Danillo Moura Santos

Projeto de sistemas embarcados: Um estudo de caso baseado em microcontrolador e seguindo AOSD

Florianópolis – SC

2006

Danillo Moura Santos

Projeto de sistemas embarcados: Um estudo de caso baseado em microcontrolador e seguindo AOSD

Monografia apresentada ao programa de Bacharelado em Ciências da Computação da Universidade Federal de Santa Catarina como requisito parcial para obtenção do grau Bacharel em Ciências da Computação

Orientador:
Professor Doutor Antônio Augusto Medeiros Fröhlich

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO

Florianópolis – SC

2006

Monografia de graduação sob o título *Projeto de sistemas embarcados: Um estudo de caso baseado em microcontrolador e seguindo AOSD*, defendida por Danillo Moura Santos e aprovada em 31 de março de 2006, em Florianópolis, Santa Catarina, pela banca examinadora constituida por:

Prof. Dr. Antônio Augusto Medeiros Fröhlich
Orientador

Prof. Dr. Lúcia Helena Martins Pacheco
Universidade Federal de Santa Catarina

Prof. Dr. Luiz Claudio Villar Santos
Universidade Federal de Santa Catarina

Fauze Valerio Polpeta, M.Sc.
Universidade Federal de Santa Catarina

Resumo

Este trabalho é a análise do projeto de um sistema embarcado seguindo duas estratégias de desenvolvimento de sistemas computacionais embarcados. As duas estratégias são a abordagem tradicional com base em microcontroladores e a abordagem proposta pelo projeto **PDSCE**, que se baseia na metodologia de Projeto de Sistemas Orientados a Aplicação para conduzir o desenvolvimento de sistemas embarcados como agregados de componentes de hardware e de software adaptados e configurados de acordo com os requisitos da aplicação alvo, dando origem à implementações na forma de **SoCs**.

Abstract

This work shows the design analysis of an embedded system based on two design strategies. The strategies are the common methodology used nowadays using microcontrollers as platforms and the **AOSD** methodology used in the project **PDSCE** to build embedded systems as components of software and hardware, adapted and configured to the application requirements, resulting in a **SoC** implementation.

Agradecimentos

Agradeço acima de tudo à minha família, por acreditar em minha capacidade e nunca faltar com o apoio emocional necessário nos momentos difíceis. Agradeço aos amigos, por proporcionarem momentos de descontração necessários à realização de qualquer trabalho. Agradeço também os colegas do laboratório, sem os quais seria muito difícil a realização deste trabalho. Agradeço também a minha namorada, por compreender momentos de ausência e de nervosismo. E por fim agradeço ao meu orientador, por aceitar alguns atrasos e lidar com estes da melhor maneira possível.

Sumário

Lista de Figuras

Lista de abreviaturas e siglas

1 Introdução p. 10

1.1 Motivação . . . . p. 12

1.2 Objetivos . . . . p. 13

1.3 Objetivo Geral . . . . p. 13

1.4 Objetivos Específicos . . . . p. 13

2 Projeto de sistemas embarcados p. 14

2.1 O que são Sistemas embarcados? . . . . p. 15

2.1.1 Principais características . . . . p. 15

2.2 Metodologia no projeto de sistemas embarcados . . . . p. 18

2.3 Níveis de abstração do projeto . . . . p. 18

2.3.1 Requisitos . . . . p. 20

2.3.2 Especificação . . . . p. 21

2.3.3 Arquitetura . . . . p. 21

2.3.4 Componentes . . . . p. 22

2.3.5 Integração do sistema . . . . p. 22

2.4 Fluxo de projeto . . . . p. 22

3 Projeto de sistemas embarcados p. 24

3.1 Projeto baseado em Microcontroladores e componentes discretos . . . . p. 24
3.2 Projeto baseado em plataformas . . . . p. 25
3.3 Projeto de Sistemas Orientados a Aplicação . . . . p. 28
3.3.1 Visão geral de AOSD . . . . p. 28
3.3.2 Famílias de abstrações independentes de cenários . . . . p. 29
3.3.3 Adaptadores de Cenários . . . . p. 30
3.3.4 Interfaces Infladas . . . . p. 30
3.3.5 Mediadores de Hardware . . . . p. 30
3.3.6 Usando mediadores de hardware para inferir componentes de hardware . . . . p. 31

4 Estudo de Caso: Projeto do grampeador de barramentos CAN p. 33
4.1 Requisitos . . . . p. 33
4.1.1 Requisitos não-funcionais . . . . p. 34
4.2 Especificação . . . . p. 34
4.3 Projeto da Arquitetura . . . . p. 36
4.4 Grampeador CAN utilizando microcontrolador . . . . p. 37
4.4.1 Microcontrolador AT90CAN128 . . . . p. 37
4.5 Grampeador CAN segundo AOSD, arquitetura em FPGA . . . . p. 41

5 Resultados e Conclusões p. 44

Referências p. 47

6 Anexos p. 49

Lista de Figuras

1 Níveis de abstração do projeto de sistemas embarcados [Wol01]. . . . . p. 19

2 O conceito de plataforma força a exploração do espaço de projeto a achar uma instância de plataforma dentre todas as possíveis [FV99] . . . . . . p. 27

3 Estrutura de software em camadas [FV99] . . . . p. 28

4 Visão geral da decomposição de domínios segundo AOSD [PF05] . . . . p. 29

5 Especialização de plataformas segundo AOSD [PF05] . . . . p. 32

6 Pacote de dados CAN [AT905] . . . . p. 33

7 Modelo Funcional do grampeador CAN . . . . p. 35

8 Diagrama das classes do projeto . . . . p. 35

9 Diagrama de blocos do grampeador de barramentos CAN . . . . p. 36

10 Pacote a ser enviado pela UART montado a partir de um pacote CAN . p. 36

11 Diagrama de blocos do microcontrolador AT90CAN128 . . . . p. 38

12 Aplicação do Grampeador CAN desenvolvida para o EPOS . . . . . . p. 40

13 Diagrama de blocos do grampeador de barramentos CAN implementado em FPGA . . . . p. 43

Lista de abreviaturas e siglas

AOSD - *Application Oriented System Design*

CAD - *Computer-aided Design*

CAN - *Controller Area Network*

DLC - *Data Length Code*

FPGA - *Field-Programmable Gate Array*

PC - *Personal Computer - IBM*

PDSCE - Plataforma de Sesenvolvimento de Sistemas Computacionais Embarcados

RISC - *Reduced Instruction Set Computer*

RTR - *Remote Transmission Request*

SoCs - *Systems-on-Chip*

UML - *Unified Modeling Language*

1 Introdução

Este trabalho é a análise do projeto de um sistema computacional embarcado seguindo duas estratégia de desenvolvimento. Serão analisadas a abordagem tradicional que faz uso de microcontroladores e componentes discretos e a abordagem proposta pelo projeto PDSCE, que se baseia na metodologia de Projeto de Sistemas Orientados a Aplicação (AOSD), no projeto de um grampeador de barramentos CAN. A metodologia AOSD, empregada no projeto PDSCE, mostrou-se adequada em alguns estudos para o desenvolvimento de software para sistemas embarcados [Frö01]. O desenvolvimento de sistemas orientados a aplicação possibilita o desenvolvimento de sistemas embarcados a partir de componentes de hardware e de software especificamente adaptados e configurados de acordo com os requisitos da aplicação alvo.

Este trabalho, como dito anteriormente, está inserido no projeto PDSCE, que é financiado pela FINEP e realizado em parceria com mais três universidades brasileiras (UFSM, PUC-RS, UNISINOS) e a empresa Taotrônics. O Projeto PDSCE tem por objetivo conceber uma plataforma de suporte ao desenvolvimento de sistemas computacionais embarcados de acordo com a metodologia de Projeto de Sistemas Orientado a Aplicações. A plataforma almejada consiste de um conjunto de ferramentas: análise de requisitos de aplicação, descrição de componentes de software e hardware, gestão de conhecimento de configuração e geração automática. Adicionalmente, o projeto investe esforços no desenvolvimento de componentes de software e hardware, bem como de aplicações que possam validar o ferramental construído.

Sistemas embarcados estão cada vez mais presentes em nosso cotidiano, o baixo custo tecnológico permitiu o aumento da capacidade do hardware, viabilizando a implementação de aplicações mais complexas. O surgimento de novas aplicações é impulsionado pelo crescimento do mercado e um melhor conhecimento das necessidades humanas. No entanto, a complexidade destas novas aplicações reflete-se no processo de seu desenvolvimento. Estudos vêm sendo realizados com o objetivo de estabelecer uma metodologia que facilite este processo. Facilitar este processo significa reduzir o tempo de projeto e desen-

volvimento, reduzir os custos de um novo projeto e diminuir gastos recorrentes (conserto de *bugs* e surgimento de novas funcionalidades). No capítulo 2.2 são listados os estudos mais relevantes.

Gordon Moore, fundador da Intel, em 1965 constatou que a cada 18 meses a quantidade de transistores dentro de um *chip* dobra (conseqüentemente a sua capacidade de processamento), enquanto os custos permanecem constantes. A lei de Moore, como é conhecida, se mantém até hoje e este aumento na capacidade dos *chips* que possibilita o desenvolvimento de sistemas embarcados que executam aplicações cada vez mais complexas. O constante crescimento do número de transistores dentro de um *chip* tornou possível o desenvolvimento de SoCs, que são sistemas inteiros com processador, memória e periféricos dentro de um só circuito integrado. Além do avanço na capacidade dos *chips*, houve um avanço também nas ferramentas de CAD que auxiliam os projetistas no desenvolvimento.

O termo sistema embarcado (*Embedded System*) não possui uma definição direta e universal, abaixo estão algumas definições propostas:

- Sistemas embarcados são sistemas de processamento de informações que estão embarcados em sistemas maiores e que normalmente não são visíveis ao usuário [Mar03];
- Sistemas embarcados são sistemas onde hardware e software normalmente são integrados e seu projeto visa o desempenho de uma função específica. A palavra embarcados leva a idéia que estes sistemas são parte de um sistema maior. Este sistema maior pode ser composto por outros sistemas embarcados [LY03];
- Um sistema embarcado é um sistema baseado em um microprocessador, que é projetado para controlar uma função ou uma gama de funções, e não para ser programado pelo usuário final como acontece com os PCs [Hea03];
- Um sistema embarcado é qualquer aparelho que possua um computador programável mas este não é projetado para ser um computador de uso geral [Wol01];

Este trabalho é composto de três partes, descritas a seguir:

Estudo e implementação de sistemas embarcados utilizando-se uma abordagem tradicional

Estudo da arquitetura dos microcontroladores AVR da Atmel, amplamente utilizados. Implementação de aplicações nesta arquitetura.

Estudo do desenvolvimento de sistemas embarcados Orientados a Aplicação

Estudar a metodologia de Desenvolvimento de Sistemas Orientados a Aplicação utilizada no PDSCE e focar o seu uso no desenvolvimento de hardware e software embarcados.

Análise do projeto seguindo as duas estratégias

Analisar os resultados do desenvolvimento utilizando as duas estratégias, ressaltando vantagens e desvantagens encontradas nos dois projetos.

1.1 Motivação

Segundo pesquisas [Ten00], é estimado que 80% de todos os processadores existentes hoje são usados em sistemas embarcados. Em 1996, foi estimado que um cidadão estadunidense adulto tinha contato com cerca de 60 microprocessadores por dia. A jornalista estadunidense Mary Ryan fez uma citação em 1995 que dizia : '...sistemas embarcados são a base do mundo eletrônico que vivemos hoje... eles são parte de quase tudo que utiliza energia elétrica.'.

Com a acirrada competição pelo mercado de produtos eletrônicos, as empresas buscam metodologias para desenvolver seus produtos de maneira mais rápida (menor *time-to-market*) e eficiente, para assim lançar produtos novos antes de seus concorrentes. Atrasos podem inviabilizar comercialmente um projeto, poucas semanas de atraso para o lançamento de um novo produto pode comprometer os ganhos estimados, além disso o tempo de mercado dos produtos é cada vez menor, sendo que o ciclo de vida pode chegar a apenas alguns meses.

