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

ALAN CARVALHO DE ASSIS

IMPLEMENTAÇÃO E AVALIAÇÃO DO PROTOCOLO FTT-CAN SOBRE O SISTEMA AUTOSAR

Porto Alegre
2011

ALAN CARVALHO DE ASSIS

IMPLEMENTAÇÃO E AVALIAÇÃO DO PROTOCOLO FTT-CAN SOBRE O SISTEMA AUTOSAR

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Controle e Automação

ORIENTADOR: Dr. Carlos Eduardo Pereira

Porto Alegre
2011

ALAN CARVALHO DE ASSIS

IMPLEMENTAÇÃO E AVALIAÇÃO DO PROTOCOLO FTT-CAN SOBRE O SISTEMA AUTOSAR

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Dr. Carlos Eduardo Pereira, UFRGS
Doutor pela Universidade de Stuttgart – Alemanha

Banca Examinadora:

Prof. Dr. Sergio Luis Cechin, UFRGS
Doutor pela Universidade Federal do Rio Grande do Sul, Brasil

Prof. Dr. Walter Fetter Lages, ITA
Doutor pelo Instituto Tecnológico de Aeronáutica, Brasil

Prof. Dr. Valner João Brusamarello, UFSC
Doutor pela Universidade Federal de Santa Catarina, Brasil

Coordenador do PPGEE: _____
Prof. Dr. Alexandre Sanfelice Bazanella

Porto Alegre, julho de 2011.

RESUMO

Nos últimos anos a indústria automotiva vem passando por problemas causados pela complexidade dos equipamentos eletrônicos existentes nos veículos e pela falta de padronização. Praticamente não existem componentes que sejam 100% compatíveis entre veículos de fabricantes diferentes.

Para resolver este problema foi criado o consórcio AUTOSAR, que especifica uma plataforma padrão para o software usado nos componentes eletrônicos dos veículos.

Este trabalho apresenta o padrão AUTOSAR, implementa o protocolo FTT-CAN (Flexible Time-Trigger Controler Area Network) como um barramento de comunicação seguindo a especificação AUTOSAR e propõe a utilização deste barramento como meio de comunicação entre as ECUs de um sistema automotivo. Esta implementação resultará em maior flexibilidade, segurança e determinismo temporal para a comunicação entre os componentes deste sistema. A utilização do FTT-CAN permitirá a inserção de novos módulos e novas mensagens na rede sem a necessidade de redefinição manual e *offline* da tabela de escalonamento, o que permitirá o desenvolvimento de aplicações *plug-and-play* em sistemas automotivos.

A dissertação apresenta uma nova abordagem para hot-plug de novos nós na rede FTT-CAN, o estudo de caso e análise do desempenho deste sistema implementado em relação a suas características de tempo real.

Palavras-chave: Real-time system, Real-time Scheduling, Holistic scheduling, FTT-CAN.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CAN	Protocolo ou Barramento CAN (Controller Area Network).
EC	Ciclo elementar.
ECU	Unidade de Controle Eletrônico.
EDF	Algoritmo de escalonamento Earliest Deadline First.
FTT	Protocolo Flexible Time-Trigger.
GPL	Licença de propósito geral (licença livre).
ISR	Rotina de serviço de interrupção.
MAC	Controlador de acesso ao meio.
OIL	Linguagem de implementação do OSEK.
OSEK	Sistema operacional para aplicações automotivas.
RM	Algoritmo de escalonamento Rate Monotonic.
RTOS	Sistema operacional de tempo real.
TM	Mensagem usada para controle do nós do FTT-CAN.
VFB	Barramento funcional virtual.

1 INTRODUÇÃO

Quando a indústria automotiva surgiu, no início do século passado, os fabricantes de automóveis não tinham que se preocupar com equipamentos elétricos ou eletrônicos, pois eles simplesmente não existiam num carro desta época. O primeiro modelo de carro comercial, o Ford Model T, não possuía nem mesmo faróis elétricos (COOK et al., 2007).

Aos poucos componentes elétricos e eletrônicos foram adicionados aos automóveis. Atualmente um carro de luxo contém centenas de microcontroladores (COOK et al., 2007). Portanto mais de 80% da inovação tecnológica dos carros atuais estão relacionados aos componentes eletrônicos (VINCENTELLI, 2000).

Os dispositivos eletrônicos são essenciais para controlar os movimentos de um carro e os processos elétricos que acontecem no mesmo, além de possibilitar o entretenimento do passageiro, estabelecer sua conectividade com o resto do mundo e para cuidar da sua segurança (VINCENTELLI, 2000). Esta dependência aumentará ainda mais com o desenvolvimento da tecnologia *x-by-wire* (WILWERT et al., 2005) que pretende substituir algumas partes mecânicas, hidráulicas e pneumáticas ainda existentes nos automóveis.

Estes dispositivos eletrônicos, mais conhecidos como unidade de controle eletrônico (*ECU - Electronic Control Unit*), devem comunicar entre si a fim trocar dados entre sensores e atuadores localizados em várias partes do automóvel. Inicialmente esta comunicação era realizada através de ligações ponto-a-ponto, porém no final dos anos 80 com a introdução do barramento CAN estes dispositivos passaram a ser interligados em rede. Esta mudança eliminou vários quilômetros de fiação e diminuiu a quantidade de pontos de falhas nas interligações, reduzindo o peso dos veículos (NAVET N.; SONG, 2005).

A introdução dos barramentos de rede na indústria automotiva resolveu inúmeros problemas, mas outras dificuldades surgiram. Por exemplo a complexidade dos softwares automotivos e a interoperabilidade entre ECUs de diferentes fabricantes. Estes problemas deverão ser resolvidos pela padronização proposta pelo consórcio AUTOSAR (Automotive Open System Architecture).

AUTOSAR (PAPERS, 2006) é um consórcio formado pelos principais fabricantes e fornecedores de componentes automotivos mundiais, com o objetivo de padronizar uma plataforma de software para a indústria automotiva. Hoje, componentes (software e hardware) de fabricantes diferentes não podem interoperar facilmente porque foram desenvolvidos utilizando padrões diferentes. Por isso, cada companhia precisa praticamente desenvolver toda a tecnologia que já existente, uma vez que elas não podem simplesmente reusar um componente desenvolvido por outra companhia.

Visando resolver este problema o consórcio AUTOSAR está definindo uma metodologia assim como uma camada de software para que a indústria automotiva possa padronizar seu desenvolvimento.

Porém apenas o uso do AUTOSAR não é suficiente para tornar a adição de novas

ECUs uma operação simples, rápida e transparente. Isto porque os sistemas de tempo real críticos e distribuídos, como os sistemas utilizados em automóveis, possuem características estáticas. O que significa que todo o tráfego de mensagens circulando na rede precisa ser conhecido *a priori* e que o mesmo não poderá mudar durante a operação do sistema.

Este fato leva a duas consequências imediatas: 1) este tipo de implementação gera um uso ineficiente dos recursos do sistema, tornando o custo de desenvolvimento mais alto; 2) dificuldade para realizar mudanças na configuração do sistema, uma vez que o acréscimo de mais mensagens na rede requer uma nova análise de escalonabilidade que em geral é realizada *offline* (fora do sistema alvo).

Para resolver estes problemas pode-se utilizar protocolos que conciliam flexibilidade e garantias temporais para aplicações de tempo real. Um destes protocolos é o FTT-CAN (*Flexible Time-Trigger on Controller Area Network*) (ALMEIDA; PEDREIRAS; FONSECA, 2002).

