadro é padrão ou estendido, com valor recessivo para estendido.
- SRR (*substitute remote request,*substituto da requisição remota): para manter a compatibilidade entre o quadro estendido e o padrão, este é um bit recessivo presente no formato estendido.
- R0 e R1: bits reservados.

Os bits acima são agrupados em uma ordem diferente entre o campo de identificação e o campo de controle, a organização é dependente da versão do protocolo como se observa nas figuras 2.3 e 2.4.

Figura 2.3: Quadro 2.0B

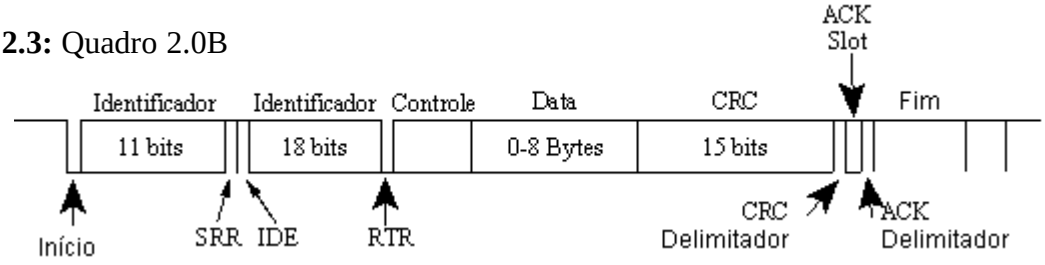
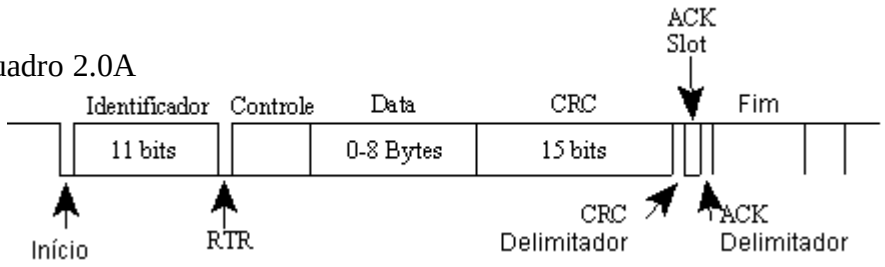


Figura 2.4: Quadro 2.0A



O campo de identificação, também é chamado de campo de arbítrio [1], já que este campo é utilizado para decidir quem continuará no barramento, agrupa os bits de identificação(11 bits) e o RTR no formato de quadro padrão. Os bits SRR, IDE e RTR fazem parte deste campo no formato estendido, sendo que a identificação possui 11 bits na primeira parte e 18 bits na segunda, totalizando 29 bits para identificação.

O campo de controle agrupa os bits IDE, r0 e mais 4 bits do DLC (data length code), este último informa tamanho do campo de dados. Utilizando o quadro estendido, o campo de controle passa a agrupar os bits R1, R0 e DLC.

O campo de dados é um segmento de 0 a 8 bytes de dados.

O campo de CRC, composto pela seqüência numérica gerada, é utilizado para verificação de erros no quadro, e um bit delimitador, sempre recessivo.

Seguindo o campo de CRC temos o campo ACK, que possui dois bits. Verificando os bits enviados pelo nó transmissor, veremos o envio de 2 bits recessivos, assim este campo delimita um tempo para que o receptor possa responder enviando um bit dominante caso tenha recebido o quadro com sucesso. Como no campo de CRC, o campo ACK é finalizado com um bit recessivo.

Para o fim do quadro temos o campo Final de quadro, composto por 7 bits recessivos. Este campo terá a funcionalidade mais detalhada na parte de detecção de erros, seção 2.5.

2.2.1.2 Quadro de Requisição Remota

Quando um nó necessita de algum dado, este nó envia um quadro remoto, uma requisição de dados. Este quadro remoto é idêntico ao quadro de dados exceto por não conter o campo de dados, e como já notado no item anterior, o bit RTR é recessivo.

2.2.1.3 Quadro de Erro

Possui um campo flag e um campo delimitador, este último consiste de 8 bits recessivos, já o campo de flag é composto por 6 bits. Os bits podem ser todos recessivos para um flag de erro ativo ou todos recessivos para um flag de erro passivo. A utilização do flag ativo ou passivo depende do estado do nó, conforme será visto no item 2.5.

2.2.1.4 Quadro de Sobrecarga

Um quadro de sobrecarga pode ser enviado por um nó para sinalizar que o receptor atual necessita de um tempo até o recebimento do próximo quadro, geralmente por falta de processamento ou memória.

O quadro é composto por 6 bits dominantes para o campo de flag, e mais 8 bits recessivos no campo delimitador do quadro.

Este quadro deve vir após um fim de quadro (para quadro de dados/remoto), um quadro de erro ou outro quadro de sobrecarga, porém deve-se respeitar a regra de no máximo dois quadros de sobrecarga consecutivos.

2.2.1.5 Espaço entre Quadros

O espaçamento entre quadros é sempre utilizado para separar quadro de dados ou requisição dos demais tipos de quadros. Este espaço é utilizado também para controle de transmissão de nós passivos conforme será visto no item 2.6.

O espaçamento de quadro é dividido em duas áreas para os nós ativos. A área de intermissão, onde um nó pode enviar um quadro de sobrecarga, é formada por 3 bits recessivos. A segunda área é aquela onde o barramento ficará livre até que ocorra alguma transmissão. Para um nó passivo, existe uma área intermediária onde ele não poderá transmitir quadros, este tempo é requerido quando ele transmitiu um quadro, o tamanho desta área é de 8 bits recessivos que são transmitidos pelo nó passivo responsável pela transmissão do quadro anterior. Os outros nós não são impedidos de realizarem transmissões neste tempo intermediário, caso isto aconteça o nó passivo pode tornar-se receptor da mensagem antes de completar o envio dos bits no barramento.

2.3 Filtragem e Validação de Mensagens

Para um nó receber uma mensagem transmitida no barramento ele deve estar preparado para receber mensagens do mesmo tipo daquela que foi transmitida. O tipo de mensagem é obtido através do identificador.

A comparação entre o identificador esperado e o identificador recebido pode levar em consideração máscaras. Para o receptor, quando uma mensagem válida é

encontrada, ou seja, não ocorreu erro até o bit Final de quadro ser recebido, e a comparação for positiva, o nó irá guardar o dado e sinalizar o sistema.

A validação de quadros pode também acontecer para o transmissor, tendo o mesmo critério de validação do receptor. Caso a mensagem não seja válida, ela será retransmitida automaticamente.

2.4 Configuração de Sincronização e Velocidade

Esta é uma parte crítica da rede, uma má configuração pode resultar em uma degradação da performance da rede ou até uma falha da mesma [18].

Para atingir a velocidade de transmissão requerida, existe uma grande combinação de configurações, que devem ter seus valores respeitados e configurados de acordo com o meio de transmissão.

CAN possui uma unidade temporal chamada “time quantum” ou tq, ela é definida pelo selecionador de taxa de transmissão do subsistema CAN, generalizando tempos: *tq = BRP / fsys*. Um bit transmitido no CAN possui a seguinte divisão [18]:

- Segmento de sincronismo: possui tamanho fixo de 1tq, é utilizado para o sincronismo entre a entrada do barramento e o relógio do sistema.
- Segmento de propagação: compensa atrasos causados pelo meio físico, incluindo tempo de transmissão, propagação do sinal no barramento e tempo para recepção do dado. Possui tamanho de 1 até 8 tq.
- Segmento de fase 1: indicará onde será feita a aquisição do valor do sinal, o qual está sendo lido no momento. Também pode ter tamanho de 1 até 8 tq, este tamanho pode ser variável durante o funcionamento para sincronização dos relógios.
- Segmento de fase 2: transmissões ocorrerão após este segmento. Pode ser configurado com 1 até 8 tq.

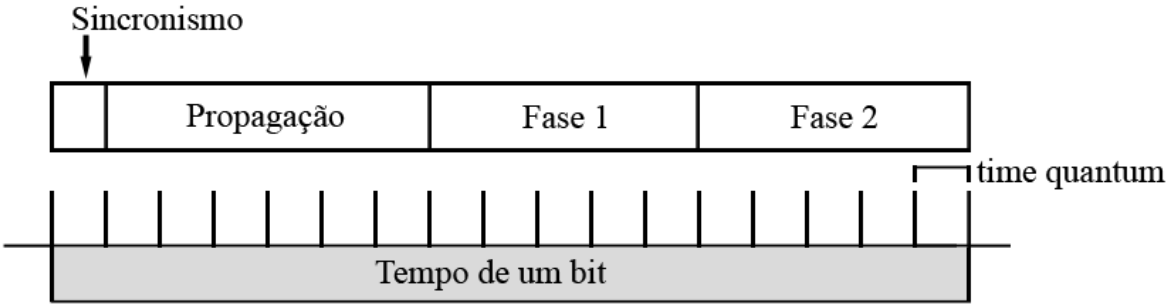


Figura 2.5: Segmentos do bit transmitido

Temos ainda um fator de correção para a sincronização: o tamanho do pulo desta, cuja variável pode ter o valor de 1 até 4 tq, e limitada também por não poder ser maior que qualquer um dos segmentos de fase. Esta variável indica o quanto o ponto de aquisição pode ser movido para efetuar uma re-sincronização.

Para o cálculo final e automático das variáveis, deve-se considerar alguns fatores comuns como pré-fixados. Estas considerações serão discutidas no capítulo sobre a implementação do mediador de hardware.

2.5 Tratamento de Erros

CAN possui um sistema muito confiável de tratamento de erros, todos os erros globais ao sistema são detectáveis, todos erros locais ao transmissor são detectáveis, uma mensagem pode conter até cinco erros distribuídos aleatoriamente, rajadas de erros com comprimento máximo de quinze bits ou de tamanho ímpar são detectados.

Podemos ter cinco tipos de erros detectáveis [1]:

- Erro de Bit: todo emissor continua monitorando os dados do barramento durante a transmissão caso o bit monitorado no barramento tenha valor diferente do que estava sendo enviado naquele momento é sinalizado este tipo de erro. Este erro não é levado em consideração caso o emissor esteja transmitindo dados do campo de arbítrio, neste caso, o emissor, cuja detecção apontou o erro, perde o controle do barramento. Existe ainda a possibilidade do emissor estar enviando um quadro de erro com flag passivo, assim, caso o bit monitorado esteja diferente ele será ignorado.

- Erro de codificação: este erro acontece quando o bit monitorado teve o mesmo valor seis vezes, na sexta ocorrência o erro é sinalizado.
- Erro de CRC: caso o valor do campo CRC transmitido não for igual ao do CRC recalculado no receptor, este erro será sinalizado.
- Erro de formação: ocorre quando um campo de formato pré-definido* possui um ou mais bits ilegais. Caso o valor do DLC seja maior que 8, não existe uma sinalização de erro definida, geralmente o erro é ignorado e o valor é dado como o maior possível.
- Erro no campo ACK: será sinalizado este erro caso o transmissor não detecte um bit dominante durante a transmissão do campo ACK.

A sinalização de erro é aplicada igualmente a todos os tipos exceto para o erro de CRC. Enquanto a sinalização padrão é transmitida no próximo bit após a detecção do erro, o erro de CRC é transmitido somente depois do recebimento do delimitador no campo ACK.

Para controle de erro no barramento, o confinamento de falhas, cada nó possui um contador de erros de transmissão e recepção. Com base nestes dois contadores, um nó pode estar em um destes modos:

- Ativo: nó em funcionamento normal.
- Passivo: nó funcional, porém, com certas restrições na transmissão de pacotes. O nó entra neste estado após o contador de erros de transmissão ter o valor maior ou igual a 128. Um nó entrando neste estado só pode voltar a ser um nó ativo caso os contadores de erros de transmissão e também de recepção estejam com valores inferiores a 128.
- Inativo: este nó não pode exercer modificações no estado do barramento. Caso o contador de erros de transmissão ultrapasse ou iguale o valor 256, ele deve entrar neste estado. Deste estado, um nó, pode tornar-se ativo após ocorrerem 128 pacotes contendo 11 bits recessivos, ao voltar ao estado ativo, o nó, terá os contadores de erros zerados.

Como um exemplo, a máquina de estados, implementada no microcontrolador AT90CAN128, pode ser dada como demonstração na figura 2.6.

* Campo de CRC, ACK, Final de quadro e quadros de Erro e Sobrecarga são de forma pré-definida e não utilizam codificação NRZ.

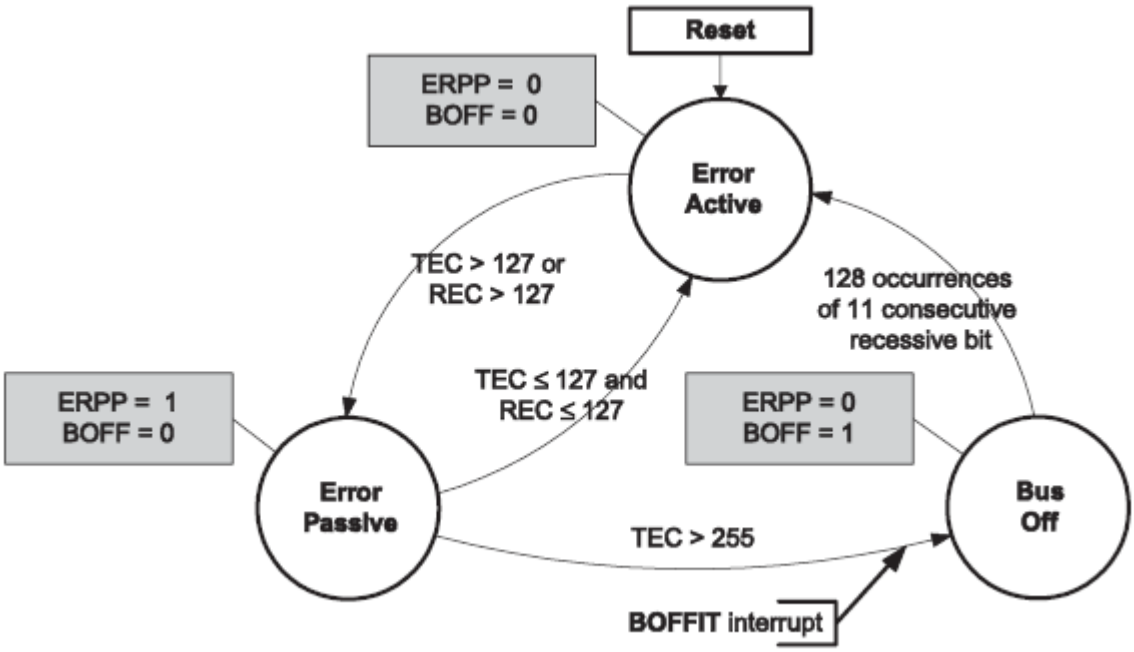


Figura 2.6: Máquina de estados implementada pelo AT90CAN128 [2]

2.6 Aplicações

Como visto anteriormente, pode ser muito interessante utilizar uma rede CAN quando lidamos com ambientes hostis, ou que necessitem de transmissão de mensagens com prioridade e capacidade de serem enviadas em tempo real. Contudo, ainda podemos aplicar CAN para redes comuns onde vários microcontroladores e sensores necessitem de interconexão. Assim verificamos que CAN está presente em todo tipo de setor, em especial os que seguem abaixo.

2.6.1 Aplicações Agrícolas

Com o aumento dos equipamentos utilizados no campo, o usuário sentiu a necessidade da interconexão entre eles, assim surgiu um padrão para interligação de equipamentos agrícolas, o ISO 11783, sendo implementado no mercado o mínimo do padrão como ISOBUS.

Este é um padrão recente, ainda em fase de desenvolvimento por vários fabricantes internacionais. No Brasil, já existem pesquisas em andamento, sendo que um grupo chamado “força tarefa ISOBUS” foi criado para ajudar com que o padrão seja utilizado também aqui no Brasil [17].

2.6.2 Aplicações Aeroespaciais

Como podemos notar, CAN é uma excelente escolha para este tipo de aplicação considerando o alto índice de interferência sofrida pelos dispositivos deste setor.

Para o setor aéreo, existe o padrão CANaerospace criado pela Stock Flight Systems, conforme a especificação [11]. É um protocolo extremamente leve e confiável para aplicações aéreas. Utilizado e padronizado posteriormente pela NASA, teve seu poder demonstrado com a rede CAN atuando como um backbone para sensores (principalmente na parte dos motores), atuadores, sistema de navegação e também para um gateway entre a internet e o sistema de vôo. Este sistema também é utilizado em empresas como a Airbus e a Bombardier.

Para aplicações espaciais, podemos citar a ESA como uma das grandes usuárias desta arquitetura, como foi demonstrada na sonda SMART-1 [12], uma sonda desenvolvida para demonstrar tecnologias chaves no futuro da exploração espacial pela agência.

CAN também foi utilizado no satélite AAUSAT-II da universidade de Aalborg [13] [14] e inúmeros outros satélites experimentais, principalmente universitários e de radioamadores, como rede local para os módulos dos satélites, provando na prática a eficiência de CAN para ambientes hostis.

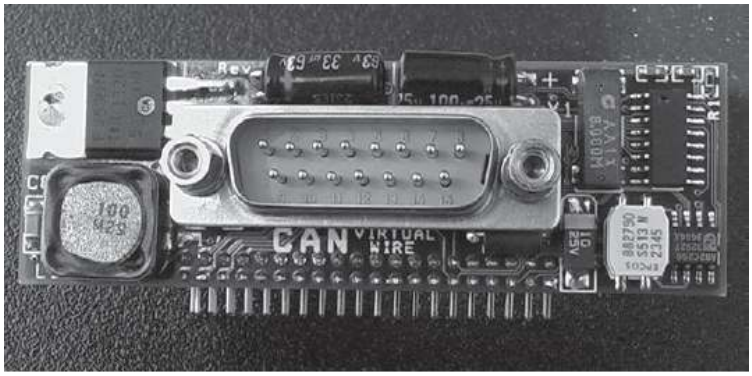


Figura 2.7: Módulo CAN desenvolvido para o RUDAK a bordo do satélite P3D (AO-40) [24]

2.6.3 Aplicações Automobilísticas

Este setor foi responsável pela criação do CAN, portanto, inúmeras aplicações utilizam as vantagens de redes CAN para o funcionamento. A criação de redes de sensores internos do veículo ajuda a melhorar o funcionamento de determinados atuadores como, por exemplo, de freios ABS, tendo informações sobre o funcionamento global do

sistema de sensores/atuadores, um determinado atuador pode ser utilizado usando uma estratégia mais adequada de acionamento.

A rede CAN é utilizada também para comunicação entre vidros elétricos, acionamento de abertura de portas, teto solar, regulagens de bancos entre outras. Sistemas de diagnóstico de componentes do carro, também, são implementados utilizando CAN.

Desenvolvido inteiramente no Brasil pela Scania, o sistema IRIS (inteligência e rastreamento integrado por satélite) utiliza CAN para comunicação entre os vários sensores presentes no sistema embutido em caminhões.

Fabricantes como a BMW, Ford, General Motors, Toyota, DaimlerChrysler são alguns dos utilizadores desta tecnologia de interconexão.

2.6.4 Aplicações Comerciais

Aplicações comerciais geralmente não necessitam das funcionalidades providas pelo CAN, porém devido a grande disponibilidade de componentes, baixo custo e boa aceitação em outras áreas, CAN também é uma alternativa atrativa entre as tecnologias de rede que competem no setor comercial.

Na automação de prédios, por exemplo, CAN pode ser utilizado para a interligação do controle de abertura de portões e portas, iluminação, ventilação, detectores de fumaça, estado dos elevadores, entre outras. Hoje, existe até cafeteiras complexas utilizando CAN para interconexão entre módulos e outras máquinas[3]. Em grande parte das aplicações comerciais, baseadas em CAN, é utilizado a camada de aplicação CANopen.

2.6.5 Aplicações Industriais

Novamente uma área onde são necessárias a robustez e resistência a interferências. Em um ambiente onde podem existir inúmeros motores elétricos e equipamentos de microondas de alta potência, a interferência eletromagnética pode tornar inoperantes outros tipos de interconexões.

O DeviceNet foi criado e mantido como uma camada de aplicação padrão, para produtos industriais, sendo assim a maior parte dos produtos da área, os quais utilizam CAN, podem ser interconectados.