O estabelecimento de uma metodologia que aborde o desenvolvimento de um sistema do príncipio ao fim do projeto, permitiria o desenvolvimento de ferramentas de CAD eficientes e possibilitaria a engenharia concorrente, ou seja, o desenvolvimento de diferentes partes do sistema em paralelo por diferentes equipes, com troca de informações entre estas, uma vez que todos conhecem todo o processo seria fácil manter um *track* do projeto.

Este é o ponto chave deste trabalho, identificar características positivas e negativas na metodologia de projeto de sistemas embarcados orientada a aplicação e o desenvolvimento utilizando componentes discretos. Esse estudo pretende contribuir para o aperfeiçoamento da metodologia de desenvolvimento de sistemas orientados a aplicação.

1.2 Objetivos

Realizar uma análise do projeto de sistemas embarcados seguindo duas abordagens de desenvolvimento, a metodologia de desenvolvimento de sistemas orientados a aplicação AOSD utilizada no projeto PDSCE aplicada também ao desenvolvimento de harware e o desenvolvimento de sistemas embarcados utilizando microcontroladores e componentes discretos.

1.3 Objetivo Geral

Estudar o desenvolvimento de sistemas embarcados usando a metodologia de desenvolvimento orientado a aplicação, comparando esta metodologia com o desenvolvimento de sistemas embarcados utilizando microcontroladores e componentes discretos.

1.4 Objetivos Específicos

Aplicar no desenvolvimento de sistemas embarcados o projeto de sistemas orientados à aplicação proposto por Fröhlich em [Frö01] e estendida por Polpeta em [PF05], visando a criação de um protótipo de um grampeador de barramentos CAN (*Controller Area Network*).

Estudar a arquitetura de microcontroladores AVR fabricados pela Atmel e as ferramentas utilizadas para desenvolvimento nesta plataforma. Este estudo será guiado pela criação de um protótipo de um grampeador de barramentos CAN, utilizado no monitoramento de barramentos deste tipo.

Estudar o desenvolvimento de SoCs, suas características e vantagens. Paralelamente a este estudo, realizar a criação de aplicações em FPGAs, utilizadas comumente como uma plataforma de prototipação de SoCs, que auxilia a verificação de projetos digitais antes da confecção do hardware.

2 Projeto de sistemas embarcados

A maioria dos processadores hoje fabricados são utilizados em sistemas embarcados [Ten00]. O uso de processadores em sistemas embarcados teve início antes do surgimento dos computadores pessoais, no início esses sistemas eram projetados principalmente para realizar funções de controle. Ainda hoje, existem aplicações de controle atendidas por sistemas embarcados. Porém existem novas aplicações que demandam grande capacidade de processamento, tais como processamento de sinais e aplicações multimedia que são possíveis graças a evolução tecnológica. Um videogame Playstation II, por exemplo, produzido pela Sony que possui um processador RISC de 128 bits que opera a 300MHz e tem capacidade de processamento superior aos supercomputadores da década de 80.

Estamos cercados por sistemas embarcados, eles estão cada vez mais presentes em nosso dia-a-dia, máquinas de lavar, televisões, eletrodomésticos em geral possuem algum tipo de processamento, automóveis, caixas de banco eletrônicos, equipamentos de comunicação como modems, roteadores, etc. são todos sistemas processados, onde algum tipo de informação é manipulada. Em uma máquina de lavar moderna, somos capazes de escolher o tipo de programa de lavagem e a duração da lavagem, estas escolhas são entradas (parâmetros) de um algoritmo que irá controlar a lavagem em si, de acordo com as opções escolhidas.

O baixo custo proporcionado pelos avanços tecnológicos possibilitaram esse cenário, e nos faz imaginar o que está por vir. Com o aumento da capacidade dos circuitos integrados, os eletrodomésticos se tornam cada vez mais "inteligentes". No início, os telefones celulares tinham apenas a função de um telefone, hoje além de telefone são também câmeras fotográficas, terminais de acesso a Internet, agendas eletrônicas e novas funcionalidades surgem a cada dia. Casas inteligentes já são realidade, apesar dos preços não serem acessíveis a grande parte da população, ambientes com sensores que sabem quando acender e apagar as luzes, sistemas de segurança silenciosos capazes de realizar chamadas telefônicas, eletrodomésticos que podem ser acessados via Internet, já estão presentes em uma casa inteligente.

Neste cenário, onde o processamento é realizado pelos processadores presentes em sistemas embarcados, ao invés de computadores pessoais ou servidores, deu origem ao termo **computação invisível** (*disappearing computer*) [Mar03], ou seja, ao invés de um aparelho específico para o processamento de informação, como o computador, o processamento estará em aparelhos que possuem outras funcionalidades. A presença de microprocessadores nestes aparelhos não será tão óbvia, por isso o nome *computação invisível*, porém como os processadores não irão desaparecer, um nome mais adequado para a língua portuguesa seria *computação onipresente*, evidenciando o processamento de informação em todos os lugares.

2.1 O que são Sistemas embarcados?

Como supracitado, sistemas embarcados não possuem interface com o usuário e executam uma função específica. Estes sistemas possuem outras características comuns, que podem auxiliar no seu projeto, estas características são mostradas a seguir.

2.1.1 Principais características

A principal característica de um sistema embarcado e comum a todos é que estes são sistemas que manipulam dados dentro de sistemas ou produtos maiores. Além desta característica, existem outras, identificadas por Peter Marwedel em [Mar03], listadas a seguir:

- Sistemas embarcados são projetados para realizar uma função ou uma gama de funções e não para serem programados pelo usuário final, como os computadores pessoais. O usuário, pode alterar ou configurar a maneira como o sistema se comporta, porém não pode alterar a função que este realiza.
- Sistemas embarcados normalmente interagem com o ambiente em que se encontram, coletando dados de sensores e modificando o ambiente utilizando atuadores.
- Sistemas embarcados devem ser confiáveis. Muitos destes sistemas realizam funções críticas, onde falhas podem causar catástrofes. A principal razão para que estes sistemas sejam a prova de falhas, é que eles interagem com o meio, causando impactos a este. Dizer que um sistema é confiável, significa que este possui certas características, listadas a seguir:

  - Estabilidade: é a probabilidade que um sistema não irá falhar.
  - Recuperação: é a probabilidade que uma falha no sistema será corrigida em um certo intervalo de tempo.
  - Disponibilidade: é a probabilidade de que um sistema estará disponível em certo tempo. Alta estabilidade e recuperação levam a uma alta disponibilidade.
  - Segurança: Um sistema deve ser seguro em dois aspectos. Ele deve ser seguro para o meio ambiente, ou seja, uma falha não acarreta em danos ao meio ou as pessoas que utilizam este sistema, e ele deve manter as informações confidencias dentro dele, sem permitir que pessoas não autenticadas manipulem estas informações.

- Sistemas embarcados possuem outras métricas de eficiência além das já conhecidas por projetistas de sistemas *desktop* e servidores. Abaixo estão algumas delas:

  - Consumo de Energia: Tendo em vista que muitos sistemas embarcados são móveis, e são alimentados por baterias, estes devem ser projetados para consumir o mínimo de energia possível. A tecnologia na fabricação de baterias evolui em ritmo muito menor que as aplicações que fazem uso destas baterias, logo a energia disponível deve ser aproveitada da melhor maneira possível.
  - Tamanho de código: Todo código da aplicação que é executada em um sistema embarcado deve estar presente neste, quase sempre em memória. Muito raramente sistemas embarcados possuem dispositivos de armazenamento magnético, como discos rigídos, para armazenar código, logo a disponibilidade de memória é muito limitada, por isso o desperdício de memória deve ser evitado pelos programadores de software embarcado.
  - Execução eficiente: O uso de recursos de hardware deve ser restrito ao que é realmente necessário para implementar uma certa função. Os requisitos de tempo devem ser satisfeitos empregando-se o mínimo de recursos possível e minimizando o consumo de energia. Visando reduzir o consumo de energia, a freqüência do *clock* e a tensão de alimentação devem ser as menores possíveis. Somente os componentes de hardware necessários devem estar presentes. Um sistema embarcado que não realiza conversão de tensão analógica para valores digitais não necessita de conversores analógico digital, a presença deste hardware ocupará espaço e causará um consumo de energia desnecessário. Componentes que não reduzem o pior tempo de execução podem ser omitidos.

  - Peso: Sistemas móveis devem ser leves. Os celulares de primeira geração mostram que sistemas móveis pesados tendem a desaparecer.
  - Custo: O uso eficiente de componentes de hardware pode baixar o custo final de um sistema embarcado, no mercado de eletrodomésticos por exemplo, a competitividade é importantíssima, e o custo é um fator decisivo para dizer se um produto possui mercado.

- Grande parte dos sistemas embarcados não possui teclados, *mouse*, monitores ou outros dispositivos encontrados em computadores pessoais para realizar interfaceamento com o usuário. Sistemas embarcados possuem interfaces dedicadas, como botões, *leds* e chaves. Por isso dificilmente o usuário reconhece a informação sendo transmitida ou processada dentro deles.

- Muitos sistemas embarcados possuem requisitos de tempo real. Não completar uma tarefa em um tempo determinado pode resultar em perda de dados e conseqüentemente de qualidade (aplicações multimídia) ou causar danos. O não cumprimento de um requisito de tempo real pode resultar em catástrofe. Sistemas de tempo real não devem utilizar componentes ou técnicas que diminuem o tempo de processamento na média, como memórias *cache*. Em sistemas de tempo real, uma resposta do sistema deve ser explicada e comprovada sem argumentos estatísticos [Kop97].

- Muitos sistemas embarcados são híbridos, pois são compostos por partes analógicas e partes digitais. As partes analógicas utilizam sinais contínuos em valores de tempo contínuos, e as partes digitais usam sinais discretos no tempo discreto.

- Tipicamente, sistemas embarcados são reativos ao ambiente, ou seja, eles estão em interação contínua com o ambiente e executam em um ritmo determinado por este. Pode-se dizer que um sistema reativo encontra-se em um estado, esperando por uma entrada. Para cada entrada recebida, ele realiza o processamento da informação e gera uma saída. Autômatos são exemplos de sistemas reativos.

A presença de características comuns a vários sistemas embarcados justifica o estudo de metodologias de desenvolvimento. Tendo em vista que o projeto de um novo sistema pode e deve utilizar componentes de hardware e software desenvolvidos em outros projetos, o estabelecimento de uma metodologia permite às equipes de desenvolvimento focar o projeto em características ainda não desenvolvidas para um determinado sistema, utilizando componentes de projetos anteriores.

2.2 Metodologia no projeto de sistemas embarcados

Segundo a definição do dicionário *online* Houaiss uma metodologia é um corpo de regras e cuidados estabelecidos para realizar uma pesquisa, ou seja, um método estabelecido de como fazer. Logo, uma metodologia de projeto de sistemas embarcados estabelece os passos a serem seguidos durante a realização do projeto. Uma estabelece um processo completo que rege todo o desenvolvimento de um sistema. Este processo rege todos os passos do projeto, desde a análise de requisitos até a manutenção e possíveis atualizações do sistema. O estabelecimento de uma metodologia impõe regras e padroniza procedimentos a serem seguidos pelos projetistas e desenvolvedores.

Segundo Waine Wolf [Wol01] utilizar uma determinada metodologia no desenvolvimento de um sistema é importante por três razões principais:

- Permite que a equipe de desenvolvimento mantenha anotações a respeito do projeto, possibilitando a confirmação de que tudo que era para ser feito foi realmente feito, certifica que requisitos como otimização de performance e redução de tamanho foram cumpridos.
- Permite o desenvolvimento de ferramentas de CAD (*Computer-aided design*) mais eficientes. Desenvolver um programa que receba um conceito chave de sistemas embarcados como entrada e gere um projeto completo é uma tarefa muito difícil, porém, dividindo o processo em passos, seria mais fácil trabalhar na automatização (ou semi-automatização) dos passos, um de cada vez.
- Uma metodologia de projeto facilita a comunicação entre os membros de uma equipe de desenvolvimento. Tendo um processo definido, os membros da equipe entedem com mais facilidade o que devem fazer, o que devem esperar de outros membros da equipe em certos momentos e o que eles devem entregar quando completarem todos os passos que deviam cumprir.