1.1 Objetivos

O objetivo deste trabalho é integrar e validar o protocolo FFT-CAN no sistema AUTOSAR e implementar o *hot-plugging* de nós nesta rede de forma que aplicações automotivas possam se beneficiar das vantagens advindas da utilização deste protocolo. Não é objetivo deste trabalho criar uma implementação comercial que a indústria possa utilizar. O objetivo é basicamente acadêmico, mas poderá servir como modelo de referência para a indústria automotiva.

1.2 Organização

Este trabalho está organizado como descrito a seguir: No Capítulo 2 são apresentados conceitos sobre sistemas distribuídos e suas características. No Capítulo 3 é apresentado sobre o sistema operacional automotivo OSEK/VDX. No Capítulo 4 alguns dos protocolos usados em redes automotivas são apresentados. Uma introdução ao padrão AUTOSAR é apresentada no Capítulo 5. Estas informações importantes para a compreensão deste trabalho.

No Capítulo 6 é apresentada a estratégia de *hot-plugging* dos nós na rede FTT-CAN, assim com o algoritmo de escalonamento. Finalmente no Capítulo 7 são apresentados os requisitos do AUTOSAR OS, a implementação do mesmo sobre o RTAI e a implementação do sistema de *hot-plugging* e a validação deste através da implementação de um sistema Steer-By-Wire.

2 SISTEMAS DISTRIBUÍDOS

Segundo (TANNENBAUM, 2002) um sistema distribuído é "uma coleção de computadores independentes que aparecem para o usuário como um único computador". O mesmo autor cita as seguintes vantagens dos sistemas distribuídos:

Economia - Microprocessadores oferecem uma melhor relação preço/performance que mainframes.[1];

Velocidade - Sistemas distribuídos podem ter mais poder computacional que mainframes;

Distribuição inerente - Algumas aplicações necessitam de soluções envolvendo maquinas em espaços físicos separados;

Confiabilidade - Se uma máquina travar ou danificar o sistema como um todo poderá continuar funcionando;

Crescimento incremental - Mais poder de processamento pode ser adicionado ao sistema com a adição de novas máquinas na rede.

Esta definição de sistemas distribuídos pode ser adequada para aplicações puramente computacionais, onde o objetivo principal é ganhar poder de processamento. Do ponto de vista de sistemas de controle esta definição é muito simplória ou até mesmo inadequada.

Portanto, uma melhor definição seria: um Sistema Distribuído é um conjunto de processos sequenciais (*p1, p2, p3 ... pn*) e uma rede com capacidade para implementar um canal de comunicação para troca de mensagens entre estes processos (BABAOGLU; MARZULLO, 1993).

Em nível de linguagem de programação a comunicação remota entre os processos poderá ser realizada através de vários paradigmas, como por exemplo: chamada remota de processos (*remote procedure call*) proposto por (BIRRELL; NELSON, 1984), broadcasts (MELLIAR-SMITH; MOSER; AGRAWALA, 1990), transações distribuídas (*distributed transactions*) (SANGORRÃN et al., 2010), objetos distribuídos (ESTEVEZ et al., 2000), memória compartilhada distribuída (ESKICIOGLU, 1996).

Os sistemas distribuídos podem transmitir mensagens de forma síncrona (*Time-Triggered*) ou de forma assíncrona (*Event-Triggered*). Mensagens síncronas são mensagens transmitida em intervalos de tempo bem definidos, em geral sincronizados com uma base de tempo local ou global. Portanto os sistemas que utilizam mensagens síncronas são determinísticos e por isso são importantes para aplicações automotivas (ALBERT; GERTH, 2003).

[1] Computador de grande porte usado para tarefas que exigem grande poder computacional.

As mensagens assíncronas são mensagens que são transmitidas a partir da ocorrência de um evento, o que justifica o nome *Event-Triggered*). O momento em que uma mensagem assíncrona vai ocorrer não pode ser previsto antecipadamente com precisão.

É importante que um sistema distribuído para aplicações de tempo real crítica, como em aplicações automotivas, seja do tipo síncrono. Assim será possível garantir a previsibilidade do sistema final.

2.1 Controle Distribuído

Em sistemas de controle complexos, como os existentes na indústria, automóveis, aviões e veículos espaciais, é utilizado uma rede de comunicação serial que permite a troca de informações e sinais de controle entre os componentes do sistema distribuídos no espaço.

Segundo (TORNGREN, 2003) os primeiros sistemas de controle distribuído via computador surgiram nos finais dos anos 1970s. Pouco tempo depois (KOPETZ, 1983) argumenta que a sincronização dos nós via um relógio global é a base fundamental para o desenvolvimento de sistemas de controle de tempo real distribuído.

Um sistema de controle distribuído em malha fechada é formado por sensores que colhem informações dos objetos a serem controlados, por um controlador que processa as informações adquiridas pelos sensores e por atuadores que recebem os dados de atuação gerados pelo controlador e atuam nos objetos sendo controlados, sendo toda a comunicação realizada através de uma rede de comunicação.

A comunicação via rede demora um certo tempo, este tempo é conhecido como atraso ou latência. Dependendo da característica da rede e da política de escalonamento das mensagens esta latência poderá ter comportamentos diferentes. Em algumas redes poderá ser praticamente constante (redes Time-Triggered) ou poderá ter variações aleatórias (redes Event-Triggered com bastante carga).

Na Figura 1 é apresentado o diagrama típico de uma rede de controle, destacando os atrasos existentes em cada ponto.

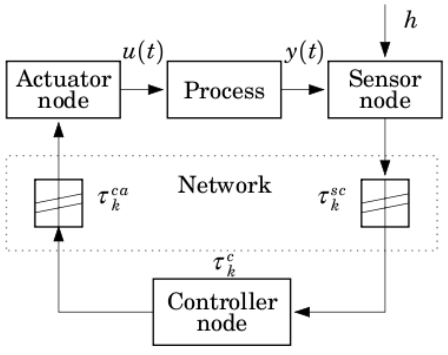


Figura 1: Rede de Controle Distribuído (NILSSON, 1998)

Onde τ_k^{sc} é o atraso na comunicação entre sensor e o controlador, τ_k^c é o atraso gasto em computação no controlador e τ_k^{ca} é o atraso na comunicação entre o controlador e o atuador. Portanto o atraso de controle (τ_k) para o sistema é igual à diferença de tempo entre o momento quando o dado foi adquirido no sensor e o momento quando o atuador foi devidamente acionado, i.e. $\tau_k = \tau_k^{sc} + \tau_k^c + \tau_k^{ca}$.

Um sério problema dos sistemas de controle distribuídos é a variação aleatória do atraso. Uma possível solução para tentar amenizar este problema é adicionar um *buffer* na entrada do controlador e do atuador, assim os dados amostrados que estão armazenados nos *buffers* são lidos em intervalos de tempo constante. Mas a capacidade de armazenamento destes *buffers* precisa ser maior que a quantidade de mensagens que seriam produzidas no pior caso de atraso, para que o atraso entre dois nós seja determinístico.

Porém a introdução de *buffers* na malha de controle poderá ocasionar na utilização de informação mais antiga que a necessária para o controle correto do sistema, o que poderá levar a uma degradação da performance (NILSSON, 1998).

Este fato demonstra a importância da rede para garantir o comportamento temporal do sistema com um todo. Uma rede com garantias temporais como o TTP/C, TTCAN e FTT-CAN são opções adequadas para aplicações em sistemas de controle distribuído de tempo real.

2.2 Modelo Event-Triggered e Time-Triggered

Existem duas estratégias para transmissão e recepção de dados em uma rede, que estão intrisicamente relacionadas com as características físicas da mesma. A primeira é o envio de dados (mensagem) no momento da ocorrência de um evento e é conhecida como Event-Triggered. Basicamente um evento pode ser entendido como uma mudança do valor de uma variável de interesse, como por exemplo mudança do valor da temperatura acima ou abaixo de um valor predefinido. A segunda é o envio de dados em espaços de tempos pré-definidos, conhecida como *Time-Triggered*, neste caso o valor da variável lida condiz com seu estado num determinado espaço de tempo. Porém não é possível inferir com exatidão quando a mesma mudou de estado, mas é possível saber que entre a leitura k e $k + 1$ o valor lido mudou, o que significa que em algum ponto no tempo entre as duas leituras um evento ocorreu.