2.6.6 Aplicações Médicas

As características da tecnologia CAN são muito adequadas para aplicações médicas, já que elas têm confiabilidade e segurança de transmissão comprovada por vários outros setores.

Das aplicações médicas onde CAN está presente podem ser citadas: suporte de vida (principalmente para recém-nascidos), raios-X, controle de equipamentos cirúrgicos e de laboratório [15].

Aplicações médicas utilizam a camada de aplicação CANopen como padrão. Esta camada foi especificada em conjunto pela GE Medical Systems, Philips Medical e Siemens Medical sob o nome da organização CiA [16]. Esta união de grandes empresas para a definição de uma camada padronizada demonstra a alta importância da tecnologia para o setor.

2.7 Comparação com outras Tecnologias

CAN é uma das tecnologias categorizadas como *fieldbus*, portanto, é interessante comparar esta tecnologia com outras da mesma categoria. Infelizmente, não temos uma metodologia de comparação capaz de demonstrar qual rede é melhor ou pior devido aos setores onde tais tecnologias são aplicadas. Assim, será exibida uma comparação geral e, ao final, uma descrição mais detalhada da rede PROFIBUS.

2.7.1 Tabelas de Comparação

As tabelas [21] a seguir irão demonstrar características físicas e operacionais das seguintes redes: CAN, AS-I [19], Foundation Fieldbus H1/HSE* e PROFIBUS DP/PA** [20, 22].

* High Speed Ethernet
** Decentralized Periphery / Process Automation

	Topologia	**Meio utilizado**	**Extensão máxima**
CAN	Barra	Fibra ótica e par trançado	1km
AS-I	Árvore, barra	2 fios	100m 300m com repetidor
Foundation Fieldbus H1	Barra, estrela	Fibra óptica e par trançado	1900m
Foundation Fieldbus HSE	Estrela		100m(par trançado) 2km(fibra óptica)
PROFIBUS DP	Anel, barra, estrela	Fibra óptica* e par trançado	100m por segmento, até 24km utilizando fibra óptica
PROFIBUS PA			

Tabela 2.7.1a

	Acesso ao meio físico	**Quantidade de conexões suportadas**
CAN	CSMA/CD+AMP	2032**
AS-I	Polling cíclico mestre/escravo	62 + 1 mestre***
Foundation Fieldbus H1	Token Ring****	240 por segmento
Foundation Fieldbus HSE	CSMA/CD	2^{32} (endereçamento IP)
PROFIBUS DP	Token Ring	32 por segmento (máximo: 126)
PROFIBUS PA		

Tabela 2.7.1b

	Tamanho máximo de transferência	**Velocidade**
CAN	8 bytes	Até 1 Mbit/s
AS-I	8 bits	167 Kbit/s
Foundation Fieldbus H1	128 bytes	31,25 Kbit/s
Foundation Fieldbus HSE	Variável (TCP/IP)	Até 100 Mbit/s

* Somente para PROFIBUS DP

** CAN não utiliza endereçamento de nodos para enviar as mensagens. A camada de aplicação irá determinar efetivamente as quantidades de nodos possíveis a serem conectados, este número também será limitado pelas características do meio físico, sendo o limite prático perto de 110 dispositivos

*** De acordo com especificação da versão 2.1

**** Passagem de bastão

PROFIBUS DP	Variável, 0 até 244 bytes	Até 12 Mbit/s
PROFIBUS PA		31,25 kbit/s

Tabela 2.7.1c

2.7.2 PROFIBUS

Com início em 1989(PROFIBUS FMS), e aprimorada em 1993(PROFIBUS DP), atualmente é tida como sendo líder do segmento com cerca de 20% do mercado [22]. É utilizada largamente para automação de manufaturas e controle de processos [23]. Seguindo ao modelo OSI, esta tecnologia especifica as camadas 1, 2 e 7. Os diferentes grupos de configurações obtidos pela junção das camadas 1, 2 e 7 geram as versões. As versões típicas são: PROFIBUS DP (voltado para automação de fábricas) e PROFIBUS PA (automação de processos). Para uma determinada aplicação, podemos ter um perfil definido, estes serão comentados na seção 2.8.2.4.

2.7.2.1 Camada Física

Assim como CAN, o PROFIBUS DP também utiliza um sinal diferencial (RS-485) para a transmissão e NRZ para codificação dos bits a serem enviados. Atingindo taxas de até 12 Mbit/s. Para o PROFIBUS PA, temos um meio chamado MBP-IS(*Manchester Coded, Bus Powered-Intrinsic Safety*) onde a transferência de dados é síncrona e com a taxa de transmissão fixa de 31,25 Kbit/s. Outros versões podem utilizar ainda o RS485-IS(até 1,5 Mbit/s) ou Fibra óptica(12Mbit/s, NRZ).

Temos três diferentes protocolos de comunicação para a camada física, cada um destes agrega funções ao anterior:

1. DP-V0: transferência cíclica de dados entre mestre e escravo, diagnóstico de dispositivo, módulo e canal
2. DP-V1: transferência acíclica de dados entre mestre e escravo, diagnóstico estendido, definindo tipos de alarmes enviados na rede (estado, atualização ou outro definido pelo fabricante)
3. DP-V2: comunicação escravo-escravo e modo isócrono

2.7.2.2 Camada de Enlace

Define a forma de acesso ao meio como mestre-escravo e o uso da passagem de bastão para ordenação do mestre que terá acesso ao meio (caso seja feito uso de mais de um

mestre na rede). As mensagens geradas para o sistema dispõem de duas prioridades possíveis: alta e baixa.

2.7.2.3 Camada de Aplicação

Define como a aplicação fará interface com a rede, oferecendo o serviço de transferência de dados.

2.7.2.4 Perfis

Como citado anteriormente, existem perfis criados para fins específicos. Estes perfis podem ser desenvolvidos por usuários ou fabricantes, serão categorizados como perfis de aplicações específicas, já que desejam algumas propriedades específicas, como performance diferenciada ou comportamento do sistema [22]. Exemplos de perfis desta categoria: PA Devices, PROFIdrive. O PROFIBUS PA, por exemplo, utiliza o perfil PA Devices e o protocolo DP-V1.

Uma segunda categoria, perfil do sistema, também é citada. Ela engloba o perfil de mestres, a qual faz a descrição de como o mestre irá atuar (transferência cíclica, acíclica, diagnósticos, etc). O perfil do sistema ainda é responsável pela descrição dos parâmetros de funcionamento do sistema. Estes parâmetros poderão ser utilizados pelos dispositivos conectados na rede a fim de completar a configuração descrita no perfil da aplicação.

2.7.2.5 Tipos de Dispositivos

Como este tipo de rede utiliza o sistema mestre-escravo, os dispositivos foram classificados da seguinte forma:

- Mestre: geralmente CLP* ou PC. A subclassificação dos mestres acontece da seguinte forma, classe 1: realiza troca de dados cíclica com os escravos, possui acesso ativo ao sistema; classe 2: é caracterizada por dispositivos usados para diagnóstico da rede e que não ficam conectados permanentemente no sistema.
- Escravo: qualquer dispositivo, de comportamento passivo, conectado na rede.

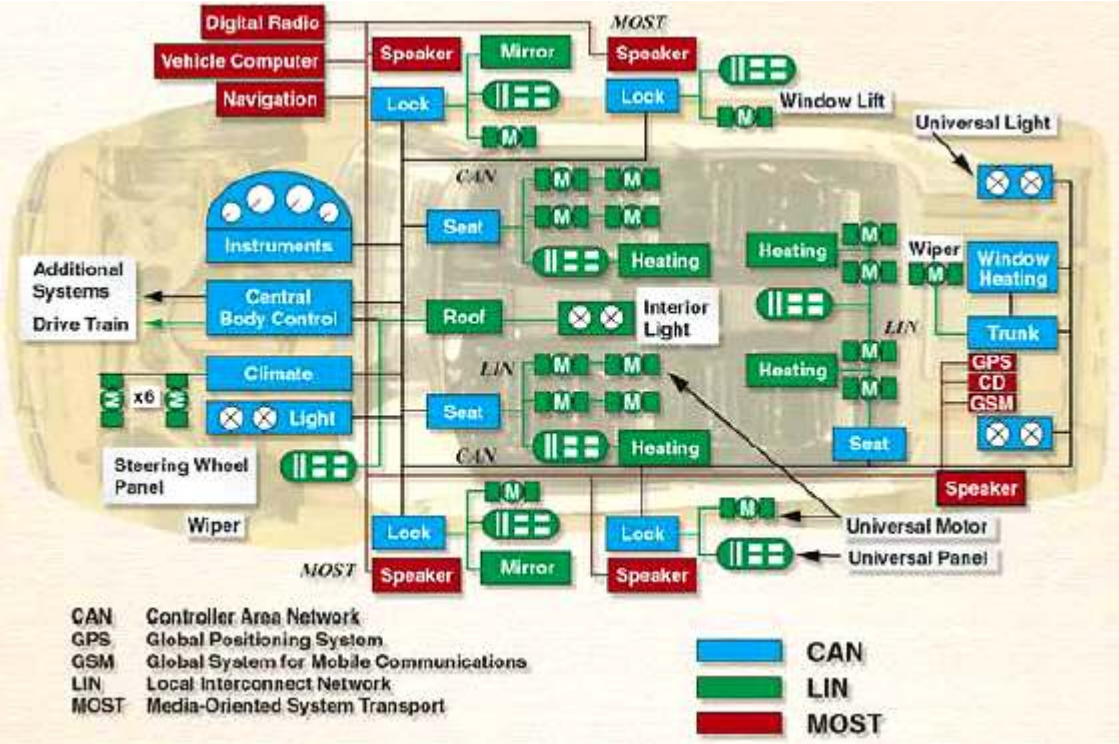
* Controlador lógico programável

2.7.3 Discussão Tecnológica

Não temos uma tecnologia disponível onde todos os casos de uso sejam enquadrados com perfeição, a escolha de uma tecnologia tornou-se muito dependente da aplicação, parâmetros como tamanho da rede, taxa de atualização de dados, número de nodos conectados e segurança da transferência da informação são significativos para a escolha do sistema [26].

Com este problema de escolha, acabamos, muitas vezes, optando pela heterogenização do sistema com a utilização duas ou mais tecnologias de conexão. Como citado em [25], esta coexistência de tecnologias tende a crescer cada vez mais devido aos requisitos atuais onde devemos contrabalançar segurança, velocidade e custos. Um bom exemplo disto são os carros modernos, hoje além de toda a parte de controle dos sistemas do carro, geralmente onde CAN é aplicada, temos uma integração com uma rede de entretenimento dos usuários do veículo, a qual deve suportar taxas de transferências muitas vezes superior devido ao tráfego multimídia. Temos também novos dispositivos de auxilio a navegação que devem trocar dados a uma grande velocidade, exigindo uma rede complementar com uma maior taxa de transferência, porém segura e que, em caso de mau funcionamento, não afete as redes principais de controle. Podemos observar na figura 2.8 uma demonstração de como poderiam estar distribuídas as diferentes tecnologias dentro de um carro.

Figura 2.8: Redes complementares dentro de um carro [25]. Temos a rede LIN para controle de dispositivos não críticos, com baixa taxa de transferência de dados e que geralmente possuem custo reduzido, a rede CAN para os dispositivos críticos do sistema, e uma rede MOST, com taxa de transferência de 24.8 Mbps, para a parte multimídia.



3 CAN para o EPOS

Neste capítulo será realizada uma breve introdução ao sistema operacional EPOS, o qual foi utilizado como base para a implementação do mediador de hardware e pesquisa para abstração de rede. A plataforma escolhida para a implementação terá sua descrição feita logo depois. Ao final do capítulo será demonstrada a estrutura do mediador CAN implementado e como CAN pode ser mapeado para uma abstração de rede do sistema.

3.1 EPOS

O EPOS é um sistema operacional implementado por Fröhlich[8], utilizando técnicas AOSD*. Com ele, Fröhlich também demonstrou a viabilidade do uso deste tipo de sistema operacional em ambientes heterogêneos, já que um sistema operacional orientado à aplicação, ou seja, configurado, utilizando somente os componentes necessários para funcionamento da mesma[8], pode ser utilizado em um ambiente de alta performance, com um menor impacto sobre a memória e outros recursos do sistema.

Esta adaptabilidade apresentada pelo EPOS, faz dele um sistema de bom custo benefício tanto para ambientes embutido, onde podem estar sendo utilizado microcontroladores de 8 bit (comuns para redes de sensores por exemplo), até sistemas de alta performance, como foi o caso do Cluster SNOW[3] utilizado para validação deste sistema operacional.

O fato de este sistema operacional poder ser utilizado com sistemas 8 bit traz uma nova perspectiva para desenvolvedores do setor, os quais, dominam mais de sessenta por cento do mercado de processadores (figura 3.1) e que possuem poucas ferramentas para o desenvolvimento, principalmente quando analisamos portabilidade das aplicaçações.

* Application Oriented System Design

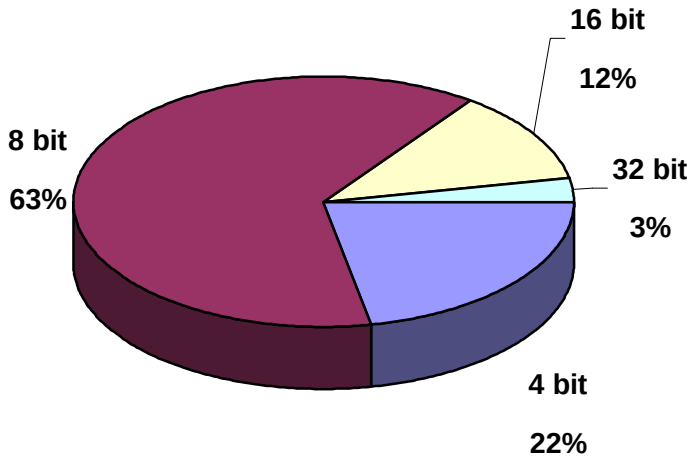


Figura 3.1: Tipos de processadores utilizados no ano 2000 [27], 98% destes foram utilizados para sistemas embutidos.

3.1.1 A escolha do EPOS

A escolha deste sistema para base da implementação desenvolvida com este trabalho não foi ao acaso. Os serviços providos pelo sistema operacional são essenciais para um desenvolvimento rápido e com menor possibilidade de erros. Assim, escolhendo o EPOS, além de prover a facilidade do uso para CAN, o desenvolvedor já poderá utilizar uma gama de funções já disponibilizada pelo sistema operacional como abstrações de threads, sensores, etc. Estas funcionalidades também aceleraram o desenvolvimento do próprio mediador CAN implementado.

Analisando a portabilidade do código, utilizando um sistema operacional multi-plataforma como o EPOS, o esforço para utilizar o mesmo aplicativo em uma plataforma distinta será pequeno, talvez inexistente, o que não aconteceria caso o aplicativo fosse desenvolvido utilizando bibliotecas ou baseando-se em uma linguagem de programação como base de sustentação para portabilidade [33].

3.1.2 Arquitetura do EPOS

O EPOS é dividido em três partes básicas, as abstrações, os aspectos e os mediadores, porém, a versão utilizada neste trabalho está propositalmente desprovida dos aspectos.

As abstrações englobam todas as implementações independentes de plataforma, por exemplo, threads, networking, etc.

Para implementação de partes específicas das plataformas, o EPOS utiliza um mediador. Caso exista uma funcionalidade comum para mais de uma plataforma suportada pelo EPOS, cada plataforma possuirá um mediador específico, porém existirá uma interface definida, um “contrato de interface” entre o sistema e a máquina [33], alcançando assim a portabilidade das abstrações.

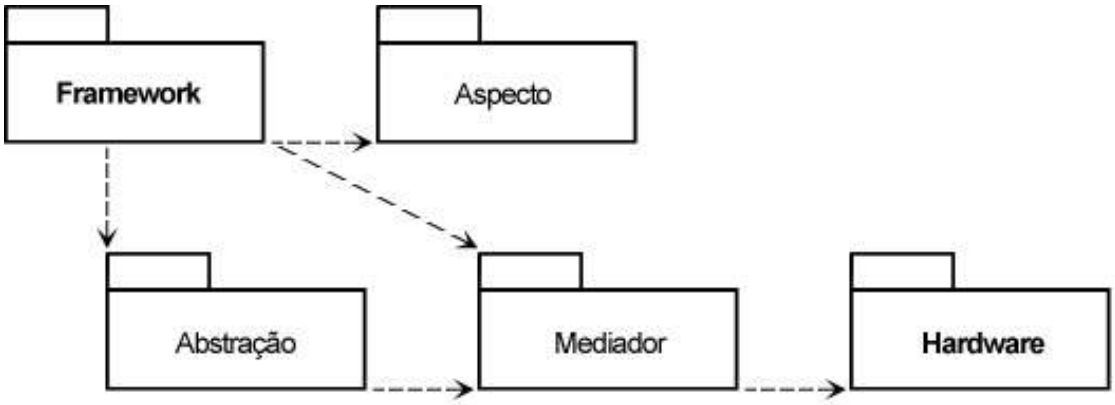


Figura 3.2: Relação entre as famílias do EPOS

3.2 Hardware utilizado

Para implementação do mediador CAN para o EPOS, foi escolhido o microcontrolador AVR, modelo AT90CAN128. Da mesma forma pela qual o EPOS foi escolhido como sistema operacional, a escolha do AVR foi favorecida por diversos fatores. O primeiro deles foi o suporte do EPOS para a arquitetura. Com o suporte provido pelo sistema operacional, foi criada uma arquitetura derivada, onde o código implementado, na maioria das vezes, retrata apenas as diferenças e funcionalidades adicionais, como algumas pequenas modificações no hardware base e adições como o suporte a CAN.

Outro fato importante é o AVR possuir uma aceitação cada vez maior no mercado, deste modo esta linha de microcontroladores tende a crescer em qualidade e em opções, tornando-se mais importante, conforme demonstrado na figura abaixo.

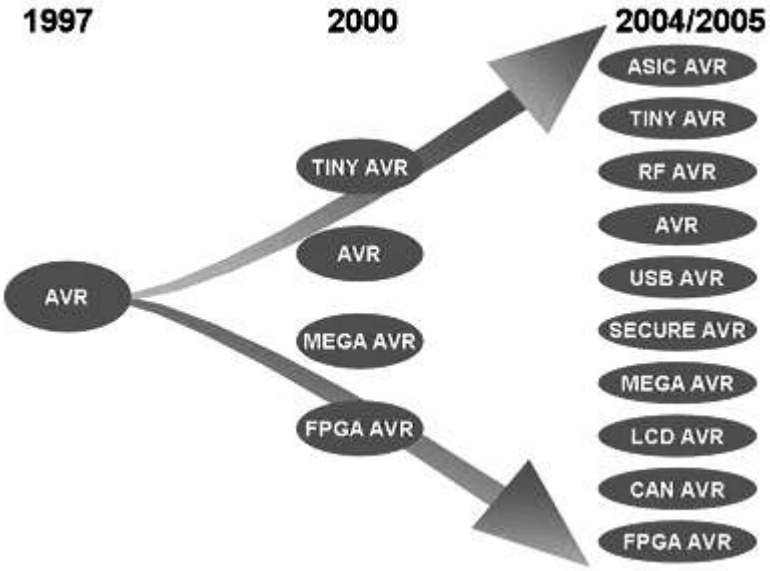


Figura 3.3: Desenvolvimento da família AVR [29]

Como indicação final, o AT90CAN128 é um dos poucos microcontroladores 8 bit existente no mercado possuindo um controlador CAN embutido, com a especificação 2.0B totalmente implementada.

3.2.1 Arquitetura básica

Um AVR é um microcontrolador RISC, baseado na arquitetura de Harvard, possui 32 registradores de 8 bits para de uso geral, o grupo de instruções possui em média 130 instruções, as quais, em grande parte, serão executadas em um único ciclo. O microcontrolador possui uma série de portas de entrada e saída de uso geral. Como um adicional de performance, as unidades, baseadas na versão Atmega, possuem uma instrução de multiplicação executada em 2 ciclos de clock.

A facilidade de programação da memória flash embutida no controlador é um dos grandes atrativos juntamente com o baixo consumo de energia e alta performance (quando comparado a outros controladores da mesma área de atuação).