Tendo em vista que a maioria dos sistemas embarcados são desenvolvidos por equipes, a coordenação é a função mais importante de uma metodologia bem definida.

2.3 Níveis de abstração do projeto

O projeto de sistemas embarcados em geral segue alguns passos. Estes podem ser vistos na Figura 1. Na visão de projeto *top-down*, a primeira etapa é o levantamento

de requisitos, seguido pela especificação, onde é criada uma descrição detalhada do que queremos. Na especificação, descrevemos como o sistema se comporta, e não como ele é construído. O detalhamento interno do sistema começa a aparecer quando desenvolvemos a arquitetura, a qual resulta na estrutura do sistema em termos de grandes componentes. Nestes componentes abstratos identificamos os módulos de software e os componentes específicos de hardware, componentes ainda não implementados são desenvolvidos como parte do projeto. Com base nestes componentes, podemos finalmente construir o sistema completo. No fim do projeto está a integração e testes dos componentes de hardware e dos componentes de software, fase em que grande parte dos *bugs* são descobertos.

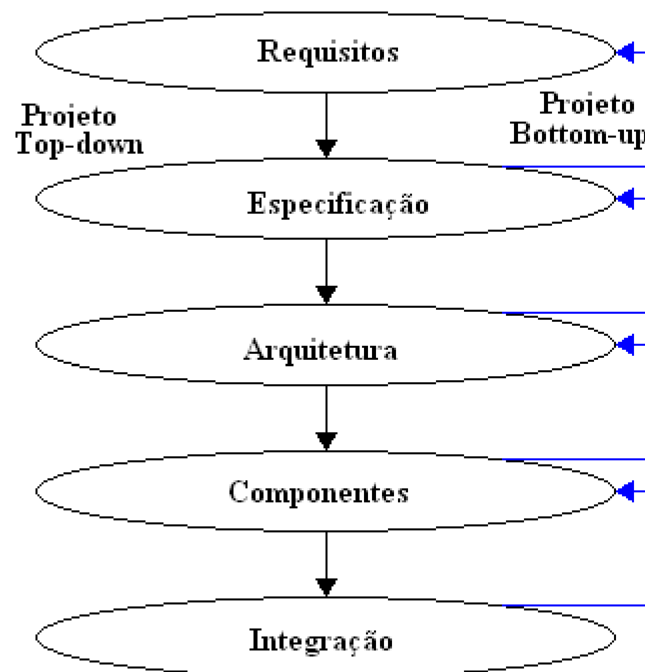

Figura 1: Níveis de abstração do projeto de sistemas embarcados [Wol01].

Na visão *top-down*, o projeto inicial é uma versão abstrata do sistema a ser desenvolvido, que é detalhada a cada passo, adicionando-se detalhes em cada fase do projeto, resultando no sistema concreto bem detalhado. Uma outra alternativa é a visão *bottom-up*, onde começamos com os componentes para construir o sistema. Os passos da visão *bottom-up*, podem ser vistos nas setas com linhas da cor azul. Muitas vezes o projeto *bottom-up* é utilizado para conhecer o funcionamento e propriedades de alguns componentes antes do fim do projeto. Essas propriedades podem ser: tempo de execução de uma determinada função, quanta memória será necessária, quanto da capacidade do barramento do sistema que será utilizado, entre outras. Isso é necessário para que possa ser

feita uma estimativa realista do comportamento do sistema como um todo.

Durante estas etapas, o projeto de sistemas embarcados deve considerar também os requisitos não-funcionais:

- Custo de produção
- Performance
- Consumo de Energia
- Interface com o usuário

A cada etapa de refinamento do projeto, estes requisitos devem ser verificados para certificar que o sistema obedece especificação inicial. A corretude de um sistema embarcado está atrelada ao cumprimento destes requisitos, logo, estes devem ser verificados previamente e não apenas no fim do projeto.

2.3.1 Requisitos

Nesta fase do projeto buscamos junto ao cliente ou solicitante do projeto informações sobre o que estamos projetando. As funções do sistema são descritas de maneira informal, de forma que sejam todas listadas de maneira clara e objetiva, evitando ambigüidades. O sucesso do projeto depende do levantamento de requisitos, um levantamento de requisitos mal feito pode confundir o desenvolvedor e levar a implementações errôneas.

Os requisitos podem ser de dois tipos, funcionais e não-funcionais. O primeiro diz respeito ao funcionamento do sistema, requisitos funcionais dizem o que o sistema deve fazer, quais suas funções. Já os requisitos não-funcionais refletem propriedades do sistema, no caso de sistemas embarcados é comum encontrar nos requisitos não-funcionais a performance do sistema, o custo de produção, o tamanho, o peso e o consumo de energia. Sendo este último importante em equipamentos alimentados por baterias.

O desenvolvimento de um protótipo em um PC pode facilitar o entendimento dos requisitos. Isto é interessante para que o cliente e o projetista saibam o que será o produto final, evitando surpresas ao longo do projeto.

2.3.2 Especificação

Na especificação os requisitos são descritos de uma maneira formal. Alguns projetistas entedem levantamento de requisitos e especificação como sinônimos. Aqui é estabelecido um contrato com o cliente, que avalia as especificações e se diz de acordo com elas. As especificações devem estar descritas em uma linguagem formal que não permita ambiguidades e possa ser utilizada durante o processo de desenvolvimento, porém deve também ser compreensível para o cliente.

Existem linguagens de especificação que facilitam este processo, dentre as mais comuns podemos citar a UML [BJ99] que permite a especificação estrutural e comportamental do sistema. UML possibilita diferentes níveis de abstração, o que facilita o processo de refinamento e adição de detalhes a cada nível do projeto. Além disso, é possível mapear objetos reais, como sensores em objetos na especificação, com interfaces e funções definidas em classes. É possível ainda, modelar o mundo externo utilizando UML. Assim objetos que interagem com o sistema também podem fazer parte da especificação.

A fase de especificação deixa claro o que o sistema deve fazer, modelando os componentes e módulos deste. Porém aqui ainda não é especificado como ele será implementado, isso faz parte do projeto da arquitetura, descrito à seguir.

2.3.3 Arquitetura

No projeto da arquitetura o sistema é descrito funcionalmente. São usados diagramas de blocos para mostrar como os componentes e módulos do sistema são conectados. As funções de cada componente são descritas e é feita a identificação de quais componentes de software e de hardware serão necessários no projeto do sistema.

Inicialmente o sistema é visto como um todo, identificando as funções que este deve cumprir. Seguindo o processo de refinamento sucessivo, os blocos de cada função são descritos em seguida. Por exemplo, se o sistema requer a realização de comunicação serial, identificamos aqui o uso de uma UART, porém fica a cargo do projetista as propriedades desta UART. Um bloco de funções gerais do sistema pode ser refinado em dois blocos: hardware e software. A partir daí os blocos são transformados em componentes.

Em todos os passos do projeto deve-se ater aos requisitos não-funcionais, ou seja, a cada refinamento deve-se verificar o cumprimento dos requisitos iniciais, como performance e consumo de energia. O conhecimento prévio do funcionamento dos componentes

selecionados ajudam o obedecimento desses requisitos.

Uma técnica utilizada no projeto da arquitetura é o CRC (Classes, Responsabilidades e Colaboradores) *card* que diz o que cada componente selecionado deve fazer e com quais outros componentes ele se comunica.

2.3.4 Componentes

O projeto de componentes é necessário quando algum componente requerido pelo sistema não está pronto e deve ser implementado ou adaptado.

O desenvolvimento de componentes é feito de acordo com as interfaces de barramentos utilizadas no sistema. Componentes de software e componentes de hardware projetados para um determinado sistema deve ter interfaces compatíveis com as interfaces já presentes no sistema.

O desenvolvimento de um componente deve seguir também os passos do projeto de todo um sistema (Requisitos, Especificação, Arquitetura e testes).

2.3.5 Integração do sistema

Normalmente, nesta etapa do projeto são descobertos grande parte dos *bugs*. A integração é a etapa mais desafiadora do projeto, pois é quando os trabalhos de diferentes equipes são unidos.

Algumas medidas podem ser adotadas para diminuir os esforços nesta fase. Depurar módulos e componentes separadamente é uma boa medida para evitar que *bugs* referentes ao funcionamento dos componentes só sejam descobertos nessa etapa.

A depuração de sistemas embarcados é mais complexa que o de sistemas de software convencionais. A ausência de uma interface de debug dificulta este processo. Por isso, atualmente projetistas desenvolvem componentes que auxiliam esse processo, como interfaces JTAG.

2.4 Fluxo de projeto

Além dos fluxos de projeto top-down e bottom-up existem outros que também obedecem quase todas as fases explicadas acima. Dentre estes estão:

- Cachoeira (*Waterfall*): Neste fluxo de projeto, a visão top-down é seguida a risca, cada fase de projeto é executada uma vez e não há *feedback* da fase seguinte para a fase anterior. Essa abordagem não é muito usada pois em diversos projetos os desenvolvedores realizam testes em uma fase inferior (menos abstrata) para voltar à fase mais abstrata e tomar decisões baseadas nestes testes.

- Espiral: Esta abordagem prevê o desenvolvimento de diversos projetos, mais detalhados a cada espiral, ou seja, a cada fluxo completo de projeto. O protótipo inicial tende a ser bastante simples, realizando apenas algumas funções do sistema final. A cada novo protótipo mais detalhes são adicionados, tornando o sistema cada vez mais próximo do final. Esse fluxo de projeto pode tornar-se muito longo caso sejam necessários muitos protótipos.

- Refinamentos sucessivos: Ideal quando a equipe não detém muito conhecimento a respeito do projeto a ser desenvolvido. Refinamentos do sistema são realizados até quando necessário. Um projeto inicial simples é construído e serve de guia para a equipe até o momento que tem-se uma visão global e realista do projeto como um todo.

Normalmente, no projeto de sistemas embarcados considera-se o desenvolvimento do software e do hardware, mesmo no caso em que o hardware como um todo não é desenvolvido no projeto, ainda há a escolha de uma placa e de alguns componentes a serem utilizados. Visando o encurtamento do tempo de desenvolvimento do projeto, pode-se realizar a especificação e o estabelecimento da arquitetura e depois criar duas equipes diferentes, uma para o desenvolvimento do hardware e outra para o desenvolvimento do software. Esse processo é chamado de engenharia concorrente e pode ser aplicado também dentro destas duas equipes iniciais, a equipe de software por exemplo, poderia ser dividida em uma equipe que desenvolve a aplicação e outra que desenvolve os componentes a serem utilizados pela outra equipe.

3 Projeto de sistemas embarcados

3.1 Projeto baseado em Microcontroladores e componentes discretos

O desenvolvimento clássico de sistemas embarcados normalmente possui como núcleo do sistema um microcontrolador e alguns outros circuitos integrados (ou componentes discretos) que atendem funções específicas, como comunicação e controle de atuação no meio. Como esse processo tem sido usado há décadas, a existência de diversas ferramentas o facilitam. Muitas das arquiteturas modernas de microcontroladores possuem diversos periféricos (como hardware de comunicação serial, *timers* etc.) em um simples *chip*. Compiladores de linguagens de alto nível estão disponíveis para as arquiteturas mais comuns de microcontroladores.