Cada uma destas abordagens têm suas vantagens e desvantagens. Sistemas Time-Triggered normalmente são determinísticos, porém faltam flexibilidade e o desenvolvimento do projeto é mais complexo. Sistemas Event-triggered possuem melhor performance e flexibilidade, entretanto podem apresentar uma grande latência e jitter (variação do atraso).

Na literatura técnica encontra-se autores que defendem os sistemas Time-Triggered (KOPETZ, 2000) e outros que defendem os sistemas Event-Triggered (ALBERT, 2004). Porém uma solução ótima seria aquela que fizesse uso das qualidades de cada abordagem. E de fato esta é o proposta de soluções como os protocolo Flexray (FLEXRAY, 2004) e FTT-CAN (ALMEIDA; PEDREIRAS; FONSECA, 2002). No Capítulo 4.5 deste trabalho o protocolo FTT-CAN é apresentado.

Nas Seções 2.2.1 e 2.2.2 são apresentadas cada uma destas abordagens, ressaltando suas características, vantagens e desvantagens.

2.2.1 Event-Triggered

Na comunicação Event-Triggered toda vez que um evento ocorrer em qualquer um dos nós que fazem parte da rede, uma mensagem deverá ser transmitida no barramento. Esta característica torna este tipo de sistema mais flexível, pois novos nós poderão ser adicionados à rede, até mesmo com a rede em operação (*hot-plugging*), e estes transmitirão suas mensagens somente quando um evento ocorrer. Porém dois eventos podem ocorrer ao mesmo tempo, na verdade ocorrerão em espaços de tempo bem próximos, o que re-

sultará numa colisão ou atraso para envio da mensagem, dependendo da característica de acesso ao meio físico implementado pela rede em questão.

A principal vantagem dos sistemas Event-Triggered é a habilidade destes sistemas de reagirem rapidamente a eventos assíncronos externos que não são conhecidos *a priori* (ALBERT; GERTH, 2003). Esta característica confere aos sistemas event-trigger melhor desempenho para aplicações de tempo real quando comparado aos sistemas time-triggered (ALBERT, 2004). No entanto cada aplicação precisa ser bem analisada para verificar qual sistema é mais adequado.

Um típico exemplo de rede que implementa os conceitos Event-Triggered é o CAN, apresentado no Capítulo 4.2 deste trabalho.

No caso específico do CAN o processo de arbitração não destrutivo (CSMA/CA - *CSMA Collision Avoidance*) garante uma baixa latência e baixo jitter para a mensagem de maior prioridade. De fato se dois nós tentam transmitir mensagens ao mesmo tempo o processo de arbitração garantirá que a mensagem de maior prioridade será transmitida sem a ocorrência de colisão ou danos aos dados transmitidos. Porém poderá acontecer de um nó tentar transmitir uma mensagem enquanto outro nó está transmitindo uma mensagem de menor prioridade, neste caso a mensagem de maior prioridade só será transmitida ao final da transmissão da mensagem de menor prioridade (TINDELL; BURNS; WELLINGS, 1996).

2.2.2 Time-Triggered

Na comunicação Time-Triggered a permissão para acessar o barramento é controlada por janelas de tempo predefinidas (TDMA - *time division multiple access*), o que resulta num comportamento determinístico durante a operação normal do sistema. Este sistema tem a segurança gerada pelo fato que a ausência de uma mensagem no barramento será facilmente detectada. Uma deficiência desta abordagem é que todas as mensagens a serem transmitidas no barramento deverão ser conhecidas durante a fase de projeto da rede e suas janelas de tempo devidamente alocadas.

Em sistemas Time-Triggered de tempo real distribuídos é assumido que o relógio de todos os nós estão sincronizados para formarem uma noção de tempo global. Isto permite que todas as observações do objeto sendo controlado tenham uma marcação do tempo da sua ocorrência (KOPETZ, 2000). Esta marcação é útil para saber se aquele dado é relevante após determinado intervalo de tempo.

Outra característica muito importante da abordagem Time-Triggered é chamado *composability*, que é a capacidade de dois subsistemas manterem as garantias temporais quando forem integrados para formar um sistema maior. Como cada janela de tempo para acessar o barramento é predefinida e reservada para cada mensagem, dois subsistemas poderão ser desenvolvidos separadamente e depois unidos. Esta característica é importante na indústria automotiva, permitindo que uma fabricante eleja fornecedores distintos para desenvolver os subsistemas que serão posteriormente integrados no veículo.

Em aplicações automotivas os comportamentos tolerante a falha e determinístico são de grande importância, neste caso é de se esperar que soluções Time-Triggered sejam superiores comparados aos Event-Triggered (ALBERT; GERTH, 2003).

3 SISTEMA OPERACIONAL OSEK

O consórcio OSEK foi criado em maio de 1993 pelas principais fabricantes de automóveis alemãs com o objetivo de específicar um padrão aberto para as unidades de comunicação distribuídas em veículos (DIRK, 1998). A palavra OSEK origina-se da expressão alemã: "Offene Systeme und deren Schnittstellen fiir die Elektronik im Kraftfahrzeug" (português: "Sistemas Abertos e suas Interfaces Correspondentes para Eletrônicos Automotivos"). Inicialmente o projeto era formado pelos parceiros BMW, Bosch, Daimler-Benz, Opel, Siemens, Volkswagen e IIIT (Institute of Industrial Information Technology) da Universidade de Karlsruhe como coordenador. Em 1994 as fabricantes de automóveis francesas Peugeot e Renault juntaram-se ao consórcio, somando o conhecimento adquirido com o projeto VDX (*Vehicle Distributed eXecutive*) cujo objetivo era semelhante ao do OSEK. Desta união resultou o OSEK/VDX. Para simplificar a escrita, ao longo desde trabalho será utilizado apenas OSEK para se referir ao OSEK/VDX.

A arquitetura aberta do OSEK engloba três áreas:

- Comunicação (troca de dados dentro e entre ECUs);
- Gerenciamento de Rede (configuração e monitoramento);
- Sistema Operacional (sistema de tempo real para a ECU e base para os outros módulos).

O sistema operacional OSEK define dois tipos de entidades que competem pela CPU, que são as rotinas de tratamento de interrupções[1] (gerenciadas pelo SO) e as tarefas (Básicas ou Estendidas). Estas entidades são dividas em três níveis de processamento, como descrito na Tabela 1.

Tabela 1: Níveis de Processamento

Nome do Nível	**Faixa de Prioridades**
Interrupção	0-126
Lógico para o Escalonador	127
Tarefa	128-254

As entidades no nível Tarefa podem ser escalonadas como non-, full- ou mixed- preemptive determinado de acordo com as prioridades a elas associadas.

As tarefas do tipo Básica devem liberar o processador nos seguintes casos: quando terminam; quando o sistema operacional chaveia para uma tarefa de maior prioridade ou

[1] do inglês: ISR - Interrupt Service Routine

quando ocorre uma interrupção e o processador chaveia para a rotina de tratamento de interrupção (ISR).

As tarefas Estendidas distinguem-se das Básicas por terem permissão de usar a chamada do sistema *WaitEvent* que muda o estado da tarefa para *waiting*. Assim a tarefa libera o processador sem a necessidade de finalizar sua execução e quando o determinado evento, pelo qual ela aguarda, ocorrer ela poderá reassumir o uso do processador mesmo sendo menos prioritária que a tarefa em execução.