3.2.1.1 Características Principais

Registradores

Existem quatro tipos de registradores no AVR: registrador de estado, registradores de propósito geral, registrador de página e registradores de entrada/saída.

O registrador de estado (SREG) guarda as informações sobre a ultima instrução aritmética executada e também possui o bit para habilitar/desabilitar globalmente as interrupções.

Os 32 registradores de propósito geral podem ser utilizados para executar as operações requeridas com alto desempenho. Dependendo da escolha do modo de endereçamento, os últimos seis registradores do banco de registradores podem ser mapeados com uma função especifica, simular registradores de 16 bits para efetuar um endereçamento indireto para a área da memória de

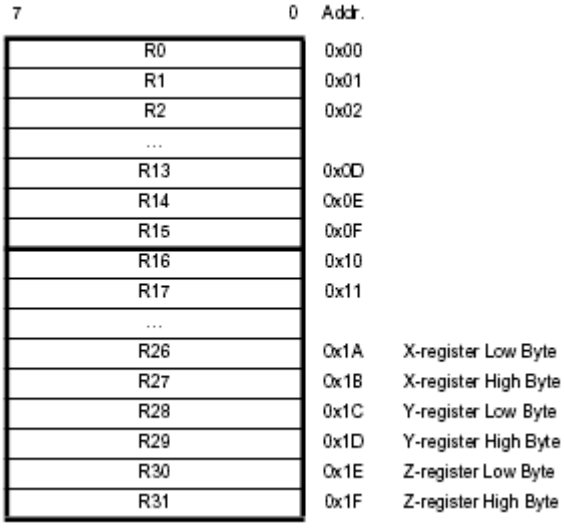


Figura 3.4: Banco de registradores do AVR [2]

dados, criando os registradores de apontador X, Y e Z.

Presente somente nos modelos que podem utilizar memória externa, o registrador de página da RAM (RAMPZ), é utilizado para dizer qual página do banco de memória externo deve ser utilizada quando uma determinada área da memória for referenciada pelo registrador Z. Como o AT90CAN128 não suporta uma memória externa com mais de 64K, este registrador somente seleciona a área da memória de programa (organizada em blocos de 64K) utilizada quando ocorrer instruções de leitura ou escrita na memória de programa.

Registradores de entrada e saída são utilizados para o controle dos “periféricos”, eles são a interface de comunicação com recursos como o CAN.

Os registradores descritos podem estar localizados em um espaço reservado de entrada/saída (registradores de 0x00 até 0x3F), sendo estes acessíveis com instrução IN/OUT. Os 32 registradores de propósito geral e os registradores de entrada e saída acima de 0x3F são somente mapeados em memória, onde, instruções comuns de acesso à memória são utilizadas para escrever ou ler os valores deles. Os registradores localizados no espaço reservado de entrada e saída, também estão mapeados em memória, facilitando a programação (não será necessário verificar o tipo de registrador utilizado).

Memória

Como citado anteriormente, a memória no AVR é dividida em memória de programa e memória de dados. A primeira é dividida em 64K x 16, devido às instruções do AVR serem em sua grande maioria de 16 ou 32 bits. As versões, derivadas da família ATmega, possuem uma opção para reservar uma parte da memória de programa para um sistema de boot. Nestes modelos ainda é possível modificar a memória de programa utilizando o próprio controlador, por exemplo, podemos construir um setor de boot que fica esperando pela programação via UART, tão logo ele receba os dados, estes dados serão escritos na memória de programa e, quando a seqüência for finalizada o aplicativo é executado.

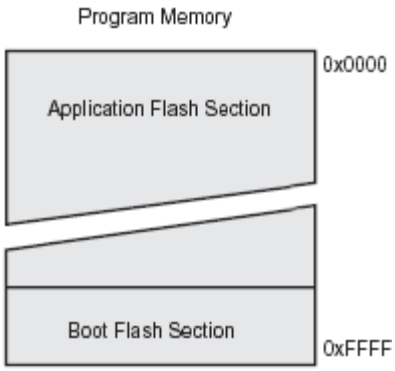


Figura 3.5: Memória de programa no AT90CAN128

A memória de dados é dividida de acordo com a figura ao lado. Caso a memória externa esteja habilitada, e ocorra a leitura ou escrita para um ponteiro maior do que a área interna da memória do controlador, a memória externa será utilizada automaticamente. Esta facilidade de acesso à memória externa irá deixar no máximo 61.184 bytes dela disponível para uso, já que os 4.352 bytes iniciais são utilizados para o mapeamento de registradores e memória interna. Caso a memória externa seja menor que 64K, pode ser feito um mapeamento da área inicial da memória externa para ela aparecer no final da área de memória mapeada no controlador.

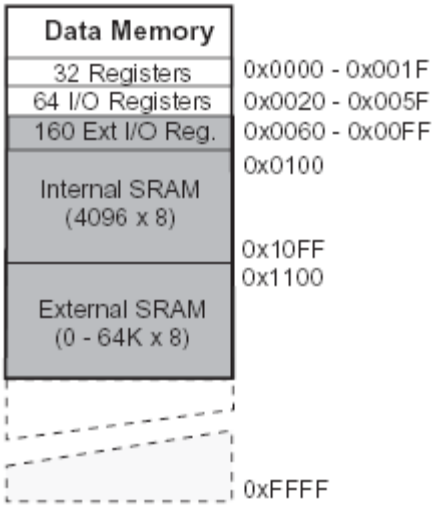


Figura 3.6: Memória de dados no AT90CAN128

3.2.1.2 Periféricos

ADC (Conversor Analógico Digital)

A maior parte dos controladores AVR possuem conversores analógico digital de 8 ou 10 bits. Nos microcontroladores com mais de uma entrada analógica, o ADC é multiplexado entre as várias entradas. Ele pode usar uma referência de voltagem externa para as aquisições ou utilizar uma referência interna de 2,56 volts.

O conversor pode operar em modo de aquisição simples, somente uma aquisição é feita ou *free run*, onde comparador realiza aquisições com um período programado. Nos modelos recentes existe a opção de realizar automaticamente uma aquisição vinculada a uma interrupção de timer ou interrupções externas.

Os resultados das aquisições podem ser melhorados utilizando o sistema em modo sleep enquanto o ADC realiza a conversão, evitando assim ruídos indesejados gerados pela unidade.

Pinos de uso geral (GPIO)

O AVR é provido de uma ou mais portas de entrada e saída para uso geral, cada uma delas é configurável ou acessada através dos registradores DDRx, PORTx e PINx (x representa uma porta, A, B, etc), cada porta pode acomodar até oito pinos de uso geral.

O registrador DDR indica a direção da porta, entrada ou saída. PORT é um registrador de valor lógico, usado para ligar ou desligar um determinado pino quando a porta é utilizada como saída. Com a configuração para saída, combinado com o bit PUD do registrador MCUCR, o PORT irá ativar ou desativar o resistor de pull-up interno. PIN é o valor efetivo (físico) da porta, este registrador é somente para leitura, mas, se houver uma escrita modificando algum bit dele, ocorrerá uma mudança de estado no bit correspondente do registrador PORT.

Cada bit dos registradores pode ser configurado separadamente sem afetar os outros presentes na mesma porta.

As portas de uso geral também podem ser configuradas para o uso de funções específicas, como a interface serial, estas funções podem ser ativadas através dos registradores específicos relacionados a elas.

Timers

Como padrão da família AVR, eles são periféricos que permitem o software realizar uma contagem temporal, baseado em um clock de referência fornecido internamente ou externamente. Os timers podem operar em diversos modos, podemos generalizar o uso do timer nas seguintes categorias:

- Normal: o contador é incrementado no ciclo programado e quando atinge o máximo número de contagem gera uma flag de overflow.
- Comparação: como no anterior, o timer é incrementado conforme o clock programado, porém, um ou mais registradores do timer armazenam um número definido pelo usuário, quando o contador do timer atinge um destes números é gerado o flag, compare match. O timer pode ser configurado para reiniciar a contagem ou continuar normalmente após este evento.
- Captura de entrada: ao ocorrer um evento externo, quando um pino determinado do AVR é acionado, o valor do contador é copiado para um registrador especial, assim é possível gerar log dos eventos ou medir o intervalo de tempo entre eles.
- PWM*: a saída do timer opera sobre um dos pinos do AVR, gerando assim uma onda pulsada modulada por largura.

* Pulse Width Modulation

Os flags gerados podem ser associados a interrupções do sistema caso necessário.

Alguns modelos de AVR, como o AT90CAN128, possuem ainda um modulador de saída por comparação, este combina a saída de dois timers para atuar em um dos pinos de saída possibilitando a geração de ondas moduladas com uma portadora.

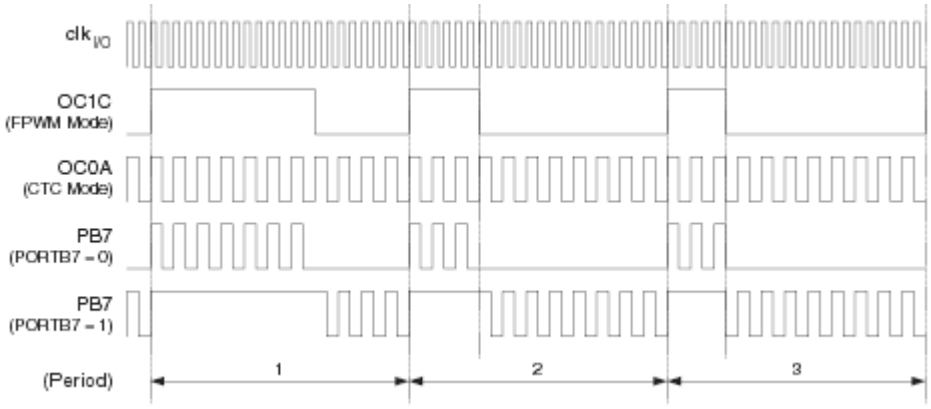


Figura 3.7: Exemplo de saída modulada no AT90CAN128

SPI (Interface Serial para Periféricos)

É um sistema de transferência de dados síncrona entre um dispositivo mestre e os periféricos ligados a ele. É do tipo *full duplex* e utiliza três linhas de sinais para trabalhar desde modo, Rx, Tx e Clock. O AVR pode trabalhar tanto no modo mestre quanto no modo escravo.

Uma das formas mais comuns e baratas de programação do microcontrolador faz-se através do SPI.

UART, USART (Receptor/Transmissor Serial Síncrono/Assíncrono)

Para famílias mais antigas do AVR era implementada uma serial universal que não apresentava modo síncrono, porém, o hardware mais comum de serial implementada nos microcontroladores ATmega hoje é a USART, sendo que esta pode operar no modo síncrono.

Em geral, a interface serial do AVR suporta transmissão utilizando vários formatos de quadro, com 5 até 9 bits de dados, 1 ou 2 bits de parada, com paridade par, ímpar ou nenhuma paridade.

A interface serial do AVR detecta os seguintes erros: erro de quadro, paridade e perda de dados (caso o dado recebido pelo controlador seja sobrescrito antes do software consumi-lo).

TWI (Interface Serial de Duas linhas)

Assim como o SPI, é um tipo de barramento para controle de periféricos, porém utiliza somente dois fios, um para clock e outro para a transmissão dos dados. O AVR suporta trabalhar tanto no modo mestre como escravo, e também em barramentos apresentando mais de um mestre.

CAN

Periférico exclusivo do modelo AT90CAN128. O controlador CAN embutido neste modelo, como citado anteriormente, é totalmente compatível com os padrões CAN 2.0A e 2.0B, suportando todos os tipos de quadros. Pode armazenar até 15 mensagens, chegar a taxa de transmissão de 1Mbit/s utilizando um clock de 8Mhz(metade do clock máximo alcançado por este modelo), possui um timer dedicado caso seja necessário implementar um protocolo TTCAN(*time triggered*).

O controlador CAN ainda pode operar em um modo somente de escuta, com isto pode ser implementado um algoritmo para achar automaticamente o baud rate da rede onde foi inserido o dispositivo. Ele fornece suporte completo a camada de enlace e parcial para a camada física, para ligação efetiva do controlador ao barramento é necessário outro dispositivo (um transceiver), o qual irá transmitir os sinais de acordo com o meio utilizado.

Outros

O AVR pode contar ainda com um comparador analógico, utilizando dois pinos de entrada analógica para verificar a diferença de tensão dois pinos específicos. O watchdog pode ser utilizado para evitar travamento por tempo indeterminado. Nas unidades mais recentes, temos a opção de utilizar o BOD* para evitar o uso do controlador, quando a alimentação do microcontrolador estiver abaixo do nível estipulado pela programação interna do mesmo.

Existem ainda modelos específicos com controlador de LCD embutido ou geradores especiais de PWM para controle de motores.

* Brown-out Detector

3.2.1.3 AT90CAN128

Este modelo específico apresenta as seguintes características[2]:

- 128 KB de memória Flash
- 4 KB de EEPROM
- 4 KB de SRAM, interna, podendo ser adicionado até 64 KB de memória externa
- Interface JTAG para programação e depuração
- Controlador CAN (2.0A e 2.0B)
- 4 Timers, dois deles com 8 bits de precisão e o restante com 16 bit de precisão, todos com divisores de clock com 10 bits de precisão
- Interface SPI, TWI e duas USART
- Velocidade de até 8MHz utilizando 2,7 volts para alimentação, e até 16MHz utilizando 4,5 volts (voltagem de operação: 2,7 até 5,5 volts)

Comparando com um ATmega128, o AT90CAN128 sofreu algumas alterações no mapeamento de registradores. De acordo com o manual de migração de plataformas da Atmel [30], a estrutura aplicada melhorou a coerência entre registradores e função designada para o mesmo, e será utilizada nos novos controladores desenvolvidos. A maior parte da mudança foi concentrada nos registradores dos timers (timer 0 do ATmega128 é o timer 2 do AT90CAN128 e vice versa), conversores analógico digital (aquisição de dados pode ser efetuada por um evento de timer, do comparador analógico ou interrupção externa) e clocks(foi adicionada a opção de direcionar a clock interno para um dos pinos do controlador).

3.2.1.4 Hardware Complementares

Como o AT90CAN128 não é disponibilizado no encapsulamento DIP[*] utilizado pela plataforma de desenvolvimento STK500, foi necessária a utilização de uma expansão, a STK501, ela possui um soquete ZIF[**] onde podemos utilizar um AVR com encapsulamento TQFP[***].

Para conectar, efetivamente, o controlador ao barramento CAN, a ATMEL fornece o módulo ATADAPCAN01, ele possui um circuito integrado responsável pela interface entre o controlador e o barramento CAN, respeitando as normas especificadas.

---

[*] Dual in-line package
[**] Zero insertion force
[***] Thin quad flat pack

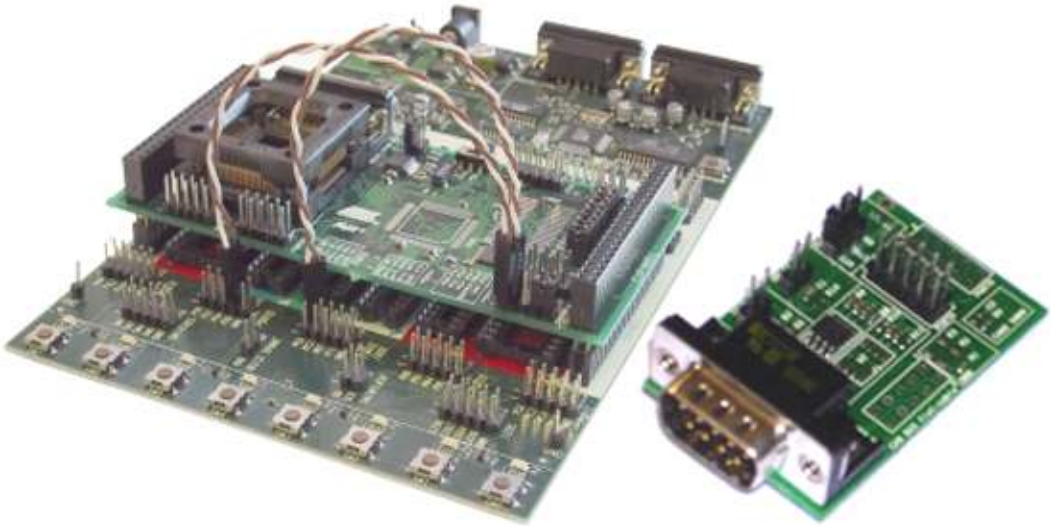


Figura 3.8: STK500+501 [28] e ATADAPCAN01

3.3 Implementação do Mediador

Com o estudo e análise do sistema operacional base e do hardware onde foi planejada a primeira implementação do mediador CAN, foi obtido o seguinte diagrama:

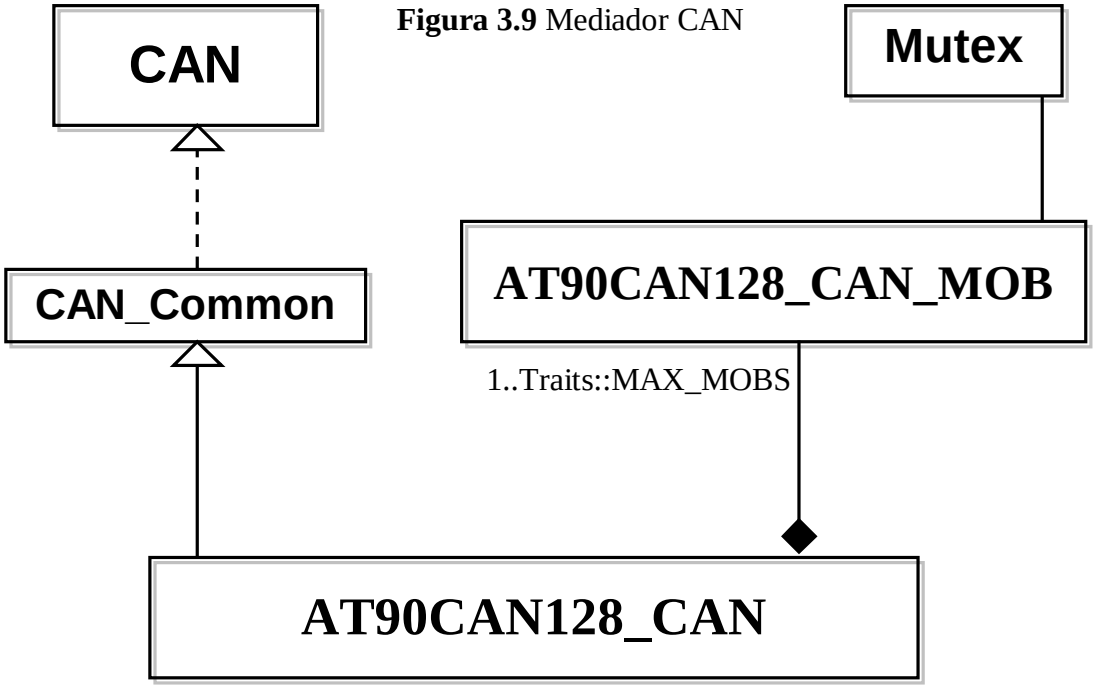


Figura 3.9 Mediador CAN

3.3.1 Considerações Associadas ao Controlador CAN do AVR

Como CAN possui uma especificação aberta e definida, grande parte das implementações, incluindo a do AVR, segue uma linha definida de informações sobre erros, transmissão de dados, entre outros. Contudo, a maneira como serão armazenadas as mensagens recebidas ou a serem transmitidas é de livre escolha do fabricante.

No AVR foi adotado o sistema de *mailbox* (caixa de correio). Este sistema de armazenamento de mensagens utiliza o conceito onde existe um objeto de mensagem com alguns registradores, associados a erros, estados e configurações daquele objeto. Este sistema foi utilizado para diminuir a sobrecarga do microcontrolador agregado ao tempo de recuperação da mensagem, assim mesmo, sendo um controlador com recursos limitados, ele consegue trabalhar na velocidade máxima da especificação CAN. Porém, este sistema pode levar a perda de mensagens transmitidas no barramento, caso o software não consiga preparar um objeto de mensagem para recepção em tempo hábil. Uma fragilidade desta implementação no AVR acontece no momento de uso dos objetos de mensagem, já as mensagens são multiplexadas. Temos ainda um sistema de prioridade onde, caso dois objetos estiverem marcados para transmissão, o de maior prioridade será enviado antes. A prioridade diminui com o aumento do índice do objeto.