O projetista do sistema, seguindo esta estratégia, identifica os requisitos da aplicação e procura por uma arquitetura adequada. Esta busca deve considerar alguns aspectos, como preço do microcontrolador, periféricos disponíveis, ferramentas de desenvolvimento existentes para aquela arquitetura, como compiladores, simuladores e *debuggers*, formatos de CIs (Circuitos Integrados) disponíveis (Microcontroladores de uma mesma arquitetura são disponibilizados com diferentes características. Por exemplo, diferentes encapsulamentos, número de pinos e características físicas) e algumas outras características. O desenvolvedor, deve também avaliar se uma arquitetura com diversos periféricos é realmente a opção mais satisfatória, ou, uma arquitetura mais simples com alguns CIs externos não atenderia os requisitos da aplicação e ainda teria um custo menor. Essa estratégia de desenvolvimento é muito dependente da experiência do projetista, a escolha da arquitetura é certamente o núcleo do projeto.

Apesar de existirem diferentes opções de microcontroladores de uma mesma arquitetura, com diferentes opções de periféricos, em muitas aplicações, nem todos os periféricos do microcontrolador são usados, o que resulta em *overhead* desnecessário (de consumo de potência ou mesmo área) para aplicação. Nessa abordagem de projeto, arquiteturas de 8

bits são muito comuns, pois diversas aplicações não necessitam de grande capacidade de processamento, e sim controle.

3.2 Projeto baseado em plataformas

O projeto baseado em plataformas é utilizado na computação há décadas. Um exemplo disso é o PC, uma arquitetura estabelecida há muito tempo que continua evoluindo. O PC possui um conjunto de instruções (ISA - *Instruction Set Architecture*) que deve ser obedecido por todos as instâncias que implementam esta plataforma. Além do conjunto de intruções, algumas outras propriedades foram sedimentadas na arquitetura x86, como barramentos PCI, USB, ISA e ainda um conjunto de dispositivos de entrada e saída, como teclados, *mouses*, monitores, impressoras entre outros. Estes padrões permitem aos desenvolvedores de periféricos e placas de extensão a abstração da plataforma alvo. Por exemplo, um desenvolvedor de placas de vídeo PCI não precisa levar em conta no projeto se a placa será instalada em um PC com processador Intel ou AMD, a sua responsabilidade é que a placa seja compatível com o padrão de barramento PCI.

A metodologia de desenvolvimento baseado em plataformas proposta por Vincentelli [VC01] é bastante difundida, e tem sido sido bastante utilizada no desenvolvimento de sistemas embarcados.

Esta metodologia foi criada com o intuito de diminuir os custos de desenvolvimento, de produção e o tempo de projeto de sistemas eletrônicos, sacrificando o mínimo possível (quando necessário) a performance. O desenvolvimento baseado em plataformas consiste em estabelecer uma plataforma que atenda uma determinada gama de aplicações, por exemplo aplicações multimedia, e possa ser usada em diferentes projetos. Segundo o próprio Vincentelli uma plataforma é uma camada de abstração do projeto que facilita alguns refinamentos, que levam a uma outra camada de abstração, e após ciclos de refinamento sucessivos, tem-se a plataforma de hardware e software desejada. Esta metodologia tem sido bastante usada, exemplos disso são a plataforma Nexperia da Philips [Phi04] para aplicações multimedia e a TI OMAP da Texas Instrument [Ins04] para telefones celulares.

Em [FV99], Ferrari e Vincentelli definem uma plataforma de hardware como sendo uma família de arquiteturas que satisfazem um conjunto de restrições que são impostos para permitir o reuso de componentes de hardware e de software. O número e rigidez das restrições impostas definem o nível e o grau de reusabilidade da plataforma. Quanto mais rígidas essas restrições, maior o grau de reusabilidade obtido. Porém, restrições rígidas

implicam em menos arquiteturas a serem escolhidas e menos otimizações dependentes da aplicação. O próprio PC é um bom exemplo de arquitetura de hardware com muitas restrições que permite o reuso do projeto. Algumas restrições impostas ao PC são:

- O conjunto de Instruções (ISA) que permite o reuso de sistemas operacionais e aplicações em código binário.
- Diversos barramentos bem especificados como ISA, PCI, USB que permite o uso de placas de expansão e de dispositivos externos.
- Especificação de diversos dispositivos de entrada e saída, teclados, áudio e vídeo.

Essas restrições são obedecidas por todos os PCs. A liberdade do desenvolvedor de PCs fica bastante limitada devido a estes fatores. Porém, sem essas restrições dificilmente o PC seria o que é hoje. Muitos perguntam porque isto não foi possível em sistemas embarcados, e a resposta é uma característica marcante nestes: custo. Os PCs são muitas vezes sub-utilizados, ou seja, sobra capacidade de hardware para a aplicação que estes se destinam, o que não seria aceitável para sistemas embarcados. Na verdade os projetistas de hoje já pensam em super-dimensionamento destes sistemas, para reduzir custos e diminuir o tempo de projeto, porém este super-dimensionamento não pode ser igual ao feito nos PCs. Pois o custo final de sistemas embarcados simples cresceriam demasiadamente.

Realizada a escolha de uma plataforma de hardware, refinamentos devem ser realizados para obter uma instância desta plataforma. Alguns detalhes são adicionados, obedecendo as restrições impostas pela plataforma. São definidos mecanismos de comunicação e componentes necessários para obter a plataforma final. Estas escolhas propiciam uma exploração do espaço de projeto, verificando as propriedades da instância da plataforma com os componentes e restrições implementadas.

A Figura 2 mostra graficamente as propriedades de uma plataforma de hardware. O cone superior representa a flexibildiade e o cone inferior representa a generalidade. Em geral, plataformas de hardware tendem a ter um cone grande em cima e um cone pequeno embaixo para comprovar quão importante o reuso do projeto e de padrões são em relação às otimizações obtidas com a perda de restrições.

Para permitir o nível de reuso do software que buscamos o conceito de plataforma de hardware isolado não é suficiente. Para que seja útil, a plataforma de hardware deve ser abstraída de uma maneira que o software da aplicação veja uma interface de alto-nível, conhecida por API (*Application Program Interface*). Uma camada de software deve ser

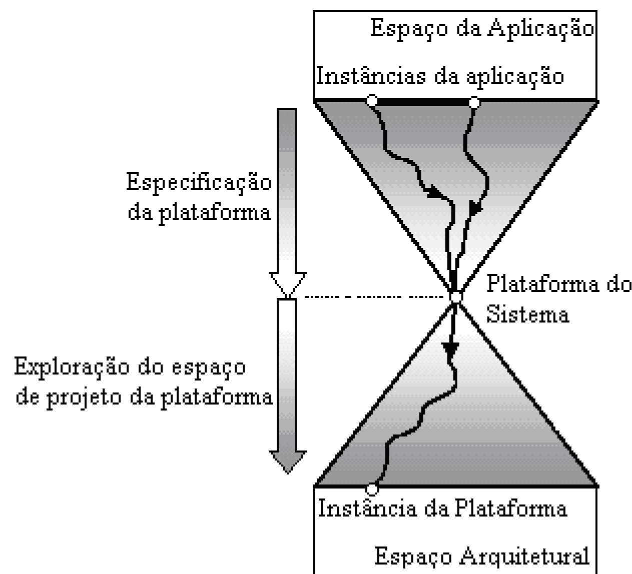

Figura 2: O conceito de plataforma força a exploração do espaço de projeto a achar uma instância de plataforma dentre todas as possíveis [FV99]

usada para prover esta abstração. Essa camada de software engloba as partes essenciais da plataforma de hardware, como:

- Os *cores* programáveis e o sub-sistema de memória são englobados por um RTOS
- O sub-sistema de entrada e saída são abstraídos pelos *drivers* de dispositivos
- A conexão de rede é abstraída por um sub-sistema de redes de comunicação

Essa camada de software é chamada plataforma de software. A Figura 3 mostra graficamente a estrutura de uma plataforma de software.

O desenvolvimento de uma plataforma que atenda os requisitos das aplicações às quais esta se destina não é trivial. O tempo de projeto deve ser levado em conta para que depois de estabelecida a plataforma, as aplicações que podem fazer uso desta ainda tenham espaço no mercado. Uma plataforma deve passar por uma análise de requisitos e viabilidade (realmente existem aplicações que seriam beneficiadas com o uso desta plataforma) antes de ser projetada.

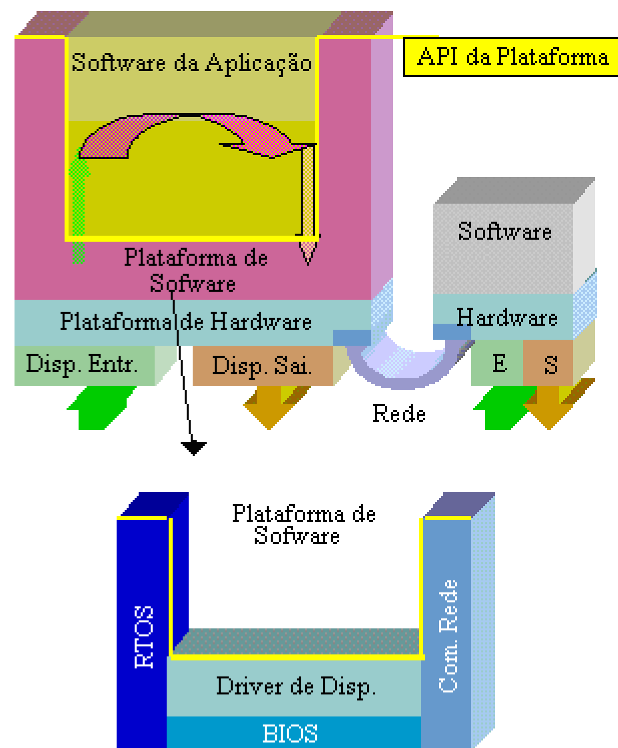

Figura 3: Estrutura de software em camadas [FV99]

3.3 Projeto de Sistemas Orientados a Aplicação

3.3.1 Visão geral de AOSD

A metodologia de desenvolvimento de sistemas orientados a aplicação (AOSD) proposta por Fröhlich em [Frö01] consiste em uma maneira de obter componentes de software através de uma cuidadosa engenharia de domínio. A engenharia de domínio de sistemas operacionais, por exemplo, identifica entidades significantes que compõe sistemas operacionais. Essas entidades são então organizadas em abstrações de membros de famílias, segundo a análise de variabilidade no contexto de *Family Based Design* proposto por [Par76]. No entanto, apesar da criteriosa separação de conceitos, algumas destas abstrações ainda são dependentes do ambiente em que se encontram.

Abstrações de membros que incorporam detalhes do ambiente em que se encontram tem poucas chances de serem reusadas em outros cenários. Considerando que um SO orientado a aplicação está extremamente ligado ao hardware, dependências de ambiente iriam depreciar o seu projeto. A depedência do hardware pode ser reduzida utilizando o conceito de separação de aspectos apresentado em *Aspect Oriented Programming* [KLM+97], no processo de decomposição do domínio. Esse conceito provê meios de identificar variações

de cenários, que ao invés de modelar um novo membro da família, define um aspecto do cenário. Por exemplo, ao invés de modelar um novo membro da família de mecanismos de comunicação que pode operar em cenários *multithreading*, *multithreading* poderia ser modelado como um aspecto do cenário, que quando ativado bloqueia o mecanismo de comunicação (ou a sua operação) quando em uma região de código definida como crítica.

AOSD propõe um procedimento de engenharia de domínio (veja Figura 4) que modela componentes de software utilizando principalmente três contruções: famílias de abstrações independentes de cenários, adaptadores de cenários e interfaces infladas.