Somente a tarefa ao qual o evento pertence pode aguardar pela ocorrência deste, porém outras tarefas e rotinas de tratamento de interrupções (ISRs) poderão ativar o evento pelo qual a tarefa "dona" do evento está aguardando. Isto significa dizer que uma tarefa só pode chamar os métodos WaitEvent e ClearEvent se ela for "dona" do evento.

Para o sistema operacional as tarefas do tipo Estendidas são mais difíceis de gerenciar e requerem mais recursos do sistema (processamento e memória).

Mesmo as tarefas Estendidas não possuem características "sofisticadas", como as existentes em sistemas operacionais de propósito geral, o que permite a utilização de microcontroladores com poucos recursos (KALINSKY, 1999). As limitações incluem:

- Tarefas não podem ser criadas ou destruídas em tempo de execução;
- Parâmetros não podem ser passados para a tarefa no momento da inicialização;
- A prioridade das tarefas não pode ser alterada durante a execução;
- Preempção não pode ser ativada ou desativada durante a execução;
- Escalonador *Round-Robin* não é utilizado.

Para evitar inversão de prioridades e *deadlocks* o OSEK utiliza o protocolo "*priority ceiling*". A idéia consiste basicamente em elevar a prioridade da tarefa que está acessando um determinado recurso que precisa ser compartilhado com outras tarefas. A prioridade da tarefa que deseja acessar o recurso deverá ser elevada para um valor mais alto que o da tarefa com prioridade mais alta dentre todas as tarefas que competem pelo recurso, porém deverá ser inferior ao da tarefa com menor prioridade dentre aquelas que não competem pelo recurso.

O padrão OSEK define também as "classes de conformidade" o que permite o desenvolvimento de sistemas compatíveis e com escalonadores mais simples ou mais complexos, de acordo com a necessidade da aplicação e com os recursos do microcontrolador. Para saber qual classe de conformidade escolher deve-se levar em consideração: se a aplicação terá apenas uma ou múltiplas tarefas por prioridade; se a aplicação utilizará tarefas Básicas ou Estendidas; se a aplicação ativará uma ou múltiplas instâncias da mesma tarefa.

Seguindo estas definições é possível escolher entre quatro classes de conformidade (BCC1, BCC2, ECC1 ou ECC2) como pode ser vista na Figura 2.

Como visto anteriormente dependendo da classe de conformidade várias tarefas podem compartilhar o mesmo nível de prioridade, porém o escalonamento é independente da classe de conformidade e do tipo da tarefa. Este fato também está evidenciado na Figura 2.

O serviço de comunicação do OSEK/VDX é construído através de objeto "message", que abstrai a infra-estrutura de comunicação, ou seja, não importa se a comunicação é

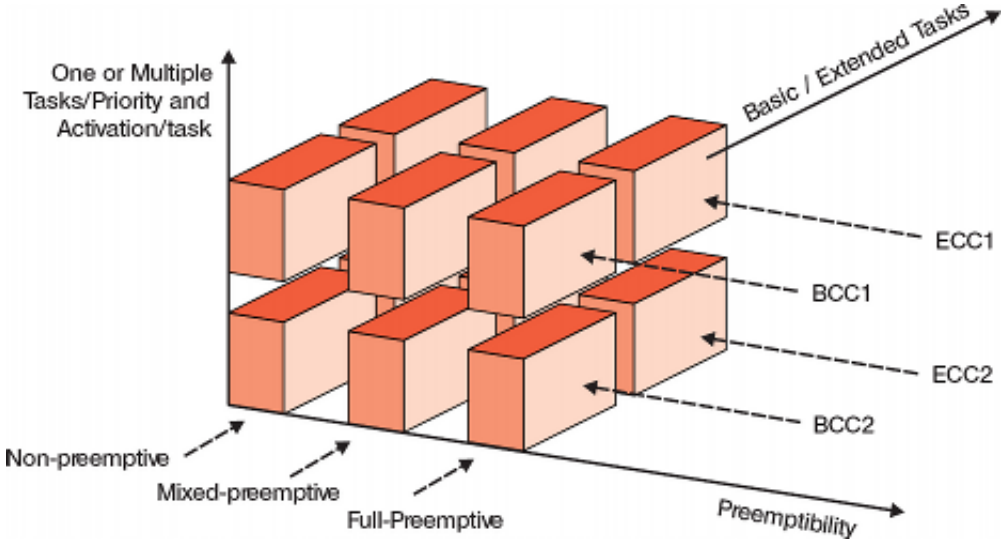


Figura 2: Escolha do Escalonador das Tarefas (KALINSKY, 1999)

local ou remota (distante, em outra ECU), para a aplicação ela fica transparente. São suportados dois tipos de mensagens: as que usam o modelo chamado quadro negro (*blackboard*), que funciona como um buffer para troca de dados; e as que usam o modelo FIFO (*First In First Out*) onde as mensagens são enfileiradas e disparadas na ordem em que chegaram.

Como o OSEK/VDX é um sistema operacional com alocação estática de memória a definição dos objetos do sistema é necessária antes da compilação (BECHENNEC et al., 2006). Como a especificação OSEK acaba definindo uma API em nível da linguagem C, juntamente com a definição dos tipos de dados relevantes, as aplicações continuam não portáveis entre implementações de diferentes fornecedores. Para resolver este problema uma nova linguagem foi criada, a OIL (*Osek Implementation Language*)(ZAHIR, 1999), que serve para definir todas as informações relevantes do objetos do S.O. que serão usadas pela aplicação. Existem softwares que irão fazer o *parsing* dos arquivos .OIL e converte-los em código fonte C que serão finalmente compilador para gerar o binário OSEK da aplicação.

Outros recursos suportados pelo OSEK:

Alarmes: Algumas aplicações precisam realizar tarefas em momentos predeterminados, para estas tarefas a utilização de Alarmes é essencial. Uma tarefa pode ser programada para ser acordada quando o relógio interno do microcontrolador atingir um tempo absoluto previamente programado ou pode ser programada para ser acordada num tempo relativo à hora atual. Além disso Alarmes podem ser programados para executar ciclicamente, com isso o desenvolvedor pode por exemplo programar uma tarefa para piscar um LED a cada 500ms;

Eventos: Eventos no OSEK/VDX são equivalentes aos semáforos do padrão `POSIX`, servem como um meio para realizar comunicação interprocessos. Os eventos possuem um papel muito importante, pois eles permitem uma tarefa entrar em modo *stand-by* e esperar por um evento específico para ser acordada. Também é possível uma tarefa enviar eventos para acordar outras tarefas ou para sincronizar uma determinada ação entre várias tarefas;

Recursos: Uma vez que o OSEK roda sobre microcontroladores é natural que o desenvolvedor queira utilizar os periféricos que este fornece. Mas poderia ocorrer problemas caso duas ou mais tarefas tentem utilizar o mesmo periférico ao mesmo tempo. Para resolver este problema o padrão OSEK conta com um gerenciador de Recursos baseado no protocolo "priority ceiling", como descrito anteriormente;

***Hooks*:** Os *hooks* funcionam de forma similar ao tratamento de exceções da linguagem de programação de alto nível. Sempre que a execução de uma tarefa gera algum problema as funções de *hooks* associadas àquele erro são chamadas para tentar minimizar o efeito do problema ou em último caso reiniciar a aplicação ou o sistema operacional.

3.1 API do OSEK

3.1.1 ActivateTask

Protótipo: `StatusType ActivateTask (TaskType TaskID)`
Descrição: Usada para mudar o estado da tarefa de `SUSPENDED` para `READY`. Se a tarefa a ser ativada possuir uma prioridade maior que a tarefa atual ela imediatamente entra em execução.