A estrutura básica de controle do sistema é baseada em um registrador de seleção de página e outro indicando o índice de maior prioridade. Registradores de estado, identificador, máscara e do buffer de dados são dependentes de cada objeto.

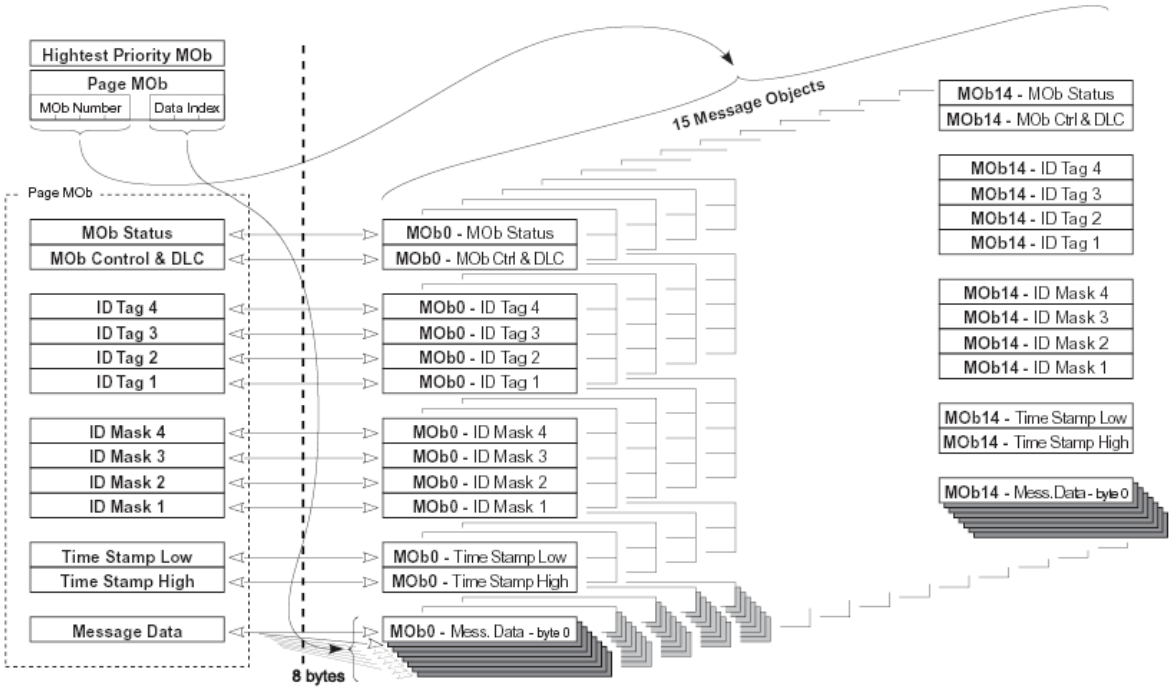


Figura 3.10: Representação do sistema de caixa de correio no AVR [2]

3.3.2 Interface com o Sistema

A interface visível para o software ou abstrações do sistema operacional é mantida na classe AT90CAN128_CAN, esta interface, deve ser conservada nas implementações de outras plataformas, conforme descrito em [33]. Em vista disto, foi analisada a implementação FlexCan [31] utilizada nos microcontroladores ColdFire fabricados pela Freescale. O FlexCan utiliza um sistema de armazenamento de mensagem semelhante ao AVR, porém, ao contrário do AVR, cada mensagem possui um endereço de memória separado. Assim, podemos aplicar o mesmo conceito de implementação seguido para o AVR, com uma classe modelando os buffers de mensagens do sistema. Mesmo para arquiteturas onde é utilizada uma implementação como a MSCAN, também utilizada em microcontrolares e DSPs* da Freescale, baseada em um buffer FIFO**, pode ser utilizada a mesma interface e, ainda, mantendo uma classe modelando o objeto de mensagem que fica disponível na fila.

3.3.3 Otimizações

A implementação apresentada pode ter o uso de memória otimizado, caso necessário, descartando-se os buffers dos identificadores e máscaras utilizados para as mensagens. Aplicando isto no código, podemos modelar ao invés das mensagens, somente o ponto de acesso a elas, conforme pode ser observado na figura 3.10, liberando cerca de 500 bytes na memória do sistema. Esta otimização é vital em sistemas onde o consumo de memória é crítico, e considerando somente o uso da pequena memória disponível internamente no microcontrolador.

3.4 Mapeando CAN para a Abstração de Rede no EPOS

Para completar o conceito AOSD, o mediador CAN deve ter a funcionalidade disponibilizada através de uma abstração de rede, a qual suporta diversos modelos de redes, de forma transparente para a aplicação. Desta forma, tornar-se fácil a utilização das diversas redes suportadas pelo sistema operacional no software usuário. Ele não precisará saber em que tipo de rede está operando para funcionar corretamente.

---

* Digital Signal Processor
** First In, First Out

3.4.1 Mapeamento de Endereços

Como já descrito anteriormente, uma rede CAN não utiliza explicitamente endereços fonte/destino, como ocorre na ethernet, para o envio de mensagens, é implementado um identificador que será enviado na mensagem. Esta mensagem é transmitida para todos os nós conectados na rede, além disso, podem existir diversos receptores para a mensagem transmitida.

Entretanto, para implementar uma abstração comum entre os diversos tipos de redes, devemos possuir um sistema comum de endereçamento dos nós entre elas.

Analisando as redes CAN, Profibus, Ethernet e derivadas, podemos classificar o sistema de endereçamento em dois modelos. O primeiro é o modelo utilizado pela rede CAN, onde mensagens identificadas são enviadas via broadcast. No segundo temos um sistema onde as mensagens contem o endereço fonte e destino, e a forma de envio pode ser feita através de broadcast, multicast ou ponto a ponto.

Considerando estes métodos de endereçamento, podemos implementar o sistema utilizado no CAN para as outras redes, ou criar um sistema de endereçamento para o CAN utilizando o campo de identificação, o qual se aproxima mais das redes restantes para que um modelo comum de endereçamento fonte/destino possa ser implementado.

Para esta escolha podemos considerar que o sistema da rede CAN é quantitativamente menos utilizado, sendo mais conveniente criar um mapeamento de um endereçamento fonte/destino para a CAN ao invés de criar uma nova implementação para cada uma das outras redes. Um problema decisivo poderia ser a performance, caso uma rede suporte somente comunicação ponto a ponto, a sobrecarga de realizar um broadcast pode ser muito grande. Outro fator que favorece esta decisão está no sistema de arbítrio na rede CAN. Se dois nós transmitirem uma mensagem com identificador igual, ao mesmo tempo (como o identificador é o mesmo nenhum dos nós irá perder o controle sobre o barramento durante a transmissão), porém com dados diferentes, será gerado um conflito de transmissão da mensagem entre os dois nós transmissores, este conflito, provavelmente, não pode ser resolvido e irá gerar um consumo indesejado de recursos da rede e dos nós durante algum tempo. Para resolver este problema pode ser utilizado um tag único para cada nó [35] no identificador da mensagem, o mesmo serve como um endereço.

Este mapeamento é satisfatório para cenários onde os nós em uso apresentam processamento e memória adequados, porém, caso os nós sejam compostos na maior parte de sensores “burros”, de baixo custo, o mapeamento fonte/destino pode ser

proibitivo. Sistemas onde é relevante somente a fonte ou tipo de mensagem (conforme será visto na seção 4), o endereço destino é uma sobrecarga desnecessária. Neste cenário, é interessante utilizar somente um endereço para identificação do sensor (resolvendo o problema da arbitragem), e espalhar a mensagem para todos os nós da rede como é feito originalmente nas redes CAN.

Ainda temos dois mapeamentos, cada um adequado para determinadas situações e inadequado para outras. Analisando mais profundamente, optar por implementar um mapeamento configurável é interessante. Caso o sistema esteja configurado para não utilizar endereço destino, o processamento do pacote deve depender somente do endereço fonte/prioridade. A implementação do envio da mensagem, sem o destinatário, em redes onde é necessário especificar um destino, será via broadcast/multicast, estes modos geralmente já são implementados em redes desde tipo.

3.4.2 Mapeamento das Propriedades de transmissão

As propriedades de transmissão relativas a cada tipo de rede também devem ser consideradas para implementar a abstração, especialmente na rede CAN, já que a prioridade de transmissão no barramento é vinculada ao identificador da mensagem.

A prioridade da mensagem será considerada o único atributo do mapeamento, já que está presente nos modelos analisados.

Em redes CAN, a prioridade de uma mensagem decresce com o aumento do identificador, possibilitando uma enorme quantidade de prioridades. Redes Profibus possuem mensagens de baixa prioridade e alta prioridade. Nas redes Ethernet onde o padrão 802.1D [36] é implementado, é possível o uso de até oito níveis de prioridades (0 até 7, sendo 7, a maior prioridade).

Para um exemplo escolhemos ao acaso um mapeamento onde as mensagens poderão ser classificadas em quatro níveis de prioridade. Nível baixo, normal, médio e alto. Em uma rede Profibus, estes atributos seriam mapeados em prioridade baixa, baixa, alta, alta respectivamente. Já para a rede Ethernet, um mapeamento possível é prioridade 0, 2, 5 e 7. Na rede CAN, 2 bits do identificador mapeiam diretamente a prioridade requisitada.

Na prática, o número de prioridades deve ser grande o suficiente para permitir um escalonamento eficiente caso um sistema tempo real seja implementado. Assim o mapeamento das prioridades deverá ser feito dinamicamente conforme a quantidade de níveis oferecidos pela rede utilizada.

3.4.3 Mapeamento para uma Abstração com Suporte Ethernet e CAN

Para a Ethernet, o mapeamento de endereço é direto, para ambas as configurações citadas na seção 3.4.1. Na rede CAN foi escolhido utilizar os 7 bits menos significativos para o endereço fonte, caso configurado, os 7 bits seguintes para o endereço destino, devido ao limite físico do barramento e funcionamento do sistema de prioridade.

Para a prioridade, temos 3 bits na Ethernet e 22 bits/15 bits no CAN (considerando a versão 2.0B). Na abstração, temos um sistema de prioridades onde a base é a maior quantidade de níveis entre os protocolos suportados, neste caso 22 bits, que são quantizados devidamente para cada tipo rede suportada.

4 Comunicação entre redes Heterogêneas

A comunicação através de diversos tipos de redes pode ser indispensável em alguns tipos de aplicações, por exemplo, na criação de uma rede de sensores onde existam diferentes grupos, separados fisicamente, onde os integrantes de cada grupo comunicam-se via CAN, e cada grupo realiza a comunicação, utilizando um meio como WiFi, Bluetooth ou outro sistema de comunicação sem fio (“WAN de CANs” [34]). Podemos necessitar, nesse sistema, uma conservação dos atributos das mensagens, como tempo de entrega, quem enviou, entre outros, os quais não estão presentes na parte sem fio da comunicação. Também uma autonomia entre os componentes participantes, sem eles ficarem vinculados ao meio de comunicação.

Observando isso, foi criado o conceito de canais de eventos para CAN [34], aqui será introduzida a idéia básica do funcionamento de um protocolo baseado em eventos.

4.1 Funcionamento

Para o protocolo alcançar seu objetivo primário, a comunicação entre redes heterogêneas, uma propriedade do mesmo é ser um protocolo de alto nível, ou seja, a aplicação não terá uma visão sobre a rede na qual está operando.

Como o termo “canais de eventos” sugere, uma mensagem passa a ser um evento. O canal será o responsável por assegurar as restrições temporais e confiabilidade de transmissão onde os eventos irão trafegar. Este sistema de canais é baseado em nós, produtores de eventos e nós assinantes, receptores dos eventos [34]. Este modelo de comunicação é chamado de “*publisher/subscriber*”, nele não existe o conceito de endereçamento para comunicação um para um, existe, no sistema que é controlado por um *middleware*, um sistema de distribuição onde os assinantes de um canal recebem as informações publicadas pelos produtores de eventos associados ao mesmo canal, realizando assim uma comunicação muitos para muitos, vital para sistemas de distribuição de eventos em tempo real [35].

Como os eventos são gerados espontaneamente pelos dispositivos, sem uma forma de sincronização específica, a sincronização será realizada pelo canal utilizado para transmissão. A característica que transforma uma simples mensagem em um evento

é a transmissão não só do conteúdo da mensagem, mas também do contexto onde ela foi produzida e dos atributos de transmissão requeridos por ela [34].

A descrição de canais e de eventos, conforme [34], é da seguinte forma:

canal := <assunto, atributos de transmissão, participantes>

evento := <assunto, contexto, atributos de transmissão, conteúdo>

Onde o assunto representa o conteúdo relacionado ao canal/evento. O atributo de transmissão, entretanto, é diferenciado entre canal e evento. No canal, ele será utilizado para representar de maneira global a qualidade e a forma de transmissão utilizada por este para acesso à rede. Para o evento, irá representar o tempo de validade do mesmo no sistema. Participantes são os nós assinantes e publicadores de um canal. O contexto de um evento pode identificar atributos como forma, tempo e localização da geração do evento.

4.2 Estrutura para implementação

Conforme [35], o modelo “publisher/subscriber” pode ser implementado através de um *middleware*, baseado na seguinte arquitetura:

ECH, *event channel handler*, é a estrutura implementada localmente no controlador para cuidar da comunicação entre a aplicação e a rede. Ela realiza as tarefas de escalonar, rotear e filtrar mensagens.

Uma das possibilidades de mapeamento do identificador de 29 bits do CAN, utilizado pelo ECH para cumprir as funções requeridas, é a seguinte: 8 bits para a prioridade, 7 bits para o identificador do nó e 14 bits para o identificador (*tag*) do assunto. Sendo os 8 bits iniciais usados para realizar escalonamento dinâmico das mensagens, e os bits restantes, o escalonamento estático.

ECB, *event channel broker*, estrutura responsável pelos mapeamentos de assuntos aos respectivos identificadores (tags) e distribuição de identificadores únicos para cada nó ligado na rede, sendo esta última função necessária para evitar conflitos de arbitragem no barramento, caso uma mensagem de mesmo identificador seja transmitida por dois nodos diferentes, conforme citado anteriormente.

Um nó que queira transmitir na rede deve, antes de tudo, consultar o ECB e requisitar uma configuração para adquirir o identificador. Após isto, ele poderá

requisitar ao ECB o tag mapeado para o assunto sobre o qual o nó deseja publicar eventos.

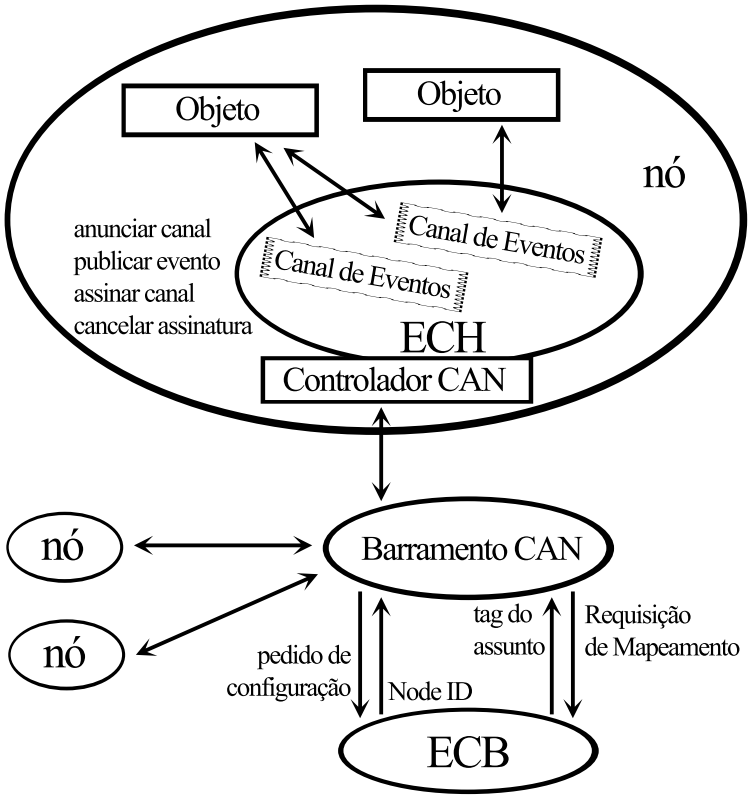


Figura 4.1: Estrutura da arquitetura “publisher/subscriber” [35]

Para utilizar esta arquitetura no EPOS, abstrações para o ECB e o EBH podem ser implementadas. Elas podem utilizar a abstração de rede citada na seção 3.4 para o acesso à rede (transmissão, recepção e mapeamento dos atributos de transmissão), tornando o sistema pronto para ser utilizado entre as várias redes suportadas.

Uma pequena mudança necessária nas implementações entre os diferentes protocolos é necessária para o mapeamento dos tags e da recuperação da prioridade original (se necessário), caso a mensagem passe por redes onde a quantidade de prioridades suportada é menor do que a rede fonte/destino. Entretanto, estas diferenças são transparentes para softwares usuário.

5 Conclusão

Com a pesquisa realizada sobre redes CAN, foi possível avaliar com mais clareza os recursos disponibilizados pela rede para melhor utilizá-los. Durante o período de pesquisa, também, foi constatada a existência de uma grande quantidade de informações disponíveis tanto no meio acadêmico quanto no meio industrial sobre redes CAN. O mesmo não foi verificado para outras redes, tornando visível o interesse na tecnologia por um grande grupo de pessoas, reforçando a citação feita na seção de motivações no início do trabalho.

Os objetivos propostos pelo trabalho foram alcançados com sucesso, embora com o desenvolvimento da abstração de rede, ainda em caráter inicial, possibilitando assim a realização de futuras aplicações e pesquisas utilizando CAN, com todo o suporte oferecido pelo sistema EPOS.

5.1 Trabalhos Futuros

Uma extensão para utilizar um sistema de comunicação disparado por tempo pode ser adicionada ao sistema. O microcontrolador utilizado para a implementação possui recursos de hardware para implementar este tipo de comunicação. Outra extensão possível é criar um algoritmo para auto-ajuste do nó com a taxa de transmissão utilizada na rede, tornando o sistema ainda mais flexível.

Para a implementação de protocolos mais complexos sobre redes CAN será indispensável possuir um analisador de tráfego na rede. Os microcontroladores disponíveis no LISHA podem ser facilmente programados para esta tarefa, sendo configurados como um nó estritamente receptor na rede CAN e repassando o trafego recebido para um computador onde os dados serão efetivamente analisados.

Futuramente, com a possível necessidade de poder de processamento para realização de tarefas mais complexas, poderá ser necessária a implementação do mediador em outras plataformas.

A pesquisa em redes de sensores, utilizando objetos sentientes, realizada no LISHA, pode explorar a comunicação baseada em eventos facilmente implementada

utilizando redes CAN, já que por definição, objetos sentientes são fracamente acoplados e comunicam-se através de eventos [37]. Aqui pode destacar-se uma implementação da comunicação entre redes heterogêneas citada neste trabalho.

Finalmente, temos pesquisas na área de comunicação em tempo real, indispensável para redes mais críticas, como as utilizadas em carros.