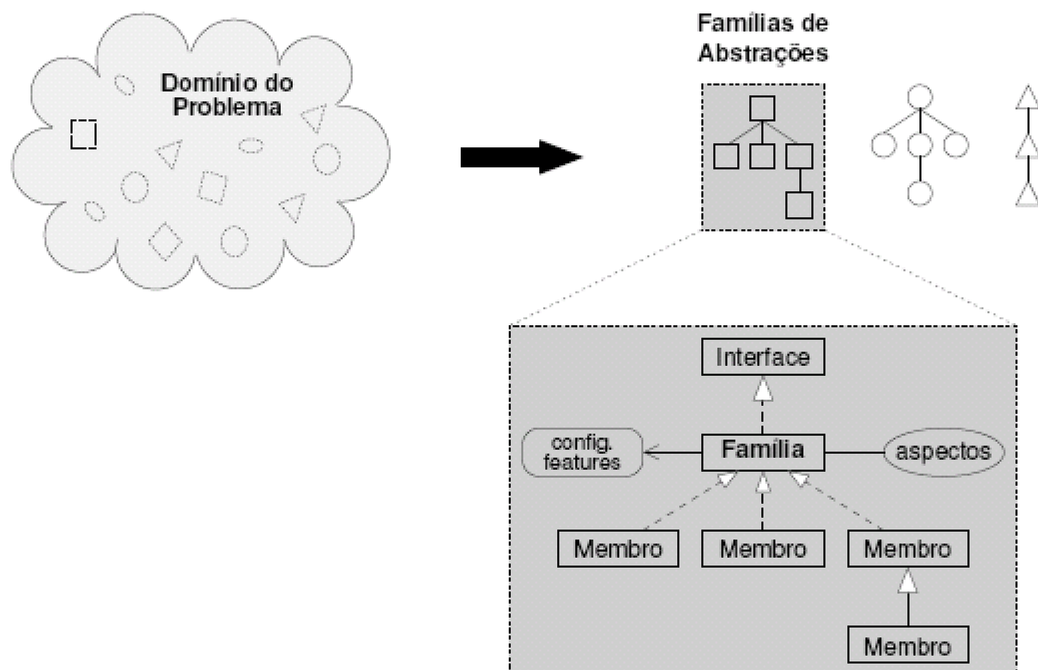

Figura 4: Visão geral da decomposição de domínios segundo AOSD [PF05]

3.3.2 Famílias de abstrações independentes de cenários

Na fase de decomposição do domínio, abstrações são identificadas a partir das entidades do domínio e são agrupadas em famílias, de acordo com suas características comuns. Ao longo dessa fase, também é realizada a separação dos aspectos, para assim modelar abstrações independentes de cenário, o que possibilita o reuso destas em cenários diferentes. Os componentes de software são então implementados de acordo com essas abstrações.

3.3.3 Adaptadores de Cenários

Como explicado anteriormente, de acordo com a metodologia AOSD, as dependências de cenário devem ser fatoradas como aspectos, mantendo assim as abstrações independentes de cenários. Contudo, para que esta estratégia funcione, devem existir modos de aplicar os aspectos às abstrações de maneira transparente. Isso é realizado usando adaptadores de cenários [FSP00], que englobam uma abstração, intermediando sua comunicação com clientes dependentes de cenários para realizar as mudanças necessárias, sem *overhead*.

3.3.4 Interfaces Infladas

Interfaces infladas contém as propriedades de todos os membros de uma família, proporcionando uma visão única da família como se esta fosse um super-componente. Estas permitem que o desenvolvedor escreva a aplicação baseado em uma interface limpa e bem conhecida, adiando a decisão de qual membro da família deve ser usado até o momento que o sistema é gerado. A associação entre uma interface inflada e um membro específico da família, será automaticamente feita pela ferramenta de configuração, que identifica quais propriedades da família foram usadas, para desta maneira escolher o membro mais simples da família que implementa a interface requisitada. Esse membro será então agregado ao SO em tempo de compilação.

3.3.5 Mediadores de Hardware

Objetivando o SO e a sua portabilidade para teoricamente qualquer arquitetura, um sistema desenvolvido segundo AOSD utiliza *mediadores de hardware* [PF04]. A principal idéia deste artefato de portabilidade é manter um *contrato de interface* entre o sistema operacional e o hardware. Cada componente de hardware é acessado a partir do seu próprio mediador, logo, é garantida a portabilidade de abstrações que o utilizam sem a criação de dependências desnecessárias. Mediadores são quase totalmente meta-programados estaticamente e se "dissolvem"nas abstrações de sistema assim que o contrato com a interface é firmado. Em outras palavras, um mediador de hardware abstrai as funcionalidades do componente de hardware correspondente através de uma interface voltada para o sistema operacional.

Mediadores de hardware possuem *propriedades configuráveis*, que permitem alguma propriedades dos componentes de hardware serem ativadas e desativadas, de acordo com os requisitos da abstração. Essas propriedades não são apenas *flags* que podem ser lig-

adas e desligadas, propriedades configuráveis podem ser implementadas utilizando programação genérica que permite a implementação de estruturas e algoritmos em software sem um *overhead* considerável. Um exemplo disso é a geração de código CRC como uma propriedade configurável de um componente de hardware de comunicação.

Como outros componentes em AOSD, os mediadores de hardware são agrupados em famílias, cada membro é a representação de uma entidade do domínio de hardware. Essa abordagem garante que o sistema será composto apenas do código objeto necessário para suportar a aplicação. Aspectos não-funcionais e as propriedades intrínsecas aos mediadores são encapsulados como aspectos de cenários que podem ser aplicados à membros da família quando requisitado.

3.3.6 Usando mediadores de hardware para inferir componentes de hardware

Os mediadores de hardware foram criados inicialmente para facilitar a portabilidade de sistemas desenvolvidos seguindo AOSD. Porém, devido a sua íntima relação com os componentes de hardware, eles podem também ser usados para identificar componentes de hardware que são realmente necessários para construir um sistema que suporte uma aplicação específica. A inferência de componentes de hardware a partir dos mediadores foi proposta por Fauze em [PF05].

No contexto de PLDs (do inglês *Programmable Logic Devices*) onde componentes de hardware (IPs) são implementados utilizando linguagens de descrição de hardware como VHDL e Verilog, mediadores de hardware poderiam ajudar na inferência dos IPs realmente necessários ao sistema e algumas propriedades desses IPs. Esses IPs são identificados assim que os mediadores de hardware são instanciados pela aplicação. Logo, o sistema conterá apenas componentes necessários para suportar o SO e conseqüentemente a aplicação.

Os componentes de hardware adequados ao funcionamento do sistema serão inferidos durante a composição do sistema, se um mediador de NIC (*Network Interface Card*) é usado, por exemplo, um IP de NIC deve ser inferido. Nesse caso, o desenvolvedor terá ainda que escolher um componente de hardware específico que seja uma NIC, porque provavelmente existe mais que um desses componentes. Isso é chamado *seleção de IP combinada*. O dispositivo de hardware é inferido considerando um requisito da aplicação e o desenvolvedor seleciona um IP específico daquele tipo de dispositivo para ser usado.

A seleção do IP a ser usado pode também ser feita seguindo apenas os requisitos da

aplicação, o desenvolvedor não toma nenhuma decisão. Esse cenário é chamado *seleção de IP discreta*. Essa abordagem espera que apenas um IP satisfaça os requisitos da aplicação, uma propriedade da aplicação determina a inferência de um IP específico. Um bom exemplo desse cenário é o uso de memória *paginada* pela aplicação. Isso resultaria na inferência de um IP de MMU com suporte a memória *paginada*.

Existe um terceiro cenário chamado *seleção explícita de IP* onde o programador pode escolher todos os componentes de hardware que serão instanciados independentemente. Isso é útil no caso de algum componente de hardware ser ocultado por abstrações do sistema, o programador ainda pode selecionar o componente que deve ser agregado ao sistema. A seleção de IPs segundo AOSD pode ser vista na Figura 5.

As *propriedades configuráveis* dos mediadores de hardware, explicadas anteriormente, podem ser empregadas nos componentes de hardware. Uma propriedade configurável pode ajudar a inferência de um componente específico. O exemplo do uso de códigos CRC por um dispositivo NIC serve também nesse caso, caso esta propriedade seja habilitada, o IP inferido deve ser capaz de gerar códigos CRC.

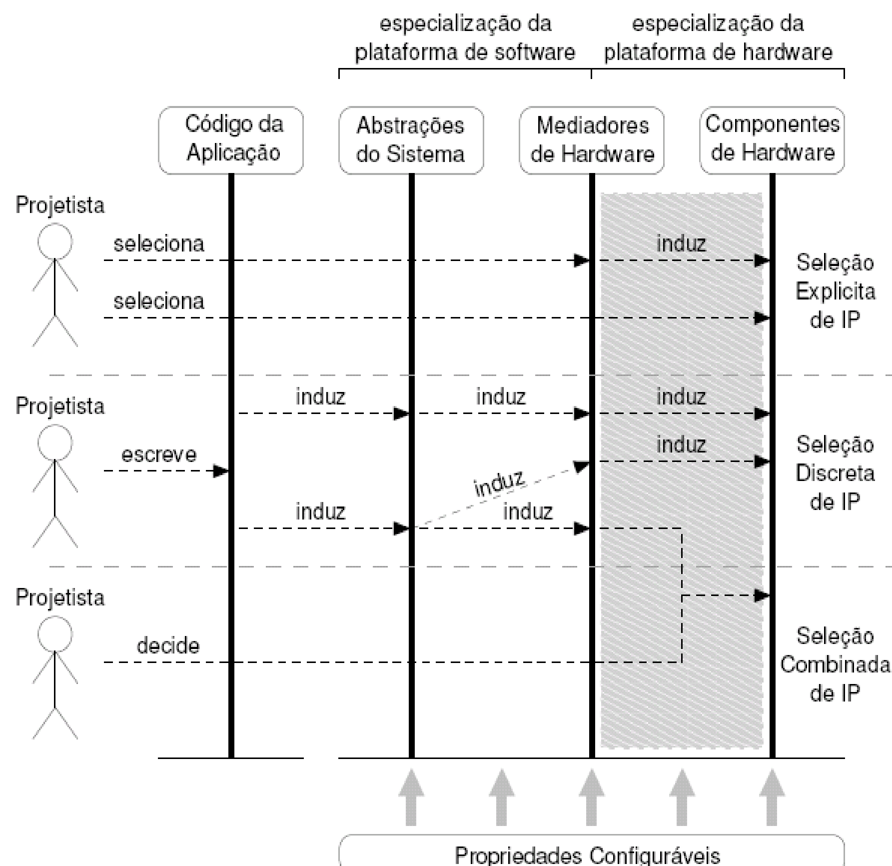

Figura 5: Especialização de plataformas segundo AOSD [PF05]

4 *Estudo de Caso: Projeto do grampeador de barramentos CAN*

4.1 Requisitos

O propósito do grampeador de barramentos CAN (*Controller Area Network*) é transformar pacotes CAN que trafegam em barramentos deste tipo em pacotes a serem transmitidos pela serial, para que possa ser feito um levantamento dos dados que trafegam em um barramento CAN. O protocolo CAN é um protocolo de tempo real, serial com *broadcast* e possui um alto nível de segurança. Os pacotes CAN possuem um identificador (veja Figura 6), que deve ser único em toda rede, esse identificador define o conteúdo do pacote e também a sua prioridade. O pacote CAN possui também um campo de 4 bits com o número de bytes de dados (DLC), que varia entre 0 e 8. Existe no pacote um bit RTR, que diz se este é um pacote de dados ou um *frame* vazio de requisição de dados. Barramentos CAN estão presentes em grande parte dos automóveis modernos, caminhões, aviões, veículos militares entre outros. A identificação de pacotes e seus conteúdos permitem o monitoramento de características de veículos que utilizem baramentos CAN na comunicação de dados, como o consumo de combustível, uso dos freios entre outras propriedades do automóvel. Mais informações sobre CAN e suas aplicações podem ser obtidas em *http://www.can.bosch.com*.