3.1.2 TerminateTask

Protótipo: `StatusType TerminateTask (void)`
Descrição: Usada para mudar o estado de uma tarefa do estado `RUNNING` para `SUSPENDED`. A tarefa não será mais escalonada pelo kernel. Para reinicia-la novamente é necessário ativar a tarefa usando a função `ActivateTask`.

3.1.3 ChainTask

Protótipo: `StatusType ChainTask (TaskType TaskID)`
Descrição: Usada para mudar o estado da tarefa atual de `RUNNING` para `SUSPENDED`. A tarefa que possuir o ID TaskID terá seu estado alterado para `READY`. Se a tarefa ativada possuir prioridade maior que a próxima tarefa em progresso, ela entrará em execução imediatamente. As principais diferenças da `TerminateTask` para a `ChainTask` é que esta só pode suspender a si mesma e não pode carregar nenhum recurso, o que significa que tarefas que usem a função `ChainTask` não podem usar Alarmes ou Periféricos.

3.1.4 Schedule

Protótipo: `StatusType Schedule(void)`
Descrição: Chama o escalonador do kernel para determinar a próxima tarefa a ser executada. O OSEK é um sistema operacional preemptivo e portanto as tarefas não precisam explicitamente chamar `Schedule`. Mas a função `Schedule` é útil para uma tarefa sair de uma seção crítica, quando nenhuma outra forma de interrupção estiver disponível. Nas seções críticas o kernel não pode interromper a tarefa e esta deverá chamar `Schedule` diretamente.

3.1.5 GetTaskID

Protótipo: `StatusType GetTaskID (TaskRefType TaskID)`
Descrição: Retorna o ID da tarefa em execução.

3.1.6 GetTaskState

Protótipo: `StatusType GetTaskState (TaskType TaskID, TaskStateRefType State)`

Descrição: Retorna o estado da tarefa requisitada (TaskID). Os possíveis valores são: `SUSPENDED`, `READY`, `RUNNING`, `WAITING`.

3.1.7 SetEvent

Protótipo: `StatusType SetEvent (TaskType TaskID, EventMaskType Mask)`

Descrição: Envia evento para uma tarefa cujo ID é igual ao TaskID. A máscara `Mask` é usada para indicar de qual evento se trata.

3.1.8 ClearEvent

Protótipo: `StatusType ClearEvent (EventMaskType Mask)`

Descrição: Usado para limpar um evento recebido pela tarefa em progresso, como definido nos bits da máscara `Mask`. Quando a tarefa recebe um evento ela deverá limpar aquele evento, caso contrário ela será acordada indefinidamente pelo mesmo evento.

3.1.9 GetEvent

Protótipo: `StatusType GetEvent (TaskType TaskID, EventMaskRefType Event)`

Descrição: Usado para pegar os eventos recebidos pela tarefa cujo identificador ID é igual a TaskID. Uma tarefa pode ser acordada por mais de um evento, então a tarefa pode usar `GetEvent` para descobrir qual evento a acordou naquele momento.

3.1.10 WaitEvent

Protótipo: `StatusType WaitEvent (EventMaskType Mask)`

Descrição: Muda o estado da tarefa atual de `RUNNING` para `WAITING`. Os bits da máscara `Mask` indicam por quais eventos a tarefa irá esperar.

3.1.11 GetAlarm

Protótipo: `StatusType GetAlarm (AlarmType AlarmID, TickRefType Tick)`

Descrição: Retorna o número de *ticks* restantes antes do alarme disparar.

3.1.12 GetAlarmBase

Protótipo: `StatusType GetAlarmBase (AlarmType AlarmID, AlarmBaseRefType Info)`

Descrição: Esta função retorna informações sobre os parâmetros interno de um alarme.

3.1.13 SetRelAlarm

Protótipo: `StatusType SetRelAlarm (AlarmType AlarmID, TickType increment,TickType cycle)`

Descrição: Programa um alarme para disparar num momento específico. O tempo é calculado somando o tempo atual com o `increment`, a partir do primeiro disparo o alarme irá disparar ciclicamente a cada `cycle` *ticks*.

3.1.14 SetAbsAlarm

Protótipo: `StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)`

Descrição: Programa um alarme para disparar num momento específico. O tempo neste caso é absoluto, calculado a partir do valor 0, que é quando o alarme foi criado. Depois do primeiro disparo o alarme irá disparar ciclicamente a cada `cycle` *ticks*.

3.1.15 CancelAlarm

Protótipo: `StatusType CancelAlarm (AlarmType AlarmID)`

Descrição: Usado para desativar um alarme. O número de Ticks não é zerado, apenas congela na posição que estava no momento que a função foi chamada. Para ser utilizado novamente o alarme deverá ser reprogramado usando as funções `SetAbsAlarm` ou `SetRealAlarm`.

3.1.16 SetAbsAlarm

Protótipo: `StatusType SetAbsAlarm (AlarmType AlarmID, TickType start, TickType cycle)`

Descrição: Programa um alarme para disparar num momento específico. O tempo nesta caso é absoluto, calculado a partir do valor 0 que é quando o Alarm foi criado. Depois do primeiro disparo o alarme irá disparar ciclicamente a cada `cycle` *ticks*.

3.1.17 GetActiveApplicationMode

Protótipo: `AppModeType GetActiveApplicationMode (void)`

Descrição: Esta função retorna o modo em que a aplicação está executando (`appMode`). Com isto é possível escrever aplicações que se comportem de forma diferente dependendo de qual modo ela está rodando, ex.: DEBUG, VALIDATION, DELIVERY, etc.

3.1.18 StartOS

Protótipo: `void StartOS (AppModeType Mode)`

Descrição: Chama a função para inicializar o kernel associando o valor `Mode` à variável global `appMode`.

3.1.19 ShutdownOS

Protótipo: `void ShutdownOS (StatusType Error)`

Descrição: Se um erro grave é detectado a tarefa pode chamar ShutdownOS passando como parâmetro o código de erro. O kernel poderá decidir qual ação tomar, mas o padrão é reiniciar o sistema e reiniciar a aplicação.

3.1.20 EnableAllInterrupts

Protótipo: `void EnableAllInterrupts (void)`

Descrição: Usado para habilitar todas as interrupções do microcontrolador. Este serviço é complementar ao `DisableAllInterrupts`.

3.1.21 DisableAllInterrupts

Protótipo: `void DisableAllInterrupts (void)`

Descrição: Usado para desabilitar todas as interrupções do microcontrolador. Este serviço é complementar ao `EnableAllInterrupts`.

3.1.22 GetResource

Protótipo: `StatusType GetResource (ResourceType ResID)`
Descrição: Usado para reservar um recurso pelo *Ceiling Protocol*.

3.1.23 ReleaseResource

Protótipo: `StatusType ReleaseResource (ResourceType ResID)`
Descrição: Usado para liberar um recurso alocado com o `GetResource`.

3.1.24 ErrorHook

Protótipo: `void ErrorHook (StatusType Error)`
Descrição: Esta rotina de *hook* é chamada pelo kernel quando um serviço retorna um código de erro diferente de `E_OK`.

3.1.25 PreTaskHook

Protótipo: `void PreTaskHook (void)`
Descrição: Esta rotina de *hook* é chamada antes do escalonador saltar para a próxima tarefa ativa.

3.1.26 PostTaskHook

Protótipo: `void PostTaskHook (void)`
Descrição: Esta rotina é chamada logo após o escalonador sair da tarefa ativa atualmente.

3.1.27 StartupHook

Protótipo: `void StartupHook (void)`
Descrição: Esta rotina de *hook* é chamada logo após a inicialização do kernel. Esta função só será executada uma única vez durante a inicialização do kernel.

subsectionShutdownHook

Protótipo: `void ShutdownHook (StatusType Error)`
Descrição: Esta rotina de *hook* é chamada pela função ShutdownOS. Assim é possível analisar e diagnosticar a origem do problema que gerou mal funcionamento no sistema.