Referências

1. BOSCH, Robert. **CAN Specification Version 2.0**. Stuttgart, 1991.
2. Atmel Corporation. **AT90CAN128 Datasheet Rev E**. Dezembro 2004.
3. CiA. **Controller Area Network (CAN): an overview**. Disponível em: <http://www.can-cia.de/can/>. Acesso em: 21 novembro 2004.
4. Kvaser. **CAN (Controller Area Network)**. Disponível em: <http://www.kvaser.se/can/main.htm>. Acesso em: 21 novembro 2004.
5. NILSSON, Staffan. **Controller Area Network - CAN Information**. Disponível em: <http://www.algonet.se/~staffann/developer/CAN.htm>. Acesso em: 21 novembro 2004.
6. RUFINO, José. **An Overview of the Controller Area Network**. Braga, Janeiro 1997. Disponível em <http://pandora.ist.utl.pt/docs/pdf/ar/ciaBrg.pdf>. Acesso em: 25 novembro 2004.
7. KAISER, Jörg. **Event channels, an integration Concept for Predictable Interaction in Embedded Heterogenes Networks**. Proceedings in Workshop on Dependable Embedded Systems. Florianópolis, Outubro 2004.
8. DE MEDEIROS FRÖHLICH, A. A. **Application-Oriented Operating Systems**. Número 17. GMD Research Series. GMD - Forschungszentrum Informationstechnik. Sankt Augustin, Agosto 2001.
9. CiA. **Press Releases - CAN Sales Figures**. 1999.
10. LIVANI ALI, Mohammad; KAISER, Jörg; JIA, Weijia. **Scheduling hard and soft real-time communication in a controller area network.** Control Engineering Practice 7. Julho 1999.
11. Stock Flight Systems. **CANaerospace Interface Specification v1.6**. Disponível em: <http://www.stockflightsystems.com/canas_16.pdf>. Acesso em: 10 junho 2005.
12. A. Elfving, L.Stagnaro, A.Winton. **SMART-1: Key technologies and autonomy implementations**. Proceedings of 4th IAA Intl.Conf. on low-cost planetary missions-John Hopkins University. Maio 2000.
13. Casper G. Christensen Morten F. Christensen, Rasmus Corlin Kasper L. Jensen René B. Jensen, Søren Larsen. **Designing And Prototyping A Protocol For**

Communicating Transparently Between Can And On-Board Computer Applications On AAUSAT-II. Institute of Electronic Systems, Aalborg University.

14. Group 03gr731 Aalborg University. **Designing, prototyping, and testing of a flexible on board computer platform for pico-satellites**. 16a. SEMCON, 2003.
15. Hitex Development Tools. **Controller Area Network in Medical Applications**. Disponível em: <http://www.hitex.co.uk/newsletter0603/can/medicalcan.html>. Acesso em: 11 junho 2005.
16. ZELTWANGER, Holger. **Controller Area Network and CANopen in Medical Equipament**. Business Briefing: Medical Devide Manifacturing & Technology. 2002.
17. MOLIN, J. P. ; INAMASU, R. ; SARAIVA, A. M. ; SOUZA, R. V. **Os benefícios da padronização ISOBUS nas máquinas agrícolas**. Campo Aberto, Canoas, v. 82, p. 30 - 31, 01 maio 2005.
18. HARTWICH, Florian; BASSEMIR, Armin. **The Configuration of the CAN Bit Timing**. 6th International CAN Conference. Turin, Novembro 1999.
19. AS-Interface. **The System | Facts**. Disponível em: <http://www.as-interface.net/System/Facts>. Acesso em: 29 de agosto de 2005.
20. PROFIBUS. **PROFIBUS Profile for Process Automation**. Disponível em: <http://us.profibus.com/PA/PROFIBUSPA.pdf>. Acesso em 2 de setembro de 2005.
21. Synergic. **The Grid Connect Fieldbus Comparision Chart**. Disponível em: <http://www.synergetic.com/compare.htm>. Acesso em: 29 de agosto de 2005.
22. PROFIBUS. **PROFIBUS Technology and Application - System Description**. PROFIBUS Trade Organization. Outubro, 2002.
23. SANTOS, Max Mauro Dias; VASQUES, Francisco; STEMMER, Marcelo Ricardo. **Avaliação das propriedades temporais de duas redes de controle: CAN e PROFIBUS.** Acta Scientiarum. Maringá, 2003.
24. GARBEE, Bdale; GREEN, Chuck; JOHNSON, Lyle; MORACO, Stephen. **Doing More with Fewer Wires in the Harness: A New Approach to Spacecraft On-Board Command and Telemetry Interfacing**. The AMSAT Journal. Novembro/Dezembro 2003.
25. PARNELL, Karen. **Put the Right Bus in Your Car**. Xcell Journal. 2004.
26. FÄRBER, Georg. **Feldbussysteme - Oberseminar Prozeßrechentechnik**. Technische Universität München. 1994/95.

27. TENNENHOUSE, D. **Proactive Computing**. Communications of the ACM, v.43, n.5. Maio 2000.
28. Atmel. **STK501 User Guide**.
29. KURILIN, Aleksey; LAMBERT, Helen. **AVR - микроконтроллеры: развитие продолжается** (AVR – microcontrolers: development continues). Disponível em: <http://www.atmel.ru/Articles/Atmel30.htm>. Acesso em: 16 de setembro de 2005.
30. Atmel. **AVR096: Migrating from ATmega128 to AT90CAN128**.
31. Freescale. **MCF5282 ColdFire Microcontroller User's Manual**. Novembro 2004.
32. Motorola. **MSCAN Block Guide**. Julho de 2004.
33. POLPETA, Fauze Valério; DE MEDEIROS FRÖHLICH, A. A. **Hardware Mediators: A Portability Artifact for Component-based Systems.** In: Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of Lecture Notes in Computer Science. Aizu, Japan. Agosto 2004.
34. KAISER, Jörg. **An Event Model for Predictable Interaction of Smart Devices**. Workshop on Dependable Embedded Systems, in conjunction with the IEEE 22nd International Symposium on Reliable Distributed Systems. Florence, Italy. Outubro 2003
35. KAISER, Jörg; Mock, M. . **Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)**. In Proceedings of 2nd International Symposium on Object-Oriented Real-Time Distributed Computing. Saint Malo, France. 1999.
36. Institute of Electrical and Electronics Engineers. **IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks - IEEE Std 802.1Q**. Dezembro, 1998
37. FITZPATRICK, Adrian; BIEGEL, Gregory; CLARKE, Siobhán; CAHILL, Vinny. **Towards a Sentient Object Model**. Workshop on Engineering Context-Aware Object Oriented Systems and Environments. Seattle, USA. 2002.

Anexo A – Código Fonte

Arquivo: ./includes/can.h

```
// EPOS-- CAN Mediator Common Package

#ifndef __can_h
#define __can_h

#include <system/config.h>

__BEGIN_SYS

class CAN_Common
{
protected:
    CAN_Common() {}

public:

    enum
    {
        //can configuration
        MAX_MOBS      = 15, // AVR message objects 0..14
        REGISTER_MOBS = 8,  // mobs per register(CANIE/CANEN)
        MAX_DATASIZE  = 8,  // Message data size(bytes)

        //mob status return codes
        MOB_TX_COMPLETED = 0,
        MOB_RX_COMPLETED = 1,
        MOB_BUSY         = 2,
        MOB_DISABLED     = 3,

        //mob rx/rx_buffer types
        MOB_COMPARE_NO_BITS  = 0,
        MOB_COMPARE_IDE_BIT  = 1,
        MOB_COMPARE_RTR_BIT  = 2,
        MOB_COMPARE_ALL_BITS = 3,
```

```
        //mob error return codes(power of 2)
        MOB_NO_ERROR    = 0,
        MOB_ERROR_DLC   = 1,
        MOB_ERROR_BIT   = 2,
        MOB_ERROR_STUFF = 4,
        MOB_ERROR_CRC   = 8,
        MOB_ERROR_FORM  = 16,
        MOB_ERROR_ACK   = 32,

        //general function return codes
        OK              = -1,
        ERROR           = -2,
        NO_DATA         = -3,
        NOT_FREE        = -4,
        INVALID_ID      = -5,
        INVALID_SIZE    = -6,
        INVALID_INDEX   = -7,
        NO_PRIORITY_MOB = -8,

        //canstatus return codes, must be power of 2(can return more
than one at a time)
        CAN_OVERLOAD_TX = 1,   // sending overload frame
        CAN_TXING       = 2,   // tx busy
        CAN_RXING       = 4,   // rx busy
        CAN_BUSOFF      = 8,   // bus off mode
        CAN_ENABLED     = 16,  // can enabled
        CAN_ERROR_PASV  = 32   // error passive mode
    };
};

__END_SYS

#include __HEADER_MACH(can)

#endif
```

Arquivo: ./includes/mach/at90can128/can.h

```
// EPOS-- AT90CAN128_CAN Declarations

#ifndef __at90can128_can_h
#define __at90can128_can_h

#include <mutex.h>
#include <mach/at90can128/memory_map.h>
#include <can.h>

__BEGIN_SYS

class AT90CAN128_CAN: public CAN_Common
{
private:
    typedef Traits<AT90CAN128_CAN> _Traits;
    static const Type_Id TYPE = Type<AT90CAN128_CAN>::TYPE;

    static const unsigned long CAN_A_BITS_ID    = 0x7FF;
    static const unsigned long CAN_B_BITS_ID    = 0x1FFFFFFF;

    enum
    {
        CANHPMOB = 0xEC - 0x20, // high priority mob
        CANGCON = 0xD8 - 0x20,
        CANGSTA = 0xD9 - 0x20,
        CANGIT  = 0xDA - 0x20,
        CANGIE  = 0xDB - 0x20,
        CANEN2  = 0xDC - 0x20,
        CANEN1  = 0xDD - 0x20,
        CANIE2  = 0xDE - 0x20,
        CANIE1  = 0xDF - 0x20,
        CANSIT2 = 0xE0 - 0x20,
        CANSIT1 = 0xE1 - 0x20,
        CANBT1  = 0xE2 - 0x20,
        CANBT2  = 0xE3 - 0x20,
        CANBT3  = 0xE4 - 0x20,
        CANTCON = 0xE5 - 0x20,
        CANTIM  = 0xE6 - 0x20,
        CANTTC  = 0xE8 - 0x20,
        CANTEC  = 0xEA - 0x20,
```

```
    CANREC  = 0xEB - 0x20
};

enum
{
    CANGCON_ABRQ     = 7,
    CANGCON_OVRQ     = 6,
    CANGCON_TTC      = 5,
    CANGCON_SYNCTTC  = 4,
    CANGCON_LISTEN   = 3,
    CANGCON_ENA      = 1,
    CANGCON_SWRES    = 0,

    CANGSTA_OVFG     = 6,
    CANGSTA_TXBSY    = 4,
    CANGSTA_RXBSY    = 3,
    CANGSTA_ENFG     = 2,
    CANGSTA_BOFF     = 1,
    CANGSTA_ERRP     = 0,

    CANGIT_CANIT     = 7,
    CANGIT_BOFFIT    = 6,
    CANGIT_OVRTIM    = 5,
    CANGIT_BXOK      = 4,
    CANGIT_SERG      = 3,
    CANGIT_CERG      = 2,
    CANGIT_FERG      = 1,
    CANGIT_AERG      = 0,
    CANGIT_ERRMASK   = 0x0F,
    CANGIT_INTMASK   = 0x5F,

    CANGIE_ENIT      = 7,
    CANGIE_ENBOFF    = 6,
    CANGIE_ENRX      = 5,
    CANGIE_ENTX      = 4,
    CANGIE_ENERMOB   = 3,
    CANGIE_ENBUF     = 2,
    CANGIE_ENERG     = 1,
    CANGIE_ENOVRT    = 0,

    CANBT1_BRP       = 1,
```

```
        CANBT1_BRPMASK   = 0x7E,

        CANBT2_SJW       = 5,
        CANBT2_SJWMASK   = 0x60,
        CANBT2_PRS       = 1,
        CANBT2_PRSMASK   = 0x0E,

        CANBT3_PHS2      = 4,
        CANBT3_PHS2MASK  = 0x70,
        CANBT3_PHS1      = 1,
        CANBT3_PHS1MASK  = 0x0E,
        CANBT3_SMP       = 0,
        CANBT3_SMPMASK   = 0x01,

        CANHPMOB_HPMOB      = 4,
        CANHPMOB_HPMOBMASK  = 0xF0
    };

   class AT90CAN128_CAN_MOB
   {
   private:
      typedef Traits<AT90CAN128_CAN> _Traits;
      static const Type_Id TYPE = Type<AT90CAN128_CAN>::TYPE;

      union object_id
      {
            unsigned long id;
            unsigned char b[4];
      };

      enum
      {
            CANPAGE  = 0xED - 0x20, // page(mob selector) - default set
to autoincrement pointer on write/read op
            CANSTMOB = 0xEE - 0x20, // mob status
            CANCDMOB = 0xEF - 0x20, // mob control and dlc
            CANIDT4  = 0xF0 - 0x20, // message id tags
            CANIDT3  = 0xF1 - 0x20,
            CANIDT2  = 0xF2 - 0x20,
            CANIDT1  = 0xF3 - 0x20,
            CANIDM4  = 0xF4 - 0x20, // message masks
```

```
	CANIDM3  = 0xF5 - 0x20,
	CANIDM2  = 0xF6 - 0x20,
	CANIDM1  = 0xF7 - 0x20,
	CANSTML  = 0xF8 - 0x20, // time stamp low
	CANSTMH  = 0xF9 - 0x20, // time stamp high
	CANMSG   = 0xFA - 0x20  // data
};


enum
{
	CANSTMOB_DLCW       = 7,
	CANSTMOB_TXOK       = 6,
	CANSTMOB_RXOK       = 5,
	CANSTMOB_BERR       = 4,
	CANSTMOB_SERR       = 3,
	CANSTMOB_CERR       = 2,
	CANSTMOB_FERR       = 1,
	CANSTMOB_AERR       = 0,
	CANSTMOB_ERRMASK    = 0x1F,
	CANSTMOB_INTMASK    = 0x7F,

	CANCDMOB_CONMOB     = 6,
	CANCDMOB_RPLY       = 5,
	CANCDMOB_IDE        = 4,
	CANCDMOB_CONMOBMASK = 0xC0,
	CANCDMOB_DLCMASK    = 0x0F,
	ENABLE_TX           = 1, // 0b01
	ENABLE_RX           = 2, // 0b10
	ENABLE_BUFFER       = 3, // 0b11

	CANPAGE_MOB            = 4,
	CANPAGE_NOINCREMENT = 8,

	CANIDT4_RTRTAG      = 2,
	CANIDT4_RB1TAG      = 1,
	CANIDT4_RB0TAG      = 0,

	CANIDM4_RTRMASK     = 2,
	CANIDM4_IDEMASK     = 0
};
```

```
   public:
      AT90CAN128_CAN_MOB() : _mob_index(0), _extended(true)
      {
            _mob_id.id = 0; _mob_mask.id = 0;
            clear();
      }
      ~AT90CAN128_CAN_MOB() {}

      int status();
      int error();
      void clear();

      void mob_index( int index ) { _mob_index = index; }

      // setting ID only takes effect when mob is set to tx/rx
      void id(unsigned long ID) { _mob_id.id = ID; }
      unsigned long id();
      bool is_extended() { return _extended; }

      // setting mask only takes effect when the mode is set to rx
      void mask(unsigned long mask) { _mob_mask.id = mask; }

      // autoreply mode is activated by rtr + reply valid flag... not
implemented!
      void rx_mode(bool extended, int max_size, bool
compare_remote_bit, bool compare_extended_bit);
      void buffer_mode(bool extended, int max_size, bool
compare_remote_bit, bool compare_extended_bit);
      void disable();

      void tx_request(bool extended);
      void tx_data(unsigned char *data, unsigned char size, bool
extended);

      int retrieve_data(unsigned char *data);

      static void clear_all();

   private:
```

```
      typedef AVR8::Reg8 Reg8;

      void prepare_rx(bool extended, int max_size, bool compare_rtr,
bool compare_ide);
      void prepare_id(bool extended);

      static void page(int mob); // Caution! static mutex locking
function, call release_page to unlock!
      static void page_release() { Page_Mutex.unlock(); }
      static int page() { return canpage() >> CANPAGE_MOB; }

      static void canmsg(Reg8 Value) { AVR8::out8(CANMSG, Value); }
      static Reg8 canmsg() { return AVR8::in8(CANMSG); }
      static void canpage(Reg8 Value) { AVR8::out8(CANPAGE, Value); }
      static Reg8 canpage() { return AVR8::in8(CANPAGE); }
      static void canstmob(Reg8 Value) { AVR8::out8(CANSTMOB, Value);
}
      static Reg8 canstmob() { return AVR8::in8(CANSTMOB); }
      static void cancdmob(Reg8 Value) { AVR8::out8(CANCDMOB, Value);
}
      static Reg8 cancdmob() { return AVR8::in8(CANCDMOB); }
      static void canidm1(Reg8 Value) { AVR8::out8(CANIDM1, Value); }
      static Reg8 canidm1() { return AVR8::in8(CANIDM1); }
      static void canidm2(Reg8 Value) { AVR8::out8(CANIDM2, Value); }
      static Reg8 canidm2() { return AVR8::in8(CANIDM2); }
      static void canidm3(Reg8 Value) { AVR8::out8(CANIDM3, Value); }
      static Reg8 canidm3() { return AVR8::in8(CANIDM3); }
      static void canidm4(Reg8 Value) { AVR8::out8(CANIDM4, Value); }
      static Reg8 canidm4() { return AVR8::in8(CANIDM4); }
      static void canidt1(Reg8 Value) { AVR8::out8(CANIDT1, Value); }
      static Reg8 canidt1() { return AVR8::in8(CANIDT1); }
      static void canidt2(Reg8 Value) { AVR8::out8(CANIDT2, Value); }
      static Reg8 canidt2() { return AVR8::in8(CANIDT2); }
      static void canidt3(Reg8 Value) { AVR8::out8(CANIDT3, Value); }
      static Reg8 canidt3() { return AVR8::in8(CANIDT3); }
      static void canidt4(Reg8 Value) { AVR8::out8(CANIDT4, Value); }
      static Reg8 canidt4() { return AVR8::in8(CANIDT4); }

   private:
      int _mob_index;
      object_id _mob_id;
```

```
      object_id _mob_mask;
      bool _extended;

      static Mutex Page_Mutex;
   };

public:
    AT90CAN128_CAN() { configure(125000, true); }

    AT90CAN128_CAN(long br, bool autostart = true) { configure(br,
autostart); }

    // values int time quantum, min..max:
    //             1..32        1..8              1..8
1..8                1..4
    AT90CAN128_CAN(int brp = 0, int prop_seg = 0, int phase_seg1 = 0,
int phase_seg2 = 0, int sjw = 0, bool autostart = true)
    {
        configure(-1, autostart, brp, prop_seg, phase_seg1,
phase_seg2, sjw);
    }

    ~AT90CAN128_CAN() { stop(); }

    inline int mob_select() { return selected_mob; }
    int mob_select(int index);

    bool mob_is_free();

    inline int mob_status() { return mobs[selected_mob].status(); }
    inline int mob_error() { return mobs[selected_mob].error(); }
    inline void mob_clear() { mobs[selected_mob].clear(); }
    inline void mob_disable() { mobs[selected_mob].disable(); }

    int mob_rx_mode(unsigned long ID, unsigned long mask = 0,
                    int max_data_size = _Traits::MAX_DATASIZE,
                    bool extended = true, int type =
MOB_COMPARE_ALL_BITS);
    int mob_buffer_mode(unsigned long ID, unsigned long mask = 0,
                        int max_data_size = _Traits::MAX_DATASIZE,
```

```
                        bool extended = true, int type =
MOB_COMPARE_ALL_BITS);
    int mob_send_data(unsigned char *data, unsigned long ID, unsigned
char size, bool extended = true);
    // use: buffer size must be at least equal than mob data size(max
= MAX_DATASIZE)
    int mob_retrieve_data(unsigned char *data, unsigned long *ID, bool
*extended);
   int mob_request_data(unsigned long ID, bool extended);

// -- below goes MOB independent code

    int high_prio_mob();//return a mob with the interrupt flag set and
of highest priority

    void put_overload_request() //the overload frame will be sent upon
next rx
    {
        cangcon(cangcon() | CANGCON_OVRQ);
    }

    inline int rx_error_count() { return canrec(); }
    inline int tx_error_count() { return cantec(); }

    int status();

    long baud_rate(long baudrate);
    long baud_rate(int brp, int prop_seg, int phase_seg1, int
phase_seg2, int sjw);
    inline long baud_rate() { return _baud_rate; }

    void start(bool wait = true, bool listenonly = false)
    {
        unsigned char conval = 1<<CANGCON_ENA;
        if(listenonly)
            conval |= 1<<CANGCON_LISTEN;
        cangcon(cangcon() | conval);
        if(wait)
            while(!(cangsta() & (1<<CANGSTA_ENFG))) {}
    }
    void stop(bool wait = true)
```

```
    {
        cangcon(cangcon() & (~(1<<CANGCON_ENA)));
        if(wait)
            while(cangsta() & (1<<CANGSTA_ENFG)) {}
    }

    static int init(System_Info *si);

private:

    typedef AVR8::Reg8 Reg8;

    void configure(long baudrate, bool autostart,
                   int brp = 0, int prop_seg = 0, int phase_seg1 = 0,
int phase_seg2 = 0, int sjw = 0)
    {
        if(autostart)
            stop(); //force stop if already running...

        if( baudrate != -1 )
            baud_rate(baudrate);
        else
            baud_rate(brp, prop_seg, phase_seg1, phase_seg2, sjw);

       AT90CAN128_CAN_MOB::clear_all();
       mob_select(0);
        if(autostart)
            start();
    }

    inline bool check_id(bool extended_id, unsigned long ID)
    {
        if((!extended_id && (ID > CAN_A_BITS_ID)) || (extended_id &&
(ID > CAN_B_BITS_ID)))
            return false;

        return true;
    }

    inline static bool mob_is_free(int mob)
    {
```

```
        if(mob < _Traits::REGISTER_MOBS)
            return !(canen2() & (1<<mob));
        else
            return !(canen1() & (1<<(mob-_Traits::REGISTER_MOBS)));
    }

    static void canhpmob(Reg8 Value) { AVR8::out8(CANHPMOB, Value); }
    static Reg8 canhpmob() { return AVR8::in8(CANHPMOB); }
    static void cangcon(Reg8 Value) { AVR8::out8(CANGCON, Value); }
    static Reg8 cangcon() { return AVR8::in8(CANGCON); }
    static void cangsta(Reg8 Value) { AVR8::out8(CANGSTA, Value); }
    static Reg8 cangsta() { return AVR8::in8(CANGSTA); }
    static Reg8 cantec() { return AVR8::in8(CANTEC); }
    static Reg8 canrec() { return AVR8::in8(CANREC); }
    static void canen2(Reg8 Value) { AVR8::out8(CANEN2, Value); }
    static Reg8 canen2() { return AVR8::in8(CANEN2); }
    static void canen1(Reg8 Value) { AVR8::out8(CANEN1, Value); }
    static Reg8 canen1() { return AVR8::in8(CANEN1); }
    static void canbt1(Reg8 Value) { AVR8::out8(CANBT1, Value); }
    static Reg8 canbt1() { return AVR8::in8(CANBT1); }
    static void canbt2(Reg8 Value) { AVR8::out8(CANBT2, Value); }
    static Reg8 canbt2() { return AVR8::in8(CANBT2); }
    static void canbt3(Reg8 Value) { AVR8::out8(CANBT3, Value); }
    static Reg8 canbt3() { return AVR8::in8(CANBT3); }

private:
    int selected_mob;
    long _baud_rate;

    static AT90CAN128_CAN_MOB mobs[_Traits::MAX_MOBS];
};

typedef AT90CAN128_CAN CAN;

__END_SYS

#endif
```