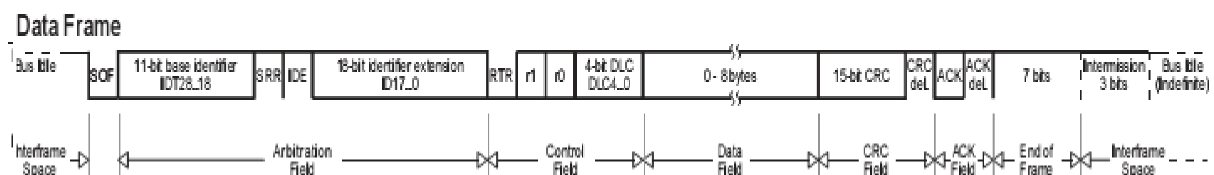

Figura 6: Pacote de dados CAN [AT905]

4.1.1 Requisitos não-funcionais

O grampeador de barramentos deve ser capaz de monitorar barramentos com *baudrates* de 250 kbps e inferiores, sem perda de pacotes. Assim, barramentos CAN onde trafegam mensagens que obedecem o protocolo J1939 podem ser grampeados. O protocolo J1939 é utilizado para conectar unidades de controle eletrônico (*electronic control units* ECU) em veículos. Essas unidades de controle podem estar nos freios, tanque de combustível e outros equipamentos do automóvel, transmitindo para uma unidade de controle central dados sobre o estado dos equipamentos. Este protocolo foi estabelecido em 1998 pelo SAE (*Society of Automotive Engineers* http://www.sae.org/products/j1939.htm) para realizar a camada de aplicação e transporte em comunicações utilizando o protocolo CAN. Barramentos onde as mensagens que trafegam seguem o protocolo J1939 operam com *baudrate* de 250Kbps. Normalmente, controladores CAN implementam a camada física e a camada de enlace de dados do modelo OSI (*Open System Interconnection*), por isso a necessidade de outro protocolo sobre o CAN. A comunicação serial, realizada entre o grampeador e o *Host* que espera os pacotes no formato serial, deve suportar velocidades de 115200 bps e inferiores.

4.2 Especificação

Na Figura 7 temos o modelo funcional do sistema, os estados em que este se encontra depois de iniciado e os eventos que fazem a mudança de estados. A inicialização não é mostrada nesta máquina de estados, pois ela visa retratar somente a operação do sistema. Após o envio do pacote CAN transformado em serial, o sistema volta a receber pacotes do barramento CAN.

A seguir, na Figura 8, vemos a estrutura de classes de nosso sistema. Duas classes representam abstrações de hardware, uma a UART e outra que representa a abstração do controlador CAN, os métodos e atributos destas classes fornecem uma interface simples de configuração e uso do controlador UART e CAN. Vemos também a classe que representa o GrampeadorCAN, classe CANListener. Esta será a classe responsável pelo funcionamento do sistema. Um objeto da classe CANListener possui dois atributos, um da classe UART e outro da classe CAN, que usa para se comunicar através de UART e de CAN. Além destas classes, foi criada também a classe CANPacket que facilita o acesso aos campos de um pacote CAN recebido.

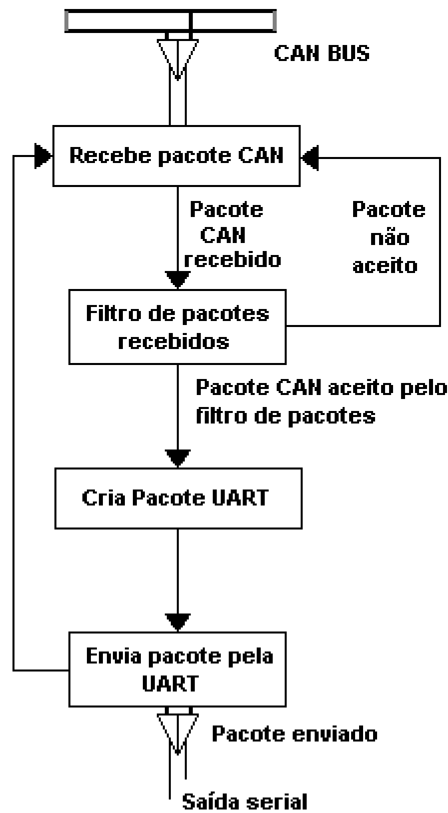

Figura 7: Modelo Funcional do grampeador CAN .

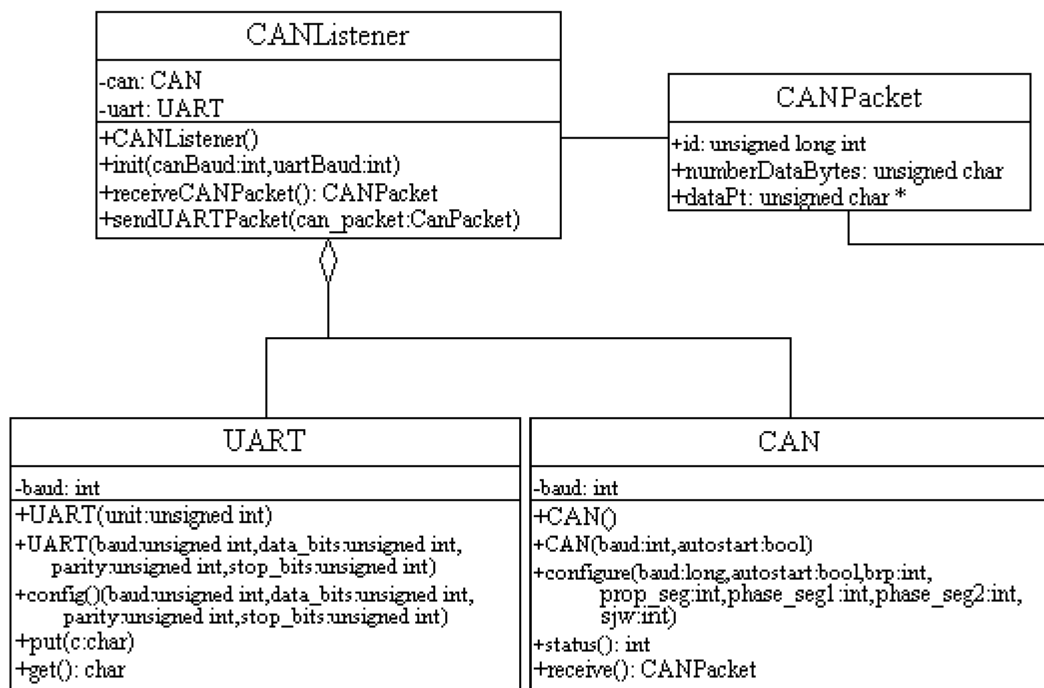

Figura 8: Diagrama das classes do projeto .

4.3 Projeto da Arquitetura

A Figura 13 mostra os blocos principais necessários no desenvolvimento do grampeador de barramentos CAN. O controlador CAN recebe os pacotes CAN que trafegam no barramento ao qual este é plugado. Esses pacotes passam por um filtro, que seleciona os pacotes a serem recebidos. Em nosso protótipo este filtro foi configurado para aceitar qualquer pacote. Os pacotes CAN aceitos são transformados em pacotes seriais (veja Figura 10) e então é executada uma rotina que envia esse pacote pela UART.

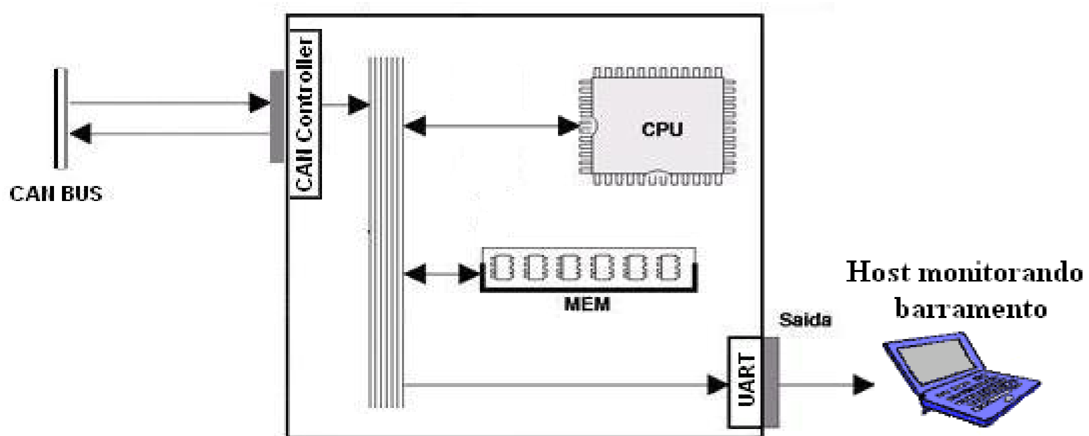

Figura 9: Diagrama de blocos do grampeador de barramentos CAN .

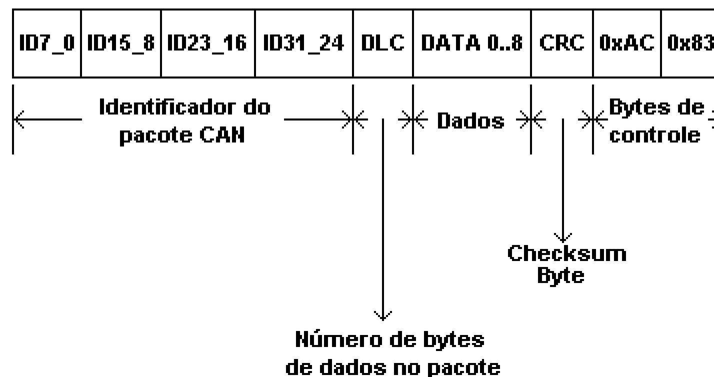

Figura 10: Pacote a ser enviado pela UART montado a partir de um pacote CAN .

O objetivo deste experimento é comparar o desenvolvimento de um sistema embarcado utilizando duas abordagens diferentes. Cada uma destas abordagens possui gera uma arquitetura diferente. Nas seções a seguir, mostramos as diferenças entre as arquiteturas utilizadas e como a estratégia de projeto influencia a arquitetura.

4.4 Grampeador CAN utilizando microcontrolador

Utilizamos em nosso projeto o microcontrolador AT90CAN128, disponível em nosso laboratório. Este microcontrolador é bastante usado no mercado e possui diversas ferramentas que auxiliam no seu uso.

4.4.1 Microcontrolador AT90CAN128

O microcontrolador AT90CAN128 faz parte da família de microcontroladores AVR da Atmel. Estes são microcontroladores RISC de 8 bits, arquitetura *Harvard* com alto desempenho e baixo consumo de energia. Este microcontrolador possui um controlador CAN integrado com filtros implementados em hardware e dispositivos de armazenamento de até 15 pacotes CAN ao mesmo tempo (*Message Objects*). O diagrama de blocos deste microcontrolador pode ser visto na Figura 11, e mais detalhes deste podem ser encontrados no seu *datasheet* [AT905].