3.2 Sistema Operacionais OSEK Open-Source

Atualmente existem vários sistemas operacionais compatíveis com o OSEK, alguns são comerciais e outros de código fonte aberto. Será dado ênfase aos SOs que forem implementados sob licença livre (GPL, BSD ou outra) para que possam servir de referência para estudos acadêmicos.

3.2.1 PICos18

O PICos18[2] é uma implementação OSEK/VDX desenvolvida pela empresa francesa Pragmatec. O código fonte deste RTOS está disponível sob licença GPL, entretanto o có-

[2]PICos18: www.picos18.com

digo fonte necessita da ferramenta comercial MPLAB da Microchip para ser compilado.

Este sistema operacional OSEK é altamente otimizado para a família PIC18 da Microchip. Isto significa que: 1) a implementação utiliza menos recursos (memória) que outras implementações; 2) o código fonte não é facilmente portável para outros microcontroladores.

O PICOS18 é compatível com as quatro classes de conformidades, ver Figura 2. Mas uma limitação do PICos18 é a quantidade de tarefas suportadas, atualmente são suportadas apenas 16 tarefas. Esta limitação se deve ao tamanho do *stack* dos microcontroladores da família PIC18.

Dentre os demais recursos suportados pelo PICos18 pode-se citar: alarmes, contadores, gerenciamento de interrupção e gerenciamento de eventos.

As específicações OSEK-COM e OSEK-NM não são suportadas por este RTOS até o presente momento.

3.2.2 TOPPERS-OSEK

O TOPPERS-OSEK[3] é um sistema operacional de código fonte aberto que implementa a especificação OSEK/VDX (CONSORTIUM, Copyright 2005) e foi desenvolvido como parte do projeto TOPPERS (*Toyohashi OPen Platform for Embedded Real-time Systems*).

O projeto TOPPERS teve início em setembro de 2003, liderado pelo professor Hiroaki Takada da universidade de Nagoia, com o objetivo de criar um sistema operacional para aplicações embarcadas e de tempo real críticas (*hard realtime*). Takada trabalha há mais de 20 anos no desenvolvimento de sistemas operacionais, aplicações embarcadas e de tempo real, sendo um dos desenvolvedores do ITRON (TAKADA; NAKAMOTO; TAMARU, 1998).

O código fonte do TOPPERS-OSEK está disponível segundo duas licenças: está disponível sob a GNU/GPL para permitir seu uso com outros software que utilizem esta licença e sob a licença TOPPERS quando o utilizador não deseja liberar o código fonte de sua aplicação. Neste caso a licença TOPPERS é menos restritiva que a GPL, fazendo apenas duas exigências: a) o Projeto TOPPERS deverá ser notificado da utilização de seu sistema operacional e b) os proprietários do Copyright e o Projeto TOPPERS serão isentos de quaisquer danos causados pelo uso do software.

O TOPPERS-OSEK foi desenvolvido com a portabilidade em mente, o código fonte é organizado de forma a manter separado a parte dependente do hardware. Atualmente vários microcontroladores são suportados, para citar alguns: m68k, h8s, arm, mips, powerpc, nios e microblaze.

As específicações OSEK-COM e OSEK-NM já são suportadas pelo TOPPERS-OSEK, porém seu código fonte só está disponível para as empresas e instituições membros do consórcio TOPPERS. Mas segundo informação disponível no site do projeto esta implementação também estará disponível para o público em breve.

3.2.3 Trampoline

O Trampoline é uma implementação OSEK/VDX desenvolvida pelo pesquisador Jean-Luc Béchennec e outros da universidade de Nantes. O código fonte deste projeto está disponível sob licença LGPL (*Lesser General Purpose License*), o que significa que aplicações comerciais podem ser desenvolvidas com código fonte fechado, porém se forem

[3] http://www.toppers.jp/en/index.html

realizadas modificações no código do sistema operacional elas devem ser liberadas para todos.

O Trampoline foi desenvolvido com alguns objetivos em mente: portabilidade para diferentes processadores/microcontroladores, limitação do consumo de memória, suporte às quatro classes de conformidade e utilização de um modo supervisor. Para simplificar a portabilidade o Trampoline utiliza uma camada de abstração de hardware (HAL), desta forma todo código acima desta camada não precisa ser modificado para funcionar em outro processador/microcontrolador.

Além disso, no código fonte do Trampoline, todas as funções que são específicas de um determinado microcontrolador são cuidadosamente isoladas, e limitadas às funções estritamente necessárias, como inicialização, troca de contexto e algumas funções específicas do hardware (ativação de interrupções, *sleeping mode*, etc).

4 PROTOCOLOS PARA COMUNICAÇÃO AUTOMOTIVA

Atualmente os carros são desenvolvidos utilizando vários módulos eletrônicos, conhecidos como ECU. Uma ECU é basicamente um subsistema composto por um microcontrolador e conjuntos de sensores e atuadores (NAVET N.; SONG, 2005).

Em geral cada ECU num automóvel realiza uma tarefa específica e precisa trocar dados com outras ECUs. Até o início dos anos 90 a troca de dados entre elas era realizada através de ligações ponto-a-ponto (NAVET N.; SONG, 2005). Isto significa que para cada ECU que necessitasse realizar troca de dados com outra ECU era necessário, ao menos, um par de fios entre elas. Então se n ECUs necessitassem comunicar entre si, seriam necessárias n^2 ligações. O resultado final era uma miríade de fios por toda a parte do veículo.

Em meados dos anos 80, a Bosch desenvolveu o barramento CAN (Controller Area Network), o que resolveu o problema das trocas de dados entre as ECUs (LEEN; HEFFERNAN, 2002). Atualmente, todas as ECUs podem usar o mesmo meio físico para se comunicar. Esta nova metodologia trouxe muitos benefícios para a indústria automotiva, como a redução do peso (resultado da eliminação de vários quilometros de fios), redução dos pontos de falhas, segurança e melhorias das funcionalidades existentes.

Entretanto a utilização apenas do barramento CAN para realizar todo o controle de um automóvel elevaria muito seu preço, devido ao preço de cada microcontrolador com um controlador CAN. Portanto barramentos mais simples e de custo menor, como o LIN, podem ser usados para controlar dispositivos simples (como vidros elétricos, faróis, travas da portas, etc). No final tem-se um automóvel com uma arquitetura mista, utilizando-se dois ou mais barramentos.

E com o surgimento das aplicações aplicações *X-By-Wire* tornou-se necessário a utilização de protocolos Time-Triggered, devido às suas garantias temporais e confiabilidade (NAVET N.; SONG, 2005). Protocolos como o TTP/C e o Flexray são utilizados neste tipo de aplicações.

Neste capítulo serão abordados vários dos protocolos acima citados, em especial LIN, CAN, TTP/C, Flexray e FTT-CAN.

4.1 O Protocolo LIN

O protocolo LIN[1] (do inglês *Local Interconnect Networks*) é um protocolo de comunicação serial projetado para controlar componentes eletrônicos mais simples do veículo (UPLAP et al., 2004), como por exemplo: vidro elétrico, retrovisor, travamento das portas, etc.

[1] http://www.lin-subbus.org

As principais vantagens do protocolo LIN segundo (DENUTO et al., 2001) são:

- Padronização;
- Baixo custo;
- Interface com único fio de 12V;
- Sincronização própria sem necessidade de cristal;
- Tempo de latência conhecido;
- Velocidade de até 20 Kbit/s.