Arquivo: ./src/mach/at90can128/can.cc

```
// EPOS-- AT90CAN128_CAN Implementation


#include <mach/at90can128/can.h>

__BEGIN_SYS

Mutex AT90CAN128_CAN::AT90CAN128_CAN_MOB::Page_Mutex;
AT90CAN128_CAN::AT90CAN128_CAN_MOB
AT90CAN128_CAN::mobs[_Traits::MAX_MOBS];

int AT90CAN128_CAN::AT90CAN128_CAN_MOB::status()
{
    unsigned char stt, scd;
    int ret;
    page(_mob_index);
    stt = canstmob();
    scd = cancdmob();
    page_release();

    if(stt & (1<<CANSTMOB_TXOK))
        ret = MOB_TX_COMPLETED;
    else if(stt & (1<<CANSTMOB_RXOK))
        ret = MOB_RX_COMPLETED;
    else
    {
        if( ( scd & CANCDMOB_CONMOBMASK ) == 0)
            ret = MOB_DISABLED;
        else
            ret = MOB_BUSY;
    }

    return ret;
}

int AT90CAN128_CAN::AT90CAN128_CAN_MOB::error()
{
    unsigned char stt;
    int ret = MOB_NO_ERROR;
    page(_mob_index);
```

```
    stt = canstmob();
    page_release();

    if(stt & CANSTMOB_DLCW)
        ret |= MOB_ERROR_DLC;
    if(stt & CANSTMOB_BERR)
        ret |= MOB_ERROR_BIT;
    if(stt & CANSTMOB_SERR)
        ret |= MOB_ERROR_STUFF;
    if(stt & CANSTMOB_CERR)
        ret |= MOB_ERROR_CRC;
    if(stt & CANSTMOB_FERR)
        ret |= MOB_ERROR_FORM;
    if(stt & CANSTMOB_AERR)
        ret |= MOB_ERROR_ACK;

    return ret;
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::clear()
{
    page(_mob_index);
    canstmob(0);
    cancdmob(0);
    canidt4(0);
    canidt3(0);
    canidt2(0);
    canidt1(0);
    canidm4(0);
    canidm3(0);
    canidm2(0);
    canidm1(0);
    for(int i = 0; i < _Traits::MAX_DATASIZE; i++)
        canmsg(0);
    page_release();
}

unsigned long AT90CAN128_CAN::AT90CAN128_CAN_MOB::id()
{
    page(_mob_index);
    if( cancdmob() & (1<<CANCDMOB_IDE) )
```

```
    {
        _extended = true;
        _mob_id.b[3] = canidt1() >> 3;
        _mob_id.b[2] = (canidt1() << 5) | (canidt2() >> 3);
        _mob_id.b[1] = (canidt2() << 5) | (canidt3() >> 3);
        _mob_id.b[0] = (canidt3() << 5) | (canidt4() >> 3);
    }
    else
    {
        _mob_id.b[2] = _mob_id.b[3] = 0;
        _mob_id.b[1] = canidt1() >> 3;
        _mob_id.b[0] = (canidt1() << 5) | (canidt2() >> 3);
        _extended = false;
    }
    page_release();

    return _mob_id.id;
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::rx_mode(bool extended, int
max_size, bool compare_remote_bit, bool compare_extended_bit)
{
    page(_mob_index);

    prepare_rx(extended, max_size, compare_remote_bit,
compare_extended_bit);

    cancdmob(cancdmob() | (ENABLE_RX << CANCDMOB_CONMOB));
    page_release();
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::buffer_mode(bool extended,
int max_size, bool compare_remote_bit, bool compare_extended_bit)
{
    page(_mob_index);

    prepare_rx(extended, max_size, compare_remote_bit,
compare_extended_bit);

    cancdmob(cancdmob() | (ENABLE_BUFFER << CANCDMOB_CONMOB));
    page_release();
```

```
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::disable()
{
    page(_mob_index);
    cancdmob(cancdmob() & (~CANCDMOB_CONMOBMASK));
    page_release();
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::tx_request(bool extended)
{
    page(_mob_index);

    prepare_id(extended);

    canidt4(canidt4() | (1<<CANIDT4_RTRTAG)); // set rtr -> data
request frame

    cancdmob(cancdmob() | (ENABLE_TX << CANCDMOB_CONMOB));
    page_release();
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::tx_data(unsigned char *data,
unsigned char size, bool extended)
{
    page(_mob_index);

    prepare_id(extended);

    size = (size & CANCDMOB_DLCMASK);
    cancdmob((cancdmob() & ~CANCDMOB_DLCMASK ) | size);

    // using autoincrement pointer feature of controller
    for(int i=0; i < size; i++)
        canmsg(data[i]);

    cancdmob(cancdmob() | (ENABLE_TX << CANCDMOB_CONMOB));
    page_release();
}
```

```
int AT90CAN128_CAN::AT90CAN128_CAN_MOB::retrieve_data(unsigned char
*data)
{
    int size;

    page(_mob_index);
    size = cancdmob() & CANCDMOB_DLCMASK;
    if(size != 0)
    {
        // using autoincrement pointer feature of controller
        for(unsigned char i=0; i < size; i++)
            data[i] = canmsg();
    }
    page_release();
    return size;
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::clear_all()
{
    for(int i = 0; i < _Traits::MAX_MOBS; i++)
    {
        page(i);
        canstmob(0);
        cancdmob(0);
        canidt4(0);
        canidt3(0);
        canidt2(0);
        canidt1(0);
        canidm4(0);
        canidm3(0);
        canidm2(0);
        canidm1(0);
        for(int i = 0; i < _Traits::MAX_DATASIZE; i++)
            canmsg(0);
        page_release();
    }
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::prepare_rx(bool extended, int
max_size, bool compare_rtr, bool compare_ide)
{
```

```
    unsigned char size;
    prepare_id(extended);

    if(extended)
    {
        canidm1((_mob_mask.b[3] << 3) | (_mob_mask.b[2] >> 5));
        canidm2((_mob_mask.b[2] << 3) | (_mob_mask.b[1] >> 5));
        canidm3((_mob_mask.b[1] << 3) | (_mob_mask.b[0] >> 5));
        canidm4(_mob_mask.b[0] << 3);
    }
    else
    {
        canidm1((_mob_mask.b[1] << 5) | (_mob_mask.b[0] >> 3));
        canidm2(_mob_mask.b[0] << 5);
        canidm3(0);
        canidm4(0);
    }
    if(compare_ide)
        canidm4(canidm4() | (1<<CANIDM4_IDEMASK));
    if(compare_rtr)
        canidm4(canidm4() | (1<<CANIDM4_RTRMASK));

    size = (max_size & CANCDMOB_DLCMASK);
    cancdmob(cancdmob() | size);
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::prepare_id(bool extended)
{
    canstmob(0);
    cancdmob(0);

    if(extended)
    {
        canidt1((_mob_id.b[3] << 3) | (_mob_id.b[2] >> 5));
        canidt2((_mob_id.b[2] << 3) | (_mob_id.b[2] >> 5));
        canidt3((_mob_id.b[1] << 3) | (_mob_id.b[0] >> 5));
        canidt4(_mob_id.b[0] << 3);
        cancdmob(cancdmob() | (1<<CANCDMOB_IDE)); // use extended 2.0B
ID (29bits)
        _extended = true;
    }
```

```
    else
    {
        canidt1((_mob_id.b[1] << 5) | (_mob_id.b[0] >> 3));
        canidt2(_mob_id.b[0] << 5);
        canidt3(0);
        canidt4(0);
        _extended = false;
    }
}

void AT90CAN128_CAN::AT90CAN128_CAN_MOB::page(int mob) // Caution!
static mutex locking function, call release_page to unlock
{
    Page_Mutex.lock();
    if(mob < _Traits::MAX_MOBS)
        canpage( mob << CANPAGE_MOB );
}

//-------------

int AT90CAN128_CAN::mob_select(int index)
{
    if(index > _Traits::MAX_MOBS || index < 0)
        return INVALID_INDEX;

    selected_mob = index;
    return OK;
}

bool AT90CAN128_CAN::mob_is_free()
{
    if(mob_is_free(selected_mob) )//&& mobs[selected_mob].status() ==
MOB_DISABLED) necessary?
        return true;
    else
        return false;
}

int AT90CAN128_CAN::mob_rx_mode(unsigned long ID, unsigned long mask,
int max_data_size, bool extended, int type)
{
```

```
    bool compare_rtr = false, compare_ide = false;
    if(!check_id(extended, ID))
        return INVALID_ID;
    if(!mob_is_free())
        return NOT_FREE;
    if(max_data_size > _Traits::MAX_DATASIZE)
        return INVALID_SIZE;

    if(type & MOB_COMPARE_RTR_BIT )
        compare_rtr = true;
    if(type & MOB_COMPARE_IDE_BIT )
        compare_ide = true;

    mobs[selected_mob].id(ID);
    mobs[selected_mob].mask(mask);
    mobs[selected_mob].rx_mode(extended, max_data_size, compare_rtr,
compare_ide);

    return OK;
}

int AT90CAN128_CAN::mob_buffer_mode(unsigned long ID, unsigned long
mask, int max_data_size, bool extended, int type)
{
    bool compare_rtr = false, compare_ide = false;
    if(!check_id(extended, ID))
        return INVALID_ID;
    if(!mob_is_free())
        return NOT_FREE;
    if(max_data_size > _Traits::MAX_DATASIZE)
        return INVALID_SIZE;

    if(type & MOB_COMPARE_RTR_BIT )
        compare_rtr = true;
    if(type & MOB_COMPARE_IDE_BIT )
        compare_ide = true;

    mobs[selected_mob].id(ID);
    mobs[selected_mob].mask(mask);
    mobs[selected_mob].buffer_mode(extended, max_data_size,
compare_rtr, compare_ide);
```

```
    return OK;
}

int AT90CAN128_CAN::mob_send_data(unsigned char *data, unsigned long
ID, unsigned char size, bool extended)
{
    if(!check_id(extended, ID))
        return INVALID_ID;
    if(size > _Traits::MAX_DATASIZE)
        return INVALID_SIZE;
    if(!mob_is_free())
        return NOT_FREE;


    mobs[selected_mob].id(ID);
    mobs[selected_mob].tx_data(data, size, extended);

    return OK;
}

int AT90CAN128_CAN::mob_retrieve_data(unsigned char *data, unsigned
long *ID, bool *extended)
{
    int size;
    if(!(mobs[selected_mob].status() & MOB_RX_COMPLETED))
        return NO_DATA;

    size = mobs[selected_mob].retrieve_data(data);
    *ID = mobs[selected_mob].id();
    *extended = mobs[selected_mob].is_extended();

    return size;
}

int AT90CAN128_CAN::mob_request_data(unsigned long ID, bool extended)
{
    if(!check_id(extended, ID))
        return INVALID_ID;
    if(!mob_is_free())
        return NOT_FREE;
```

```
    mobs[selected_mob].id(ID);
    mobs[selected_mob].tx_request(extended);

    return OK;
}

int AT90CAN128_CAN::high_prio_mob()
{
    int mob = (canhpmob() >> CANHPMOB_HPMOB);

    if(mob < _Traits::MAX_MOBS)
        return mob;
    else
        return NO_PRIORITY_MOB;
}

int AT90CAN128_CAN::status()
{
    unsigned char stt;
    int ret = 0;

    stt = cangsta();
    if(stt & CANGSTA_OVFG)
        ret |= CAN_OVERLOAD_TX;
    if(stt & CANGSTA_TXBSY)
        ret |= CAN_TXING;
    if(stt & CANGSTA_RXBSY)
        ret |= CAN_RXING;
    if(stt & CANGSTA_BOFF)
        ret |= CAN_BUSOFF;
    if(stt & CANGSTA_ENFG)
        ret |= CAN_ENABLED;
    if(stt & CANGSTA_ERRP)
        ret |= CAN_ERROR_PASV;

    return ret;
}

long AT90CAN128_CAN::baud_rate(long baudrate)
{
```

```
    unsigned char BRP, PRS, PHS1, PHS2, SJW, SMP, TimeQuanta;
    long BRPCalc;
    /*
    Tsyns = (BRP + 1)/CLKio  = 1Tq, CAN TimeQuantum
    Tsjw = Tq * (SJW + 1)
    Tprs = Tq * (PRS + 1),  Tprs >= (2*Delay), Delay =
max(OutPutDelay(node 1) + BusDelay + InputDelay(node 2))
                            Compensates physical delays
    Tphs1 = Tq * (PHS1 + 1)
    Tphs2 = Tq * (PHS2 + 1)
    Here we will use fixed 8 time quanta (8 to 25 can be used)
    So with 1Tq for Tsyns, 1 + PSR, 1 + PHS1 and 1 + PHS2, 1 + SJW we
are left with 4Tq
    At this time PHS2 will be set to 0 and PHS1 = 1 and PRS = 3
    As Tsjw can't be bigger than any of PHS, so SJW will be 0
    This will give one of the combinations that give exactly 87.5%
sample point
    (calculated by PSH1+PRS+Tsyns = 5Tq, 5Tq/8Tq = 87.5%, default of
CANOpen and DeviceNet)
    SMP will be set to 0 because if it is activated, Tprs is increased
in 1Tq
    Now we have to calculate the prescaler...
    */
    PRS = 2;   // 2
    PHS1 = 1;  // 1 for 75% sample point like atmel manual
    PHS2 = 1;  // 1
    SJW = 0;   // 0
    SMP = 0;
    // Below code can be manteined as is if the above parameter
calculation changes
    TimeQuanta = 4 + PRS + PHS1 + PHS2 + SJW;

    BRPCalc = baudrate;
    BRP = (Traits<AVR8>::CLOCK / (BRPCalc * TimeQuanta)) - 1;

    BRP = (BRP << CANBT1_BRP) & CANBT1_BRPMASK;
    canbt1(BRP);

    SJW = (SJW << CANBT2_SJW) & CANBT2_SJWMASK;
    PRS = (PRS << CANBT2_PRS) & CANBT2_PRSMASK;
    canbt2(SJW | PRS);
```

```
    PHS1 = (PHS1 << CANBT3_PHS1) & CANBT3_PHS1MASK;
    PHS2 = (PHS2 << CANBT3_PHS2) & CANBT3_PHS2MASK;
    SMP = (SMP << CANBT3_SMP) & CANBT3_SMPMASK;
    canbt3(PHS2 | PHS1 | SMP);

    _baud_rate = Traits<AVR8>::CLOCK / (int)(TimeQuanta * (BRP +
1));// return the real baudrate...

    return _baud_rate;
}

long AT90CAN128_CAN::baud_rate(int brp, int prop_seg, int phase_seg1,
int phase_seg2, int sjw)
{
    unsigned char BRP, PRS, PHS1, PHS2, SJW, SMP, TimeQuanta;
    long BRPCalc;

    //Considering the received time quanta we adjust to avr register
setting just subtracting one

    BRP = brp - 1;
    PRS = prop_seg - 1;
    PHS1 = phase_seg1 - 1;
    PHS2 = phase_seg2 - 1;
    SJW =  sjw - 1;
    SMP = 0;

    TimeQuanta = prop_seg + phase_seg1 + phase_seg2 + sjw;

    BRP = (BRP << CANBT1_BRP) & CANBT1_BRPMASK;
    canbt1(BRP);

    SJW = (SJW << CANBT2_SJW) & CANBT2_SJWMASK;
    PRS = (PRS << CANBT2_PRS) & CANBT2_PRSMASK;
    canbt2(SJW | PRS);

    PHS1 = (PHS1 << CANBT3_PHS1) & CANBT3_PHS1MASK;
    PHS2 = (PHS2 << CANBT3_PHS2) & CANBT3_PHS2MASK;
    SMP = (SMP << CANBT3_SMP) & CANBT3_SMPMASK;
    canbt3(PHS2 | PHS1 | SMP);
```

```
    _baud_rate = Traits<AVR8>::CLOCK / (int)(TimeQuanta * brp);//
return the real baudrate...

    return _baud_rate;
}

__END_SYS
```

Arquivo: ./src/mach/at90can128/can_init.cc

```
// EPOS-- AT90CAN128_CAN Initialization


#include <mach/at90can128/can.h>

__BEGIN_SYS

// Class initialization
int AT90CAN128_CAN::init(System_Info * si)
{
    db<AT90CAN128_CAN>(TRC) << "AT90CAN128_CAN::init()\n";
    for( int i = 0; i < _Traits::MAX_MOBS; i++ )
   mobs[i].mob_index(i);

    return 0;
}

__END_SYS
```

Anexo B – Artigo

Suporte a Redes CAN para Aplicações Embarcadas

Alessandro Barreiros Maurici

Laboratório de Integração Software e Hardware
Universidade Federal de Santa Catarina

alebm@inf.ufsc.br

Abstract. *Utilization of communication networks for embedded systems, gone from optional to a necessity, these networks type is called Fieldbus.*
CAN, Controller Area Network, is a specification of interconnection and protocol for communication. CAN networks are relatively old, but, with the miniaturization and falling prices of electronics circuits, this kind of networks became growing higher. However we still face software implementation problems, as an example, the low or none reuse of code is one of the problems.
A project of Application-Oriented Operating System, like EPOS, for these kind of system, minimizes time and costs for a developer do implement a software, which may or have to run in mixed platforms.
This work will show an introduction of CAN and EPOS, also some explanation about AVR microcontroller basic architecture, this microcontroller is utilized in the EPOS mediator implementation, which will provide CAN access into AVR running EPOS system.