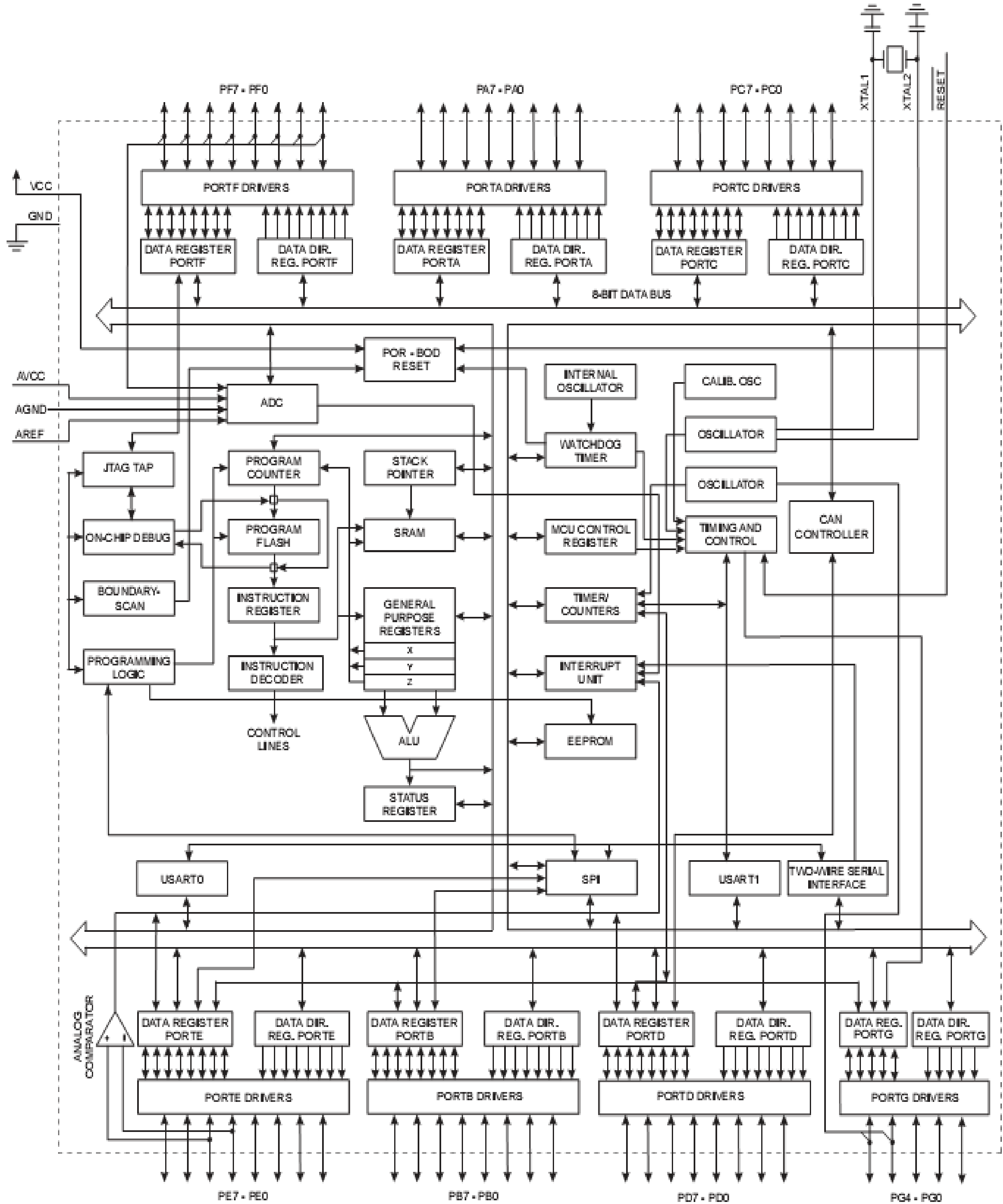

Figura 11: Diagrama de blocos do microcontrolador AT90CAN128

Esse microcontrolador possui Compiladores C (avr-gcc), depuradores (JTag), simuladores (AVRStudio) e *kits* de desenvolvimento que facilitam o desenvolvimento de projetos que o utilizem. O desenvolvimento do grampeador CAN nesta arquitetura foi bem tranquilo, pois o sistema operacional EPOS (*Embedded Parallel Operating System* [Frö01]) utilizado no suporte ao desenvolvimento da aplicação é compatível com essa arquitetura e possui porte para este microcontrolador. Além do uso das interfaces disponibilizadas pelo sistema operacional EPOS, em um trabalho de conclusão de curso anterior, Maurici [MAU] implementou o suporte a redes CAN em aplicações embarcadas para esta arquitetura. Assim, uma interface de rede compatível com CAN facilitou o desenvolvimento do grampeador.

Utilizamos para realizar os testes da implementação dois kits de desenvolvimento STK500 [STK03] produzidos também pela Atmel, esses *kits* são utilizados para desenvolvimento utilizando qualquer microcontrolador da linha AVR, porém para que pudesse ser utilizado o microcontrolador AT90CAN128 neste *kit*, um adaptador teve que ser usado [STK04]. Esse adaptador é uma extensão do *kit* STK500 para microcontroladores com um encapsulamento diferente. Esses dois *kits* foram conectados a um barramento CAN, e a um *host* que gerava os pacotes seriais em uma UART e monitorava a outra UART para certificar o recebimento dos mesmos pacotes gerados. Um AT90CAN128 funcionava como um gerador de pacotes, enviando para o barramento o que recebia pela serial, e o outro era o grampeador de barramentos CAN em si.

O código presente na Figura 12 mostra a aplicação executada no AT90CAN128 que realizava a função de grampeador de barramento. Esta aplicação cria um CANListener e manda que este execute a sua função. A chamada da função *receiveCANPacket* faz a aplicação aguardar pela chegada de um pacote CAN, que depois de recebido será enviado pela UART pela função *sendUARTPacket*. A aplicação fica em laço infinito, recebendo pacotes CAN e os enviando pela UART.

```
#include <display.h>
#include <uart.h>
#include <utility/ostream.h>
#include <cpu.h>
#include <machine.h>
#include "CANListener.h"
#include "CANPacket.h"

__USING_SYS

int main()
{
    Display d;

    CANPacket packet;

    d.puts("This is CAN Listener on EPOS-- for AT90CAN128\n");

    CANListener listener();
    listener.init(250000, 115200);  // inits listener CAN baudrate 250000
                                    // and UART baudrate 115200

    for(;;)
    {
        packet = listener.receiveCANPacket();   // waits for a CAN packet
        listener.sendUARTPacket(packet);        // sends the packet to UART
    }
}
```

Figura 12: Aplicação do Grampeador CAN desenvolvida para o EPOS

4.5 Grampeador CAN segundo AOSD, arquitetura em FPGA

A implementação do grampeador CAN foi realizada em uma FPGA modelo VirtexII-Pro XC2V30 presente na plataforma ML310 da empresa Xilinx. Esse modelo de FPGA possui dois *hardcores* do processador PowerPC 405 de 32 bits, um desses *cores* foi utilizado no projeto. Além deste processador, alguns outros IPs inferidos a partir da aplicação também foram sintetizados na FPGA.

A aplicação do grampeador de barramentos CAN necessitou de uma interface UART para enviar os pacotes criados com o conteúdo dos pacotes CAN recebidos. Esses pacotes como dito anteriormente, na especificação do projeto, tem tamanho variável, dependendo do número de bytes de dados do pacote CAN recebido. Esse IP de UART foi obtido pela seleção combinada de IPs, pois na ferramenta utilizada existiam duas UARTs disponíveis.

Um IP de controlador de barramentos CAN era necessário para permitir a comunicação do sistema com barramentos CAN. Este IP foi inferido diretamente da aplicação (seleção discreta de IP), devido ao uso do mediador de hardware relativo ao CAN, e a existência de um único IP de controlador CAN que fosse *open source* encontrado em (http://www.opencores.org). Infelizmente esse IP não era compatível com o padrão de barramentos utilizado no sistema, o CoreConnect [IBM99], explicado mais adiante. Isso resultou em um grande esforço de integração de hardware.

Um IP controlador de interrupção também foi explicitamente selecionado para compor o sistema, para permitir a comunicação dos periféricos de entrada e saída (CAN, UART) com o processador.

A aplicação do grampeador de barramentos CAN utilizou na execução a memória DDR presente na própria plataforma de desenvolvimento, e por isso foi explicitamente selecionado um IP controlador de memória DDR. Esse controlador também estava disponível na ferramenta de desenvolvimeto EDK.

Além da ferramenta EDK (*Embedded Development Kit*), o ISE (*Integrated Software Enviroment*) ambos da Xilinx foram usados na sintetização e integração do sistema. A ferramenta Modelsim da Mentor Graphics foi utilizada na simulação e testes da ponte construída para o IP controlador CAN como dito a seguir.

A integração dos componentes de hardware foi feita utilizando a lógica de barramentos CoreConnect da IBM, presente no EDK e utiizado pela arquitetura escolhida (PPC 405).

A lógica de barramentos CoreConnect é composta por três barramentos:

- Barramento Local do Processador (*Processor Local BUS* - PLB);
- Barramento de periféricos do *chip* (*On-Chip Peripheral BUS* - OPB);
- Barramento de controle de registradores de dispositivos (*Device Control Register BUS* - DCR).

Periféricos de alta performance (como o controlador de memória DDR) são conectados ao barramento de alta-largura de banda e baixa latência PLB. IPs mais lentos (como controaldor UART e controlador CAN) são conectados ao OPB, o que reduz o tráfego no PLB, resultando em uma melhora de performance para todo o sistema. O barramento de controle dos registradores dos dispositivos permite a movimentação síncrona dos dados de registradores de propósito geral dos IPs e a CPU.

Os componentes de hardware disponibilizados pela Xilinx no EDK, como a UART, o controlador de memória e de interrupção são compatíveis com estes padrões de barramento e são facilmente integrados ao sistema.

O controlador CAN era compatível com barramentos *Wishbone* [Ope02] e por isso foi necessário adaptar sua interface para o barramento OPB presente no sistema. Os desenvolvedores deste IP projetaram a lógica e a interface deste muito dependente, o que dificultou a criação de uma nova interface para o IP (OPB). Portanto, uma ponte foi desenvolvida entre o barramento OPB e a interface Wishbone do IP. Devido a diferença entre os dois barramentos (OPB e Wishbone) o desenvolvimento desta ponte foi bastante trabalhoso, uma máquina de estados foi necessária para traduzir sinais OPB para Wishbone e criar alguns sinais que não existem em barramentos OPB. Algumas simulações, utilizando Modelsim, foram realizadas para certificar o funcionamento da ponte implementada.

Para realizar a adaptação de interfaces entre barramentos, geramos um novo *core* à partir do *wizard* existente no EDK para criar e importar novos periféricos. Esse *core* serviu como base para a ponte entre o barramento OPB e a interface Wishbone do controlador CAN. Além disso foram criados dois sinais externos neste *core* para que o controlador CAN pudesse se comunicar com o mundo externo (TX e RX) e um para que o controlador pudesse gerar interrupções para o processador.

Para este projeto utilizamos o PPC 405 configurado com clock de 100MHZ, o clock do barramento PLB foi setado para 100MHZ e o clock do barramento OPB para 50MHz.

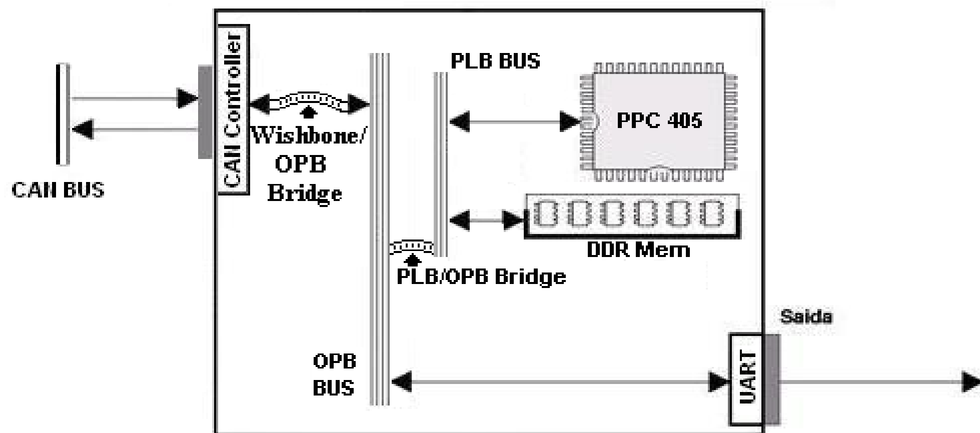

Figura 13: Diagrama de blocos do grampeador de barramentos CAN implementado em FPGA .

O clock do barramento OPB foi setado em 50MHz porque o IP controlador CAN não suportava clocks superiores a 80MHZ, por isso optou-se por 50MHz. Utilizamos também uma BRAM 96Kb, BRAMs são blocos de memória criados dentro da FPGA. Utilizamos esta memória para facilitar a carga da imagem do sistema gerado, que é feita pela aplicação presente na BRAM.

O sistema operacional EPOS também é compatível com o PPC405, logo já existia um porte pronto para esta arquitetura. Porém a interface de configuração e utilização do controlador CAN presente neste projeto é muito diferente da existente no microcontrolador AT90CAN128, este controlador é similar ao SJA1000 da Philips [Phi00], e foi projetado inicialmente para ser usado com microcontroladores 8051. Como dito acima, os desenvolvedores do IP controlador de CAN criaram neste uma interface *Wishbone* apropriada para barramentos *on-chip*.