O protocolo LIN implementa o conceito de único-mestre/múltiplos-escravos (*single-master/multiple-slave*). Apenas o nó mestre poderá transmitir o cabeçalho da mensagem e somente um nó escravo deverá responder a este cabeçalho. O cabeçalho é composto pelo sinal de *break*, seguido pelo campo de sincronização e pelo campo do identificador. O escravo responde enviando o campo de dados.

O nó escravo executa apenas a tarefa escrava (*slave task*), mas o nó mestre executa a tarefa escrava e a tarefa mestre (*master task*). A tarefa mestra é responsável por gerar a tabela de escalonamento das mensagens e por produzir o cabeçalho do frame. A tarefa escrava é responsável por responder ao cabeçalho do frame e de repassar as mensagens recebidas para a camada de aplicação.

O protocolo LIN é baseado na interface serial UART (*Universal Asynchronous Receiver/Transmitter*), portanto a codificação do frame é formada por um bit de início (*start-bit*, nível baixo), 8 bits de mensagem codificados em *non-return-to-zero* (NRZ) e um bit de parada (*stop-bit*, nível alto). O LIN não utiliza o bit de paridade, uma vez que este bit é opcional para o frame UART, portanto são utilizados 10 bits para codificar um byte de dados.

Uma incompatibilidade com o padrão UART existe na transmissão símbolo de *break*, o protocolo LIN utiliza um sinal de *break* com no mínimo 13 bits. Existem duas técnicas que tornam possíveis a um microcontrolador com uma UART padrão conseguir transmistir 13 ou mais bits para simbolizar um *break* do LIN.

A primeira consiste em: 1) diminuir 30% o *bitrate* da UART em relação ao *bitrate* usado no LIN; 2) transmitir os 10 bits do símbolo *break*, uma vez que o *bitrate* está mais lento estes 10 bits corresponderão ao tempo de bit dos 13 bits do LIN; 3) reconfigurar a UART para transmitir o restante do frame no bitrate padrão do LIN. A desvantagem dessa técnica é a sobrecarga de processamento no microcontrolador, pois para cada mensagem transmitida é necessário repetir este processo.

A segunda técnica consiste em manter o bit que configura a transmissão do símbolo de *break* ativado por mais tempo que o necessário, isto acarretará a transmissão de duas sequências de 10 bits, ou seja, 20 bits de break. A desvantagem dessa técnica é que será desperdiçada mais largura de banda transmitindo uma quantidade maior de bits break.

Este sinal de break juntamente com o campo de sincronismo são usados para que os nós escravos do barramento LIN possam sincronizar seus relógios. Isto permite que sejam utilizados microcontroladores de baixo custo que utilizam apenas o oscilador interno (RC - resistor/capacitor). Somente o modo mestre precisará de um relógio mais preciso, utilizando cristal de quartzo ou ressonador cerâmico, para realizar a sincronização dos nós escravos.

O fato do protocolo LIN ser simples, permitir a utilização de microcontroladores de baixo custo e utilizar apenas um fio para transmissão de dado torna-o muito atrativo para a indústria automotiva. Porém o protocolo vem sofrendo modificações para simplificar a integração de novos componentes nos veículos de forma rápida e transparente.

De fato a especificação LIN teve um aumento drástico de complexidade na segunda maior revisão (Revisão 2.0) (RUFF, 2003). Nesta nova versão foram incorporadas características que tornaram o LIN mais flexível, porém a implementação do sistema tornou-se mais complexa. Foram adicionados recursos como: detecção automática de bit-rate, diagnóstico, suporte a mensagens esporádicas, entre outros.

Um nó mestre LIN 2.0 é compatível com escravos LIN 1.3, porém o oposto não é verdadeiro, ou seja, não é possível a um mestre LIN 1.3 comunicar-se com escravos LIN 2.0. Neste trabalho será apresentado as características do LIN baseando-se na última especificação disponível, revisão 2.1.

4.1.1 Formato do frame LIN

O frame padrão do protocolo LIN é apresentado na Figura 3.

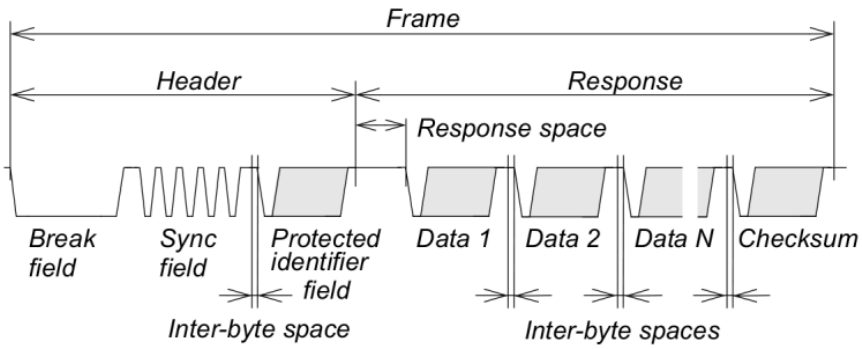


Figura 3: Frame de Protocolo LIN

O frame do LIN é formado pelo subframe de Cabeçalho e pelo subframe de Resposta, sendo o primeiro construído pelo nó mestre e o segundo pelo nó escravo.

O Cabeçalho do LIN é composto pelos campos *Break*, *Sync* e *Identificador Protegido*. O campo Break é identificado por 13 bits dominantes (nível lógico 0) e um bit delimitador recessivo (nível lógico 1). O campo Sync é formado por um byte 0x55 (além do *start-bit* e *stop-bit*). O campo Identificador Protegido é formado por dois subcampos: Identificador do Frame e Paridade.

O Identificador do Frame é formado por 6 bits, portanto valores de 0 a 63 podem ser usados. Já o subcampo Paridade é formado por 2 bits (P0 e P1).

Os identificadores do frame são divididos em três categorias:

- Valores de 0 a 59 (0x3B) são usados para frames carregando sinais;
- 60 (0x3C) e 61 (0x3D) são usados para carregar dados de diagnóstico e configuração;
- 62 (0x3E) e 63 (0x3F) são reservados para uso em futuras aplicações do protocolo.

As paridades são calculadas a partir dos bits do Identificador do Frame, segundo as equações:

$$P0 = ID0 \oplus ID1 \oplus ID2 \oplus ID4 \quad (1)$$

$$P1 = \neg(ID1 \oplus ID3 \oplus ID4 \oplus ID5) \quad (2)$$

O subframe de Resposta é composto de 1 a 8 bytes de dados seguido pelo campo Checksum. Para entidades representadas por mais de um byte, o byte menos significativo é enviado primeiro (*little-endian*). Os campos de dados são nomeados Data1, Data2, ... até Data8.

O campo Checksum é gerado pelo resto da divisão, da soma de todos os valores, por 256. O checksum é calculado sobre os bytes de dados e sobre o campo Identificador Protegido no caso do LIN 2.x ou apenas sobre os bytes de dados para o LIN 1.x. No primeiro caso o checksum é chamado de checksum melhorado e no segundo de checksum clássico.

4.1.2 Tipos de Frames

Alguns tipos de frames são apenas para propósitos específicos, como será visto nas subseções abaixo. Nem todos os tipos de frames serão suportados por determinados nós, isto depende da função deste nó.

4.1.2.1 Frame incondicional

Os frames incondicionais carregam sinais e seus identificadores de frame podem assumir os valores entre 0 e 59 (0x3B). Neste tipo de frame é utilizado o modelo publicador/assinante (*publisher/subscriber*). O publicador de um frame incondicional (tarefa escrava) sempre deverá responder ao cabeçalho produzido pela tarefa mestre.

Todos os assinantes de um frame incondicional deverão receber a mensagem publicada e repassá-la para a aplicação. As formas como estas comunicação podem ocorrer entre os nós escravos e o nó mestre são apresentadas na Figura 4.