Resumo. *A utilização de redes de comunicação para sistemas embutidos deixou de ser uma ferramenta opcional para tornar-se uma necessidade, estas redes são denominadas Fieldbus.*
CAN, Controller Area Network, é uma especificação de interconexão e protocolo para comunicação. Redes CAN são relativamente antigas, porém, com a miniaturização e queda dos custos dos circuitos eletrônicos, o uso deste tipo de rede vem crescendo muito. Entretanto ainda enfrentamos o problema da implementação de software para sistemas embutidos, por exemplo, o pequeno ou nenhum reaproveitamento de código é um dos problemas enfrentados.
Um projeto de Sistema Operacional Orientado à Aplicação, como o EPOS, para sistemas deste gênero, minimiza o tempo e o custo para um desenvolvedor implementar um software, o qual deverá ou poderá rodar em diferentes plataformas.
Neste trabalho será exibida uma visão sobre CAN e do sistema operacional EPOS, também será dada uma introdução sobre a arquitetura básica do microcontrolador AVR, utilizado na implementação de um mediador, que possibilitará o uso de redes CAN no AVR utilizando o sistema EPOS.

1. Introdução

A necessidade, da comunicação entre dispositivos eletrônicos, gera uma área de estudos cujo campo para pesquisa é vasto. Com os avanços tecnológicos obtidos nos sistemas digitais, conseguimos dispositivos cada vez menores, a necessidade de comunicar estes pequenos dispositivos com outros da mesma categoria ou até com sistemas maiores é indispensável. Exemplos de sistemas assim podem estar em qualquer lugar, no carro, no avião, em casa.

Este trabalho visa amparar o programador de software para plataformas embutidas, dando suporte ao nível de sistema operacional para utilização de dispositivos embutidos com funcionalidade de conexão a redes CAN, uma estrutura de rede a qual provê várias opções desejáveis ou até indispensáveis para redes embutidas.

A primeira parte deste trabalho irá introduzir o que é e como funciona CAN. Em seguida será feita uma breve apresentação ao sistema EPOS, utilizado na implementação gerada,

juntamente com uma visão da arquitetura da família de microcontroladores AVR e algumas considerações sobre a implementação. Na parte final serão feitas, conclusão e análise sobre pesquisas futuras.

2. Controller Area Network

Com o desenvolvimento iniciado em 1983, na empresa BOSCH, para ter uma solução de uma interna para automóveis, "Controller Area Network"(CAN), foi anunciada oficialmente em 1986 pela BOSCH na Alemanha. Inicialmente para uso em unidades de controle eletrônico nos carros produzidos pela Mercedes. Em 1987 surgiram os primeiros circuitos integrados para CAN, fabricados pela Intel e pela Philips.

2.1. Características Gerais

CAN é um barramento serial para interligar dispositivos em rede. Como citado anteriormente, foi criada inicialmente para uso em sistemas de automóveis, mas logo teve o uso estendido para aplicações industriais. Hoje este barramento é usado primariamente em sistemas embutidos. Como detalhado por Livani, Kaiser e Jia [6], CAN possui facilidades que são muito desejadas na área da computação embutida, como tolerância a EMI, prioridade de mensagens, recuperação de falhas, entre outras.

Uma rede CAN pode interligar até 2032 dispositivos, sendo que o limite prático é de aproximadamente 110 dispositivos [4], cada um destes é tratado como um nó da rede. No nível físico, o link serial mais usado é composto de dois fios, o sinal tem característica diferencial, é capaz de operar até 1 Mbps, tendo restrições de velocidade em virtude da distância entre os nós. Para uma rede, com extensão 1 km, a velocidade por ser reduzida até 50Kbps. Cada nó ligado a este link serial é capaz de ouvir, simultaneamente a outros nós, os dados transmitidos na rede. A escrita, porem, é uma operação permitida somente para um dispositivo por vez.

O protocolo CAN 2.0A tornou-se o padrão ISO 11898-1 em 1993. A última versão do protocolo é a 2.0B. A maior diferença entre as duas versões é a quantidade de bits no identificador, 11 bits para o 2.0A e 29 bits para o 2.0B.

2.2. Arquitetura

A rede é multi-master, ou seja, pode ter mais de um nó controlador, o que facilita a criação de um sistema redundante. Para isto, usa como protocolo de acesso o CSMA/CD+AMP (Carrier Sense Multiple Access/Collision Detection + Arbitration on Message Priority), desta forma, CAN trabalha de modo semelhante a ethernet comum, mas ao invés de corrigir colisões de transmissão fazendo com que os dois nós em conflito parem de transmitir, a rede CAN usa um arbítrio de comparação binária para definir a prioridade das mensagens e decide qual será enviada. Quanto menor o valor associado maior a prioridade. Os bits que trafegam na rede recebem uma denominação de dominante e recessivo, um bit dominante representa o valor lógico 0 e o recessivo, o valor lógico 1 [1].

As mensagens, transmitidas no barramento, não contem endereços de transmissor ou receptor, elas podem possuir um identificador único de acordo com o conteúdo, assim cada receptor pode testar este identificador, portanto, se ele identificar um conteúdo relevante, a mensagem é processada.

CAN especifica, de acordo com a referência ISO/OSI, a camada física e a camada de enlace. A camada de aplicação é definida em nível de usuário.

2.2.1. Tipos de Quadros

CAN utiliza variados tipos de quadros para envio de dados, requisição de dados, propagação de erros, e mensagens de notificação de sobrecarga.

Como citado anteriormente, o quadro de dados é diferente para os padrões 2.0A e 2.0B. Este quadro é composto basicamente do identificador (11/29 bits), de um campo de controle, do

dado a ser enviado, do CRC do quadro e um campo denominado ACK, utilizado para os receptores sinalizarem recebimento do quadro. Pode ser enviado até 8 bytes de dados em cada quadro.

O quadro de requisição remota é enviado quando um nó necessita de algum dado. Este quadro é idêntico ao quadro de dados exceto por não conter o campo de dados, e possui um dos bits de controle de valor contrário ao quadro de dados.

O quadro de erro possui um campo flag e um campo delimitador, este último consiste de 8 bits recessivos, já o campo de flag é composto por 6 bits. Os bits podem ser todos recessivos para um flag de erro ativo ou todos recessivos para um flag de erro passivo. A utilização do flag ativo ou passivo depende do estado do nó, conforme será visto no item 2.5.

Um quadro de sobrecarga pode ser enviado por um nó para sinalizar que o receptor atual necessita de um tempo até o recebimento do próximo quadro, geralmente por falta de processamento ou memória. Este quadro é composto por 6 bits dominantes para o campo de flag, e mais 8 bits recessivos no campo delimitador do quadro.

2.3. Filtragem e Validação de Mensagens

Para um nó receber uma mensagem transmitida no barramento ele deve estar preparado para receber mensagens do mesmo tipo daquela que foi transmitida. O tipo de mensagem é obtido através do identificador.

A comparação entre o identificador esperado e o identificador recebido pode levar em consideração máscaras. Para o receptor, quando uma mensagem válida é encontrada, ou seja, não ocorreu erro até o bit Final de quadro ser recebido, e a comparação for positiva, o nó irá guardar o dado e sinalizar o sistema.

A validação de quadros pode também acontecer para o transmissor, tendo o mesmo critério de validação do receptor. Caso a mensagem não seja válida, ela será retransmitida automaticamente.

2.4. Configuração de Sincronização e Velocidade

Esta é uma parte crítica da rede, uma má configuração pode resultar em uma degradação da performance da rede ou até uma falha da mesma [10].

Para atingir a velocidade de transmissão requerida, existe uma grande combinação de configurações, que devem ter seus valores respeitados e configurados de acordo com o meio de transmissão.

CAN possui uma unidade temporal chamada “time quantum” ou tq, ela é definida pelo selecionador de taxa de transmissão do subsistema CAN, generalizando tempos: $tq = BRP / fsys$. Um bit transmitido no CAN possui a seguinte divisão [18]: seguimento de sincronismo, propagação, fase 1 e fase 2. Cada uma destas divisões pode ser configurada com 1tq até 8tq, exceto pelo sincronismo que sempre é 1tq.

2.5. Tratamento de Erros

CAN possui um sistema muito confiável de tratamento de erros, todos os erros globais ao sistema são detectáveis, todos erros locais ao transmissor são detectáveis, uma mensagem pode conter até cinco erros distribuídos aleatoriamente, rajadas de erros com comprimento máximo de quinze bits ou de tamanho ímpar são detectados.

Temos cinco tipos de erros detectáveis [1]: erro de bit, erro de codificação, erro de CRC, erro de formação e erro no campo ACK.

A sinalização de erro é aplicada igualmente a todos os tipos exceto para o erro de CRC. Enquanto a sinalização padrão é transmitida no próximo bit após a detecção do erro, o erro de CRC é transmitido somente depois do recebimento do delimitador no campo ACK.

Para controle de erro no barramento, o confinamento de falhas, cada nó possui um contador de erros de transmissão e recepção. Com base nestes dois contadores, um nó pode estar em um destes modos:

- Ativo: nó em funcionamento normal.
- Passivo: nó funcional, porém, com certas restrições na transmissão de pacotes. Neste estado o nó pode voltar a ser ativo caso os contadores de erros sejam inferiores a 128.
- Inativo: este nó não pode exercer modificações no estado do barramento. Este pode tornar-se ativo, e com os contadores de erros zerados, após ocorrerem 128 pacotes contendo 11 bits recessivos.

2.6. Aplicações

Como visto anteriormente, pode ser muito interessante utilizar uma rede CAN quando lidamos com ambientes hostis, ou que necessitem de transmissão de mensagens com prioridade e capacidade de serem enviadas em tempo real. Contudo, ainda podemos aplicar CAN para redes comuns onde vários microcontroladores e sensores necessitem de interconexão. Assim verificamos que CAN está presente em todo tipo de setor, em especial os que seguem abaixo.

- Aplicações Agrícolas: existe um padrão, ISO 11783, para interconexão de maquinário agrícola.
- Aplicações Aeroespaciais: como podemos notar, CAN é uma excelente escolha para este tipo de aplicação considerando o alto índice de interferência sofrida pelos dispositivos deste setor. No setor aéreo, existe o padrão CANaerospace [7], utilizado e padronizado posteriormente pela NASA. Este também é utilizado por empresas como a Airbus e a Bombardier. Na área espacial, podemos citar a ESA como uma das usuárias desta arquitetura, como visto na sonda SMART-1 [8], desenvolvida para demonstrar tecnologias chaves no futuro da exploração espacial. CAN também foi utilizado em outros inúmeros satélites experimentais, principalmente universitários e de radioamadores.
- Aplicações Automobilísticas: este setor foi responsável pela criação do CAN, portanto, inúmeras aplicações utilizam as vantagens de redes CAN para o funcionamento. A criação de redes de sensores internos do veículo ajuda a melhorar o funcionamento de determinados atuadores como, por exemplo, de freios ABS, tendo informações sobre o funcionamento global do sistema de sensores/atuadores, um determinado atuador pode ser utilizado usando uma estratégia mais adequada de acionamento. Fabricantes como a BMW, Ford, General Motors, Toyota, DaimlerChrysler são alguns dos utilizadores desta tecnologia de interconexão.
- Aplicações Comerciais: geralmente não requerem as funcionalidades providas pelo CAN, porém devido a grande disponibilidade de componentes, baixo custo e boa aceitação em outras áreas, CAN também é uma alternativa atrativa entre as tecnologias de rede que competem no setor comercial. Grande parte das aplicações comerciais, baseadas em CAN, utiliza a camada de aplicação CANopen.
- Aplicações Industriais: novamente uma área onde são necessárias a robustez e resistência a interferências. A camada DeviceNet foi criada e mantida como uma camada de aplicação padrão para produtos industriais.
- Aplicações Médicas: As características da tecnologia CAN são adequadas para aplicações médicas, já que elas têm confiabilidade e segurança de transmissão. Aplicações médicas utilizam a camada de aplicação CANopen como padrão. Esta camada foi especificada em conjunto pela GE Medical Systems, Philips Medical e Siemens Medical sob o nome da organização CiA [9]. Esta união de grandes empresas para a definição de uma camada padronizada demonstra a alta importância desta tecnologia para o setor.

2.7. Comparação com outras Tecnologias

CAN é uma das tecnologias categorizadas como *fieldbus*, portanto, é interessante comparar esta tecnologia com outras da mesma categoria. Infelizmente, não temos uma metodologia de

comparação capaz de demonstrar qual rede é melhor ou pior devido aos setores onde tais tecnologias são aplicadas.

2.7.1. Tabelas de Comparação

As tabelas [13] a seguir irão demonstrar características físicas e operacionais das seguintes redes: CAN, AS-I [11], Foundation Fieldbus H1/HSE e PROFIBUS DP/PA [12, 14].

	Topologia	**Meio utilizado**	**Extensão máxima**
CAN	Barra	Fibra ótica e par trançado	1km
AS-I	Árvore, barra	2 fios	100m 300m com repetidor
Foundation Fieldbus H1	Barra, estrela	Fibra óptica e par trançado	1900m
Foundation Fieldbus HSE	Estrela		100m(par trançado) 2km(fibra óptica)
PROFIBUS DP	Anel, barra, estrela	Fibra óptica e par trançado	100m por segmento, até 24km utilizando fibra óptica
PROFIBUS PA			

	Acesso ao meio físico	**Quantidade de conexões suportadas**
CAN	CSMA/CD+AMP	2032 [**]
AS-I	Polling cíclico mestre/escravo	62 + 1 mestre
Foundation Fieldbus H1	Token Ring	240 por segmento
Foundation Fieldbus HSE	CSMA/CD	2^{32} (endereçamento IP)
PROFIBUS DP	Token Ring	32 por segmento (máximo: 126)
PROFIBUS PA		

	Tamanho máximo de transferência	**Velocidade**
CAN	8 bytes	Até 1 Mbit/s
AS-I	8 bits	167 Kbit/s
Foundation Fieldbus H1	128 bytes	31,25 Kbit/s
Foundation Fieldbus HSE	Variável (TCP/IP)	Até 100 Mbit/s
PROFIBUS DP	Variável, 0 até 244 bytes	Até 12 Mbit/s
PROFIBUS PA		31,25 kbit/s

2.7.2. Discussão Tecnológica

Não temos uma tecnologia disponível onde todos os casos de uso sejam enquadrados com perfeição, a escolha de uma tecnologia tornou-se muito dependente da aplicação, parâmetros como tamanho da rede, taxa de atualização de dados, número de nodos conectados e segurança da transferência da informação são significativos para a escolha do sistema [16].

Com este problema de escolha, acabamos, muitas vezes, optando pela heterogenização do sistema com a utilização duas ou mais tecnologias de conexão. Como citado em [15], esta coexistência de tecnologias tende a crescer cada vez mais devido aos requisitos atuais onde devemos contrabalançar segurança, velocidade e custos. Um bom exemplo disto são os carros modernos, hoje além de toda a parte de controle dos sistemas do carro, geralmente onde CAN é aplicada, temos uma integração com uma rede de entretenimento dos usuários do veículo, a qual deve suportar taxas de transferências muitas vezes superior devido ao tráfego multimídia. Temos também novos dispositivos de auxilio a navegação que devem trocar dados a uma grande

[**] CAN não utiliza endereçamento de nodos para enviar as mensagens. A camada de aplicação irá determinar efetivamente as quantidades de nodos possíveis a serem conectados

velocidade, exigindo uma rede complementar com uma maior taxa de transferência, porém segura e que, em caso de mau funcionamento, não afete as redes principais de controle.

3. CAN para o EPOS

3.1. EPOS

O EPOS é um sistema operacional implementado por Fröhlich[5], utilizando técnicas AOSD*. Com ele, Fröhlich também demonstrou a viabilidade do uso deste tipo de sistema operacional em ambientes heterogêneos, já que um sistema operacional orientado à aplicação, ou seja, configurado, utilizando somente os componentes necessários para funcionamento da mesma[5], pode ser utilizado em um ambiente de alta performance, com um menor impacto sobre a memória e outros recursos do sistema.

Esta adaptabilidade apresentada pelo EPOS, faz dele um sistema de bom custo benefício tanto para ambientes embutido, onde podem estar sendo utilizado microcontroladores de 8 bit (comuns para redes de sensores por exemplo), até sistemas de alta performance, como foi o caso do Cluster SNOW[3] utilizado para validação deste sistema operacional.

O fato de este sistema operacional poder ser utilizado com sistemas 8 bit traz uma nova perspectiva para desenvolvedores do setor, os quais, dominam mais de sessenta por cento do mercado de processadores e que possuem poucas ferramentas para o desenvolvimento, principalmente quando analisamos portabilidade das aplicaçações.

3.1.1. A escolha do EPOS

A escolha deste sistema para base da implementação desenvolvida com este trabalho não foi ao acaso. Os serviços providos pelo sistema operacional são essenciais para um desenvolvimento rápido e com menor possibilidade de erros. Assim, escolhendo o EPOS, além de prover a facilidade do uso para CAN, o desenvolvedor já poderá utilizar uma gama de funções já disponibilizada pelo sistema operacional como abstrações de threads, sensores, etc. Estas funcionalidades também aceleraram o desenvolvimento do próprio mediador CAN implementado.

Analisando a portabilidade do código, utilizando um sistema operacional multi-plataforma como o EPOS, o esforço para utilizar o mesmo aplicativo em uma plataforma distinta será pequeno, talvez inexistente, o que não aconteceria caso o aplicativo fosse desenvolvido utilizando bibliotecas ou baseando-se em uma linguagem de programação como base de sustentação para portabilidade [18].

3.1.2. Arquitetura do EPOS

O EPOS é dividido em três partes básicas, as abstrações, os aspectos e os mediadores, porém, a versão utilizada neste trabalho está propositalmente desprovida dos aspectos.

As abstrações englobam todas as implementações independentes de plataforma, por exemplo, threads, networking, etc.

Para implementação de partes específicas das plataformas, o EPOS utiliza um mediador. Caso exista uma funcionalidade comum para mais de uma plataforma suportada pelo EPOS, cada plataforma possuirá um mediador específico, porém existirá uma interface definida, um “contrato de interface” entre o sistema e a máquina [18], alcançando assim a portabilidade das abstrações.

3.2. Hardware utilizado

Para implementação do mediador, foi escolhido o microcontrolador AVR, modelo AT90CAN128. Da mesma forma pela qual o EPOS foi escolhido como sistema operacional, a escolha do AVR foi favorecida por diversos fatores. O primeiro deles foi o suporte do EPOS

* Application Oriented System Design

para a arquitetura. Com o suporte provido pelo sistema operacional, foi criada uma arquitetura derivada, onde o código implementado, na maioria das vezes, retrata apenas as diferenças e funcionalidades adicionais, como algumas pequenas modificações no hardware base e adições, como o suporte a CAN.

Outro fato importante é o AVR possuir uma aceitação cada vez maior no mercado, deste modo esta linha de microcontroladores tende a crescer em qualidade e em opções, tornando-se mais importante. Como indicação final, o AT90CAN128 é um dos poucos microcontroladores 8 bit existente no mercado possuindo um controlador CAN embutido, com a especificação 2.0B totalmente implementada.

3.2.1. Arquitetura básica

Um AVR é um microcontrolador RISC, baseado na arquitetura de Harvard, possui 32 registradores de 8 bits para de uso geral, o grupo de instruções possui em média 130 instruções, as quais, em grande parte, serão executadas em um único ciclo. O microcontrolador possui uma série de portas de entrada e saída de uso geral. A facilidade de programação da memória flash embutida no controlador é um dos grandes atrativos juntamente com o baixo consumo de energia e alta performance (quando comparado a outros controladores da mesma área de atuação).

3.2.1.1. Características Principais

Registradores

Existem quatro tipos de registradores no AVR: registrador de estado, registradores de propósito geral, registrador de página e registradores de entrada/saída.

O registrador de estado (SREG) guarda as informações sobre a ultima instrução aritmética executada e também possui o bit para habilitar/desabilitar globalmente as interrupções.

Os 32 registradores de propósito geral podem ser utilizados para executar as operações requeridas com alto desempenho.

Presente somente nos modelos que podem utilizar memória externa, o registrador de página da RAM (RAMPZ), é utilizado para dizer qual página do banco de memória externo deve ser utilizada.

Registradores de entrada e saída são utilizados para o controle dos “periféricos”, eles são a interface de comunicação com recursos como o CAN.

Os registradores descritos podem estar localizados em um espaço reservado de entrada/saída, sendo estes acessíveis com instrução IN/OUT. Os 32 registradores de propósito geral e alguns registradores de entrada e saída são somente mapeados em memória, onde, instruções comuns de acesso à memória são utilizadas para escrever ou ler os valores deles. Os registradores localizados no espaço reservado de entrada e saída, também estão mapeados em memória, facilitando a programação.