5 Resultados e Conclusões

Vantagens e desvantagens foram encontradas no uso das duas abordagens. Na abordagem utilizando microcontroladores e componentes discretos, percebemos que o desenvolvimento utilizando microcontroladores como o AT90CAN128 da Atmel, diminui o tempo de desenvolvimento. O fabricante, Atmel nesse caso, disponibiliza ferramentas de depuração e simulação. Além disso, existem exemplos de uso dos periféricos de seus microcontroladores em código C e Assembly. Dentre as vantagens da arquitetura utilizando FPGA e o PowerPC 405 está a possibilidade de desenvolver aplicações que demandam maior poder de processamento e também a capacidade de adicionar novos periféricos e alterar o hardware dos periféricos existentes para adequá-los melhor a cada aplicação.

Ainda hoje, microcontroladores são muito utilizados, seu baixo custo e a gama de ferramentas disponíveis levam muitos projetistas adotá-los em seus projetos. Empresas buscam agregar o maior valor possível aos seus projetos, logo, quanto menor o custo de produção e desenvolvimento de um sistema maior será o lucro obtido com as vendas. O tempo curto de projeto também é fundamental para sistemas embarcados, o que torna a escolha de uma arquitetura baseada em microcontrolador uma boa opção.

A grande desvantagem no uso de microcontroladores é a impossibilidade de adicionar ou retirar alguns periféricos depois do projeto estar pronto. No mercado atual, onde novas funcionalidades para sistemas antigos surgem todos os dias, essa característica é importante. Automóveis possuem computadores de bordo com funções como cálculo de combustível disponível, consumo de combustível em uma determinada velocidade, dentre outras. Uma aplicação recente de localização, utiliza estes computadores de bordo para possibilitar ao usuário saber sua localização e prever rotas para uma determinada viagem. Esse tipo de nova funcionalidade seria extremamente difícil de adaptar a um sistema dimensionado para a tarefa anterior, pois cálculo de rota e comunicação por satélite, além de um módulo GPS, necessitaria também de um poder de processamento adicional.

Mesmo no início de um projeto, como o realizado neste trabalho, os periféricos pre-

sentes no microcontrolador não podem ser removidos. O AT90CAN128 possui *timers*, PWM (*Phase correct Pulse Width Modulator*), ADCs (*Analog Digital Conversor*), SPI (*Serial Peripheral Interface*) e alguns outros periféricos, que mesmo não sendo utilizados permanecem no *chip*. Além de espaço, esses periféricos também consomem energia, aumentando assim o consumo do sistema embarcado como um todo. Algumas arquiteturas proveem meios de desligar periféricos não utilizados, porém existem periféricos que não podem ser desligados.

Fabricantes como a própria Atmel disponibilizam hoje arquiteturas de microcontroladores de 32 bits para aplicações que demandem maior poder de processamento (AT91SAM), estes também possuem um grande número de periféricos e interfaces. Porém devido ao baixo custo proporcionado pelos avanços tecnológicos, as FPGAs possuem um custo benefício melhor que estes.

A metodologia AOSD utilizando como plataforma o PPC 405 mostrou-se mais adequada a realidade e mais flexível no caso dos periféricos. Utilizamos o PPC 405 como plataforma pois este estava disponível na ferramenta EDK, que utilizamos para integração e síntese do hardware, e pela existência do compilador(GCC) para esta arquitetura, o que facilita o desenvolvimento em C/C++. A ferramenta EDK também possui disponível o processador Microblaze, desenvolvido pela própria Xilinx, porém este não possui um porte estável do GCC. O PPC 405 possui muito mais poder de processamento que esta aplicação que desenvolvemos demanda. O uso deste processador encarece muito o produto final, porém permite a criação de novas funcionalidades para o sistema.

No caso do grampeador de barramentos CAN, entre as funcionalidades que podem ser agregadas é o cálculo do consumo de combustível em veículos, o cálculo de desgaste dos freios e outros. Cálculo de combustível, por exemplo, demandaria algum processamento, pois na especificação do protocolo J1939, protocolo estabelecido na indústria automobilística que utiliza o protocolo CAN para transporte, existem pacotes de informações como a quantidade de combustível, velocidade, distância percorrida, porém a informação de número de litros por quilômetro percorrido deveria ser calculada a partir da informação de mais de um pacote.

Com o uso de componentes de hardware sintetizáveis como o IP do controlador CAN podemos implementar também funções muito utilizadas em hardware. Por exemplo, o controlador CAN poderia ter em hardware um filtro de pacotes similar ao do AT90CAN128, que permitisse filtrar diversas mensagens, ou intervalos de identificadores diferentes.

O desenvolvimento utilizando FPGA levou mais tempo, devido principalmente a necessidade de adaptação do controlador CAN ao barramento utilizado pelo sistema (CoreConnect), que gerou esforços de implementação não existentes no desenvolvimento utilizando microcontroladores. Porém, caso a plataforma de hardware a ser utilizada, já tivesse todos os periféricos necessários compatíveis com seus barramentos ou devidamente adaptados, e o sistema operacional fosse compatível com esta plataforma, o tempo de desenvolvimento seria praticamente o mesmo para as duas abordagens.

Este trabalho possibilitou o aprendizado de conceitos fundamentais no desenvolvimento de sistemas embarcados. A comparação entre as duas metodologias proporcionou um amplo aprendizado sobre o tema. As dificuldades de adaptação do IP controlador CAN obrigaram um entendimento profundo das ferramentas de CAD e desenvolvimento baseado em plataformas já prontas.

Referências

[AT905] At90can128 datasheet. Technical report, 2005.

[BC02] Reinaldo A. Bergamaschi and John Cohn. The a to z of socs. *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design table of contents*, pages 790 – 798, 2002.

[BJ99] Rumbaugh James Booch, Grady and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Boston, 1 edition, 1999.

[Frö01] Antônio Augusto Medeiros Fröhlich. Application-oriented operating systems. Tese de doutorado, Forschungszentrum Informationstechnik, 2001.

[FSP00] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario adapters: Efficiently adapting components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, USA, 2000.

[FV99] Alberto Ferrari and Alberto Sangiovanni Vincentelli. System design: Traditional concepts and new paradigms. *Proceedings of IEEE International Conference on Computer Design*, pages 2–13, Outubro 1999.

[Hea03] Steve Heath. *Embedded System Design*. Newnes, San Francisco, 1 edition, 2003.

[IBM99] IBM. The coreconnect bus architecture. Technical report, IBM, 1999.

[Ins04] Texas Instrument. Omap texas instrument technology. Technical report, Texas Instrument, 2004.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[Kop97] H. Kopetz. *Real-Time Systems*. Kluwer Academic Publishers, Boston MA USA, 1 edition, 1997.

[LY03] Qing Li and Caroline Yao. *Real-Time Concepts for Embbeded Systems*. CMP-Books, Oxford, 2 edition, 2003.

[Mar03] Peter Marwedel. *Embedded System Design*. Kluwer Academic Publishers, Dortmund, 1 edition, 2003.

[MAU] Master's thesis.

[Ope02] Silicore Opencores. Wishbone system-on-chip (soc) interconnection architecture for portable ip cores. Technical report, Opencores, 2002.

[Par76] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[PF04] Fauze Valério Polpeta and Antônio Augusto Fröhlich. Hardware mediators: a portability artifact for component-based systems. *In: Proceedings of the International Conference on Embedded and Ubiquitous Computing*, 3207:271–280, 2004.

[PF05] Fauze Valério Polpeta and Antônio Augusto Fröhlich. On the automatic generation of soc-based embedded systems. *In: Proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation*, 2005.

[Phi00] Philips. Sja1000 datasheet. Technical report, Philips, 2000.

[Phi04] Philips. Philips nexperia platform. Technical report, Philips, 2004.

[Sim03] David E. Simon. *An Embedded Software Primer*. Addison-Wesley, Boston, 1 edition, 2003.

[STK03] Stk500 user's guide. Technical report, 2003.

[STK04] Stk501 user's guide. Technical report, 2004.

[Ten00] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43 – 50, Maio 2000.

[VC01] Alberto Sangiovanni Vincentelli and John Cohn. Platform-based design. *IEEE Design e Test*, 18(6):23 – 33, Novembro 2001.

[Wol01] Waine Wolf. *Computers as Components - Principles of Embedded Computing System Design*. Morgan Kaufmann Publishers, San Francisco, 1 edition, 2001.

[Xil04] Xilinx. Xilinx logiccore plb ipif (v2.01.a). Technical report, Xilinx, 2004.

6 Anexos

```
/* at90can128_app.cc file */

#include <display.h>
#include <uart.h>
#include <utility/ostream.h>
#include <cpu.h>
#include <machine.h>
#include "CANListener.h"
#include "CANPacket.h"

__USING_SYS

int main()
{
    Display d;

    CANPacket packet;
    CANListener listener;

    d.puts("This is CAN Listener for EPOS-- for AT90CAN128\n");


    listener.init(250000, 115200);  // inits listener CAN baudrate 250000
                         // and UART baudrate 115200
    for(;;)
    {
        packet = listener.receiveCANPacket();  //waits for a CAN packet
        listener.sendUARTPacket(packet);
    }
```

```
}
/* end at90can128_app.cc file */

/* CANListener.h file*/

#ifndef __CANListener_H
#define __CANListener_H

#include <can.h>
#include <uart.h>

#include "CANPacket.h"

__USING_SYS

class CANListener
{
        public:
        CANListener() {
        }

        void init(int can_baud, int uart_baud) {

            _can.configure(can_baud, true);
            _uart.config(uart_baud, 8, 0, 1);
        }

        CANPacket& receiveCANPacket() {

            unsigned char * buffer;
            unsigned char netID = 0;
            unsigned long RetrievedID;
            bool IsExtended;

            _can.mob_rx_mode(netID);

            while( ( _can.mob_retrieve_data(buffer, &RetrievedID, &IsExtended) )
== CAN::NO_DATA ) {}
```

```
    _packet.id(RetrievedID);
    _packet.dlc(1);
    _packet.buffer(buffer);

    return _packet;
}

void sendUARTPacket(CANPacket& packet) {

    unsigned char id[4], dlc = 0, index = 0, data[8];
    unsigned char * buffer;
    id[0] = packet.id();
    id[1] = packet.id() >> 8;
    id[2] = packet.id() >> 16;
    id[3] = packet.id() >> 24;
    buffer = packet.data();

    while (buffer && (dlc <= 8)) {
        data[dlc] = *buffer;
        dlc = dlc + 1;
        buffer++;
    }

    _uart.txd(id[3]);
    _uart.txd(id[2]);
    _uart.txd(id[1]);
    _uart.txd(id[0]);
    _uart.txd(dlc);
    while (dlc) {
        _uart.txd(data[index]);
        index++;
        dlc--;
    }
    _uart.txd(0x55);
    _uart.txd(0xAC);
    _uart.txd(0x83);
}
```

```
        private:

        CAN _can;
        UART _uart;
        CANPacket _packet;
};

#endif /* CANListener_H */
/*end CANListener.h file*/


/* CANPacket.h file*/

#ifndef __CANPacket_H
#define __CANPacket_H

__USING_SYS

class CANPacket
{
        public:

        CANPacket() {}

        CANPacket(int id, unsigned char dlc, unsigned char* data) :
            _id(id), _dlc(dlc), _data_buffer(data) {}

        unsigned long id() {return _id;}

        unsigned char dlc() {return _dlc;}

        unsigned char * data() {return _data_buffer;}

        void id(unsigned long id) {_id = id;}

        void dlc(unsigned char dlc) {_dlc = dlc;}
```

```
        void buffer(unsigned char * buffer) {_data_buffer = buffer;}

        private:

        unsigned long _id;
        unsigned char _dlc;
        unsigned char* _data_buffer;
};

#endif /* __CANPacket_H */
/*end CANPacket.h file*/
```