Memória

Como citado anteriormente, a memória no AVR é dividida em memória de programa e memória de dados. A primeira é dividida em 64K x 16, devido às instruções do AVR serem em sua grande maioria de 16 ou 32 bits. As versões, derivadas da família ATmega, possuem uma opção para reservar uma parte da memória de programa para um sistema de boot. Nestes modelos ainda é possível modificar a memória de programa utilizando o próprio controlador, por exemplo, podemos construir um setor de boot que fica esperando pela programação via UART, tão logo ele receba os dados, estes dados serão escritos na memória de programa e, quando a seqüência for finalizada o aplicativo é executado.

Periféricos

A família de microcontroladores AVR é constituída, existindo variações, pelos seguintes periféricos:

- ADC (Conversor Analógico Digital): A maior parte dos controladores AVR possuem conversores analógico digital de 8 ou 10 bits.
- Pinos de uso geral (GPIO): O AVR é provido de uma ou mais portas de entrada e saída para uso geral, cada uma pode acomodar até oito pinos de uso geral.
- Timers: como padrão da família AVR, eles são periféricos que permitem o software realizar uma contagem temporal, baseado em um clock de referência fornecido internamente ou externamente. Estes timers podem operar em diversos modos, desde contadores até geradores de PWM.
- SPI (Interface Serial para Periféricos): é um sistema de transferência de dados síncrona entre um dispositivo mestre e os periféricos ligados a ele. É do tipo *full duplex* e utiliza três linhas de sinais para trabalhar desde modo, Rx, Tx e Clock. Uma facilidade importante do AVR é a facilidade de programação do mesmo através do SPI ligando-o através de um circuito simples a uma porta paralela de um PC.
- UART, USART (Receptor/Transmissor Serial Síncrono/Assíncrono): interface amplamente utilizada para comunicação serial, com protocolo compatível com a RS232 dos PCs(salvo o nível elétrico).
- TWI (Interface Serial de Duas linhas): assim como o SPI, é um tipo de barramento para controle de periféricos, porém utiliza somente dois fios.
- CAN: periférico exclusivo do modelo AT90CAN128. O controlador embutido neste modelo, é totalmente compatível com os padrões CAN 2.0A e 2.0B. Pode armazenar até 15 mensagens, chegar a taxa de transmissão de 1Mbit/s utilizando um clock de 8Mhz(metade do clock máximo alcançado por este modelo), possui um timer dedicado caso seja necessário implementar um protocolo TTCAN(*time triggered*). O controlador CAN pode operar em um modo somente escuta, com isto, pode ser implementado um algoritmo para achar automaticamente o baud rate da rede onde foi inserido o dispositivo.
- Outros: o AVR pode contar com um comparador analógico, utilizando dois pinos de entrada analógica para verificar a diferença de tensão entre os pinos. Muitos possiuem watchdog embutido. Nas unidades mais recentes, podemos utilizar o BOD (Brown-out Detector) para evitar o uso do controlador, quando a alimentação do mesmo estiver abaixo do nível estipulado pela programação interna. Existem ainda modelos com controlador de LCD ou geradores especiais de PWM para controle de motores.

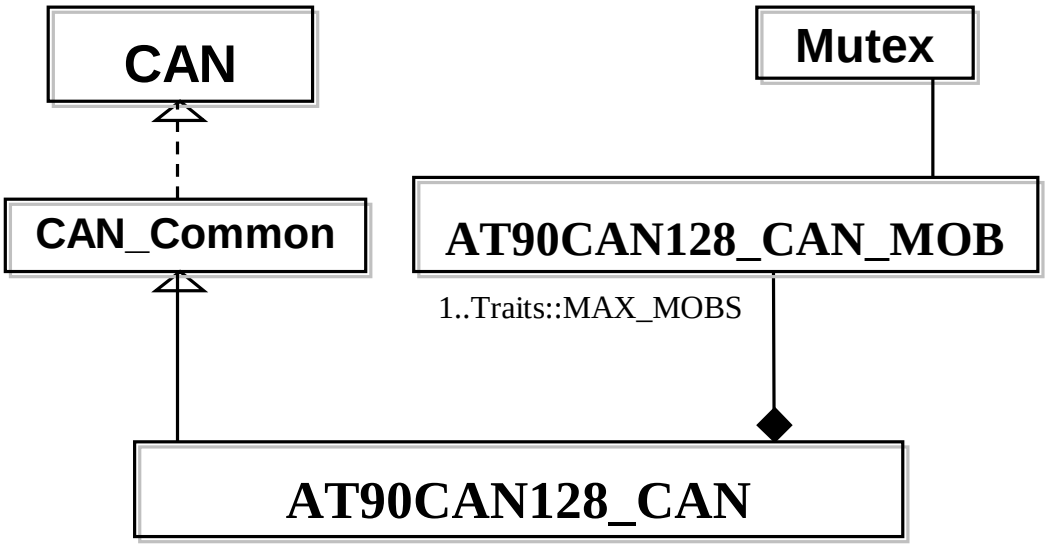
3.2.1.3. AT90CAN128

Este modelo específico apresenta as seguintes características[2]:

- 128 KB de memória Flash, 4 KB de EEPROM
- 4 KB de SRAM, interna, podendo ser adicionado até 64 KB de memória externa
- Interface JTAG para programação e depuração
- Controlador CAN (2.0A e 2.0B)
- 4 Timers, dois deles com 8 bits de precisão e o restante com 16 bit de precisão, todos com divisores de clock com 10 bits de precisão
- Interface SPI, TWI e duas USART
- Velocidade de até 8MHz utilizando

3.3. Implementação do Mediador

Com o estudo e análise do sistema operacional base e do hardware onde foi planejada a primeira implementação do mediador CAN, foi obtido o seguinte diagrama:



3.3.1. Considerações Associadas ao Controlador CAN do AVR

Como CAN possui uma especificação aberta e definida, grande parte das implementações, incluindo a do AVR, segue uma linha definida de informações sobre erros, transmissão de dados, entre outros. Contudo, a maneira como serão armazenadas as mensagens recebidas ou a serem transmitidas é de livre escolha do fabricante.

No AVR foi adotado o sistema de *mailbox* (caixa de correio). Este sistema de armazenamento de mensagens utiliza o conceito onde existe um objeto de mensagem com alguns registradores, associados a erros, estados e configurações daquele objeto. Este sistema foi utilizado para diminuir a sobrecarga do microcontrolador agregado ao tempo de recuperação da mensagem, assim mesmo, sendo um controlador com recursos limitados, ele consegue trabalhar na velocidade máxima da especificação CAN. Porém, este sistema pode levar a perda de mensagens transmitidas no barramento, caso o software não consiga preparar um objeto de mensagem para recepção em tempo hábil. Uma fragilidade desta implementação no AVR acontece no momento de uso dos objetos de mensagem, já as mensagens são multiplexadas. Temos ainda um sistema de prioridade onde, caso dois objetos estiverem marcados para transmissão, o de maior prioridade será enviado antes. A prioridade diminui com o aumento do índice do objeto.

A estrutura básica de controle do sistema é baseada em um registrador de seleção de página e outro indicando o índice de maior prioridade. Registradores de estado, identificador, máscara e do buffer de dados são dependentes de cada objeto.

3.3.2. Interface com o Sistema

A interface visível para o software ou abstrações do sistema operacional é mantida na classe AT90CAN128_CAN, esta interface, deve ser conservada nas implementações de outras plataformas, conforme descrito em [18]. Em vista disto, foi analisada a implementação FlexCan [17] utilizada nos microcontroladores ColdFire fabricados pela Freescale. O FlexCan utiliza um sistema de armazenamento de mensagem semelhante ao AVR, porém, ao contrário do AVR, cada mensagem possui um endereço de memória separado. Assim, podemos aplicar o mesmo conceito de implementação seguido para o AVR, com uma classe modelando os buffers de mensagens do sistema. Mesmo para arquiteturas onde é utilizada uma implementação como a MSCAN, também utilizada em microcontrolares e DSPs da Freescale, baseada em um buffer

FIFO, pode ser utilizada a mesma interface e, ainda, mantendo uma classe modelando o objeto de mensagem que fica disponível na fila.

3.3.3. Otimizações

A implementação apresentada pode ter o uso de memória otimizado, caso necessário, descartando-se os buffers dos identificadores e máscaras utilizados para as mensagens. Aplicando isto no código, podemos modelar ao invés das mensagens, somente o ponto de acesso a elas, liberando cerca de 500 bytes na memória do sistema. Esta otimização é vital em sistemas onde o consumo de memória é crítico, considerando somente o uso da pequena memória disponível internamente no microcontrolador.

3.4. Mapeando CAN para a Abstração de Rede no EPOS

Para completar o conceito AOSD, o mediador CAN deve ter a funcionalidade disponibilizada através de uma abstração de rede, a qual suporta diversos modelos de redes, de forma transparente para a aplicação. Desta forma, tornar-se fácil a utilização das diversas redes suportadas pelo sistema operacional no software usuário. Ele não precisará saber em que tipo de rede está operando para funcionar corretamente.

3.4.1. Mapeamento de Endereços

Como já descrito anteriormente, uma rede CAN não utiliza explicitamente endereços fonte/destino, como ocorre na ethernet, para o envio de mensagens, é implementado um identificador que será enviado na mensagem. Esta mensagem é transmitida para todos os nós conectados na rede, além disso, podem existir diversos receptores para a mensagem transmitida.

Entretanto, para implementar uma abstração comum entre os diversos tipos de redes, devemos possuir um sistema comum de endereçamento dos nós entre elas.

Analisando as redes CAN, Profibus, Ethernet e derivadas, podemos classificar o sistema de endereçamento em dois modelos. O primeiro é o modelo utilizado pela rede CAN, onde mensagens identificadas são enviadas via broadcast. No segundo temos um sistema onde as mensagens contem o endereço fonte e destino, e a forma de envio pode ser feita através de broadcast, multicast ou ponto a ponto.

Considerando estes métodos de endereçamento, podemos implementar o sistema utilizado no CAN para as outras redes, ou criar um sistema de endereçamento para o CAN utilizando o campo de identificação, o qual se aproxima mais das redes restantes para que um modelo comum de endereçamento fonte/destino possa ser implementado.

Para esta escolha podemos considerar que o sistema da rede CAN é quantitativamente menos utilizado, sendo mais conveniente criar um mapeamento de um endereçamento fonte/destino para a CAN ao invés de criar uma nova implementação para cada uma das outras redes. Um problema decisivo poderia ser a performance, caso uma rede suporte somente comunicação ponto a ponto, a sobrecarga de realizar um broadcast pode ser muito grande. Outro fator que favorece esta decisão está no sistema de arbítrio na rede CAN. Se dois nós transmitirem uma mensagem com identificador igual, ao mesmo tempo (como o identificador é o mesmo nenhum dos nós irá perder o controle sobre o barramento durante a transmissão), porém com dados diferentes, será gerado um conflito de transmissão da mensagem entre os dois nós transmissores, este conflito, provavelmente, não pode ser resolvido e irá gerar um consumo indesejado de recursos da rede e dos nós durante algum tempo. Para resolver este problema pode ser utilizado um tag único para cada nó [19] no identificador da mensagem, o mesmo serve como um endereço.

Este mapeamento é satisfatório para cenários onde os nós em uso apresentam processamento e memória adequados, porém, caso os nós sejam compostos na maior parte de sensores “burros”, de baixo custo, o mapeamento fonte/destino pode ser proibitivo. Sistemas onde é relevante somente a fonte ou tipo de mensagem (conforme será visto na seção 4), o endereço destino é uma sobrecarga desnecessária. Neste cenário, é interessante utilizar somente

um endereço para identificação do sensor (resolvendo o problema da arbitragem), e espalhar a mensagem para todos os nós da rede como é feito originalmente nas redes CAN.

Ainda temos dois mapeamentos, cada um adequado para determinadas situações e inadequado para outras. Analisando mais profundamente, optar por implementar um mapeamento configurável é interessante. Caso o sistema esteja configurado para não utilizar endereço destino, o processamento do pacote deve depender somente do endereço fonte/prioridade. A implementação do envio da mensagem, sem o destinatário, em redes onde é necessário especificar um destino, será via broadcast/multicast, estes modos geralmente já são implementados em redes desde tipo.

3.4.2. Mapeamento das Propriedades de transmissão

As propriedades de transmissão relativas a cada tipo de rede também devem ser consideradas para
zimplementar a abstração, especialmente na rede CAN, já que a prioridade de transmissão no barramento é vinculada ao identificador da mensagem.

A prioridade da mensagem será considerada o único atributo do mapeamento, já que está presente nos modelos analisados.

Em redes CAN, a prioridade de uma mensagem decresce com o aumento do identificador, possibilitando uma enorme quantidade de prioridades. Redes Profibus possuem mensagens de baixa prioridade e alta prioridade. Nas redes Ethernet onde o padrão 802.1D [20] é implementado, é possível o uso de até oito níveis de prioridades (0 até 7, sendo 7, a maior prioridade).

Para um exemplo escolhemos ao acaso um mapeamento onde as mensagens poderão ser classificadas em quatro níveis de prioridade. Nível baixo, normal, médio e alto. Em uma rede Profibus, estes atributos seriam mapeados em prioridade baixa, baixa, alta, alta respectivamente. Já para a rede Ethernet, um mapeamento possível é prioridade 0, 2, 5 e 7. Na rede CAN, 2 bits do identificador mapeiam diretamente a prioridade requisitada.

Na prática, o número de prioridades deve ser grande o suficiente para permitir um escalonamento eficiente caso um sistema tempo real seja implementado. Assim o mapeamento das prioridades deverá ser feito dinamicamente conforme a quantidade de níveis oferecidos pela rede utilizada.

3.4.3. Mapeamento para uma Abstração com Suporte Ethernet e CAN

Para a Ethernet, o mapeamento de endereço é direto, para ambas as configurações citadas na seção 3.4.1. Na rede CAN foi escolhido utilizar os 7 bits menos significativos para o endereço fonte, caso configurado, os 7 bits seguintes para o endereço destino, devido ao limite físico do barramento e funcionamento do sistema de prioridade.

Para a prioridade, temos 3 bits na Ethernet e 22 bits/15 bits no CAN (considerando a versão 2.0B). Na abstração, temos um sistema de prioridades onde a base é a maior quantidade de níveis entre os protocolos suportados, neste caso 22 bits, que são quantizados devidamente para cada tipo rede suportada.

5. Conclusão

Com a pesquisa realizada sobre redes CAN, foi possível avaliar com mais clareza os recursos disponibilizados pela rede para melhor utilizá-los. Durante o período de pesquisa, também, foi constatada a existência de uma grande quantidade de informações disponíveis tanto no meio acadêmico quanto no meio industrial sobre redes CAN. O mesmo não foi verificado para outras redes, tornando visível o interesse na tecnologia por um grande grupo de pessoas, reforçando a citação feita na seção de motivações no início do trabalho.

Os objetivos propostos pelo trabalho foram alcançados com sucesso, embora com o desenvolvimento da abstração de rede, ainda em caráter inicial, possibilitando assim a

realização de futuras aplicações e pesquisas utilizando CAN, com todo o suporte oferecido pelo sistema EPOS.

5.1. Trabalhos Futuros

Uma extensão para utilizar um sistema de comunicação disparado por tempo pode ser adicionada ao sistema. O microcontrolador utilizado para a implementação possui recursos de hardware para implementar este tipo de comunicação. Outra extensão possível é criar um algoritmo para auto-ajuste do nó com a taxa de transmissão utilizada na rede, tornando o sistema ainda mais flexível.

Para a implementação de protocolos mais complexos sobre redes CAN será indispensável possuir um analisador de tráfego na rede. Os microcontroladores disponíveis no LISHA podem ser facilmente programados para esta tarefa, sendo configurados como um nó estritamente receptor na rede CAN e repassando o trafego recebido para um computador onde os dados serão efetivamente analisados.

Futuramente, com a possível necessidade de poder de processamento para realização de tarefas mais complexas, poderá ser necessária a implementação do mediador em outras plataformas.

A pesquisa em redes de sensores, utilizando objetos sentientes, realizada no LISHA, pode explorar a comunicação baseada em eventos facilmente implementada utilizando redes CAN, já que por definição, objetos sentientes são fracamente acoplados e comunicam-se através de eventos [21]. Finalmente, temos pesquisas na área de comunicação em tempo real, indispensável para redes mais críticas, como as utilizadas em carros.

Referências

1. BOSCH, Robert. **CAN Specification Version 2.0**. Stuttgart, 1991.
2. Atmel Corporation. **AT90CAN128 Datasheet Rev E**. Dezembro 2004.
3. CiA. **Controller Area Network (CAN): an overview**. Disponível em: <http://www.can-cia.de/can/>. Acesso em: 21 novembro 2004.
4. NILSSON, Staffan. **Controller Area Network - CAN Information**. Disponível em: <http://www.algonet.se/~staffann/developer/CAN.htm>. Acesso em: 21 novembro 2004.
5. DE MEDEIROS FRÖHLICH, A. A. **Application-Oriented Operating Systems**. Número 17. GMD Research Series. GMD - Forschungszentrum Informationstechnik. Sankt Augustin, Agosto 2001.
6. LIVANI ALI, Mohammad; KAISER, Jörg; JIA, Weijia. **Scheduling hard and soft real-time communication in a controller area network.** Control Engineering Practice 7. Julho 1999.
7. Stock Flight Systems. **CANaerospace Interface Specification v1.6**. Disponível em: <http://www.stockflightsystems.com/canas_16.pdf>. Acesso em: 10 junho 2005.
8. A. Elfving, L.Stagnaro, A.Winton. **SMART-1: Key technologies and autonomy implementations**. Proceedings of 4th IAA Intl.Conf. on low-cost planetary missions-John Hopkins University. Maio 2000.
9. ZELTWANGER, Holger. **Controller Area Network and CANopen in Medical Equipament**. Business Briefing: Medical Devide Manifacturing & Technology. 2002.
10. HARTWICH, Florian; BASSEMIR, Armin. **The Configuration of the CAN Bit Timing**. 6th International CAN Conference. Turin, Novembro 1999.
11. AS-Interface. **The System | Facts**. Disponível em: <http://www.as-interface.net/System/Facts>. Acesso em: 29 de agosto de 2005.
12. PROFIBUS. **PROFIBUS Profile for Process Automation**. Disponível em: <http://us.profibus.com/PA/PROFIBUSPA.pdf>. Acesso em 2 de setembro de 2005.

13. Synergic. **The Grid Connect Fieldbus Comparision Chart**. Disponível em: <http://www.synergetic.com/compare.htm>. Acesso em: 29 de agosto de 2005.
14. PROFIBUS. **PROFIBUS Technology and Application - System Description**. PROFIBUS Trade Organization. Outubro, 2002.
15. PARNELL, Karen. **Put the Right Bus in Your Car**. Xcell Journal. 2004.
16. FÄRBER, Georg. **Feldbussysteme - Oberseminar Prozeßrechentechnik**. Technische Universität München. 1994/95.
17. Freescale. **MCF5282 ColdFire Microcontroller User's Manual**. Novembro 2004.
18. POLPETA, Fauze Valério; DE MEDEIROS FRÖHLICH, A. A. **Hardware Mediators: A Portability Artifact for Component-based Systems.** In: Proceedings of the International Conference on Embedded and Ubiquitous Computing, volume 3207 of Lecture Notes in Computer Science. Aizu, Japan. Agosto 2004.
19. KAISER, Jörg; Mock, M. . **Implementing the Real-Time Publisher/Subscriber Model on the Controller Area Network (CAN)**. In Proceedings of 2nd International Symposium on Object-Oriented Real-Time Distributed Computing. Saint Malo, France. 1999.
20. Institute of Electrical and Electronics Engineers. **IEEE Standards for Local and Metropolitan Area Networks: Virtual Bridged Local Area Networks - IEEE Std 802.1Q**. Dezembro, 1998
21. FITZPATRICK, Adrian; BIEGEL, Gregory; CLARKE, Siobhán; CAHILL, Vinny. **Towards a Sentient Object Model**. Workshop on Engineering Context-Aware Object Oriented Systems and Environments. Seattle, USA. 2002.