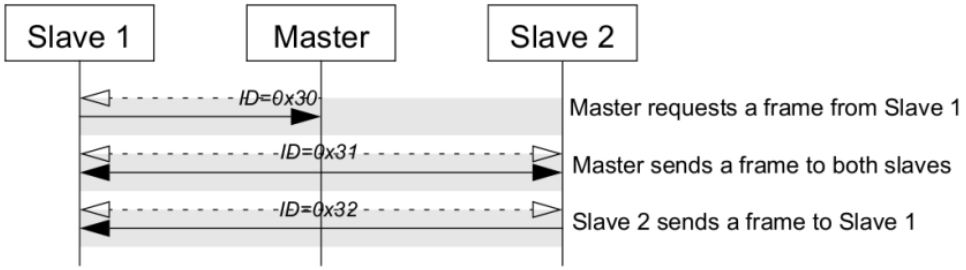


Figura 4: Três transferências de dados via frame incondicional

4.1.2.2 Frame Event Triggered

O frame Event-Triggered foi adicionado ao LIN para permitir que nós escravos respondam apenas quando os mesmos possuírem dados disponíveis. Um frame Event-Triggered é associado aos sinais definidos para um ou mais Frame Incondicional, por exemplo, um frame Event-Triggered com ID=0x12 pode estar associado ao Frame Incondicional ID=0x13 do nó escravo 1 e ao Frame Incondicional ID=0x14 do nó escravo 2.

Neste exemplo citado, sempre que o mestre enviar o frame Event-Triggered com ID=0x12 os nós 1 ou 2 poderão responder caso algum deles tenha um dado atualizado

para enviar. Se o nó 1 não possuir dado para transmitir, mas o nó 2 possuir, ele irá criar um frame de Resposta com o primeiro byte de dados contendo seu ID (nó 2: ID=0x14) e os demais bytes contendo o dado do sinal disponível.

Se acontecer de dois ou mais nós responderem ao mesmo tempo a um frame Event-Triggered uma colisão será detectada pelo nó mestre. Neste caso o nó mestre consultará uma tabela de resolução de colisão para saber qual nó é mais prioritário. No exemplo citado, suponhamos que o nó 1 seja o mais prioritário, então o nó mestre enviará um frame Incondicional contendo o ID do nó 1 (0x13) para que o mesmo possa transmitir seu dado, em seguida enviará outro frame com o ID do nó 2.

O frame Event-Triggered evita que o nó mestre fique consultando cada nó individualmente para saber se eles têm dados atualizados para enviar. Isto melhora o tempo de resposta do sistema.

4.1.2.3 Frame Esporádico

O objetivo do frame esporádico é adicionar um comportamento dinâmico no escalonamento do protocolo LIN, mas sem perder o determinismo do sistema como um todo.

Um frame Esporádico é associado a um grupo de frames Incondicionais e é transmitido pelo nó mestre, num slot destinado ao frame Esporádico, quando houver uma atualização de um sinal de um frame Incondicional do grupo. Caso sinais de diferentes frame Incondicionais sejam atualizados ao mesmo tempo, o nó mestre transmitirá o mais prioritário no slot de frame Esporádico atual e os demais serão transmitidos nos próximos frames Esporádicos ou quando seus frames Incondicionais forem escalonados.

Normalmente os slots dos frames Esporádicos ficam vazios, uma vez que frames Esporádicos só são enviados quando ocorre uma atualização de um sinal fora da fase em que normalmente este é transmitido no frame Incondicional.

Apenas o nó mestre pode enviar frames Esporádicos, uma vez que apenas a tarefa mestre sabe quando um determinado frame Incondicional está pendente para transmissão.

4.1.2.4 Frame de Diagnóstico

Os frames de diagnósticos possuem o Identificador de Frame igual a 60 (0x3C), chamados de frame de requisição do mestre, ou 61 (0x3D), chamados de frame de requisição do escravo. Um frame de diagnóstico sempre contém 8 bytes de dados.

Antes de transmitir um frame de requisição o nó mestre consulta o módulo de diagnóstico para saber se a mensagem de diagnóstico deverá ser transmitida ou se o barramento deverá ficar em silêncio. O nó escravo deverá enviar um frame de resposta incondicionalmente.

O frame de Diagnóstico utiliza o checksum clássico.

4.1.2.5 Frame Reservados

Frames Reservados são identificados pelo Identificador de Frame igual a 62 (0x3E) e 63 (0x3F). Frames Reservados não devem ser usados numa rede LIN 2.x.

4.2 O Protocolo CAN

O protocolo CAN foi desenvolvido pela empresa alemã Bosch no fim dos anos 80 para uso em sistemas automotivos. O protocolo foi padronizado como ISO 11898-2 para aplicações de alta velocidade (até 1Mbps) e como ISO 11519-2 para aplicações de baixa

velocidade (até 125Kbps) (CALHA, 2006).

O CAN (*Controller Area Network*) é um protocolo de comunicação serial adequado para aplicações em rede de sensores, atuadores e outros nós de sistema de tempo real distribuídos (JOHANSSON; TORNGREN; NIELSEN, 2005).

O protocolo CAN é dividido em duas camadas do modelo OSI: camada física e enlace de dados.

O conteúdo das mensagens no barramento CAN é descrito pelo campo identificador (*Identifier*), este campo não é relacionado ao endereço de origem ou destino da mensagem, mas refere-se ao significado dos dados. Desta forma os nós na rede que desejarem receber mensagens com determinado tipo de dados ativarão seus filtros para deixar passar apenas as mensagens de seu interesse.

Cada mensagem poderá conter de 0 a 8 bytes. O campo identificador é único para cada mensagem e é representado por um número de 11 bits (CAN 2.0A), porém só é possível obter 2032 identificadores, uma vez que no CAN é proibido identificadores com os sete bits mais significativos iguais a '1' (TINDELL; BURNS; WELLINGS, 1996).

O identificador também define a "prioridade" da mensagem, como descrito na Seção 4.2.2. O barramento CAN implementa diferentes tipos de *frames*, utilizados para funções específicas. Estes frames são apresentados a seguir na Seção 4.2.1.

A característica mais importante do CAN é seu mecanismo de tratamento de colisões. Se várias estações estão tentando transmitir ao mesmo tempo e uma delas escrever '0' no barramento, todas as demais que monitorarem o barramento irão ver '0' na rede. Da mesma forma somente se todas as estações estiverem transmitindo '1' que elas poderão ver '1' no barramento. Isto significa que o bit '0' é dominante na rede e o bit '1' recessivo. Este fato é essencial para o processo de arbitragem do barramento, como descrito na Seção 4.2.2.

4.2.1 Tipos de Frames

Conforme (FREESCALE, 2001), a transferência de mensagens no barramento CAN são realizadas e controladas através de quatro tipos de frames:

- **frame de dados** - carrega dados de um transmissor para um receptor;
- **frame remoto** - é enviado por um nó da rede sempre que ele desejar que outro nó transmita uma mensagem com o mesmo identificador;
- **frame de erro** - é transmitido por qualquer nó que tenha detectado algum erro no barramento;
- **frame de sobrecarga** - é usado para gerar um atraso extra entre os quadros de dados e/ou remotos.

4.2.1.1 Frame de Dados

O frame de dados é formado por sete campos diferentes: "início do frame" (SOF - *Start of Frame*), "campo de arbitragem" (*Arbitration field*), "campo de controle" (*Control field*), "campo de dados" (*Data field*), "campo CRC" (*Cyclic redundancy check*), "campo ACK" (*Acknowledge*) e "fim do frame" (EOF - *End of Frame*).

A Figura 5 apresenta os campos do frame de dados.

O campo "início do frame", como o próprio nome diz, serve para indicar o início de um frame. É formado apenas por um bit dominante.

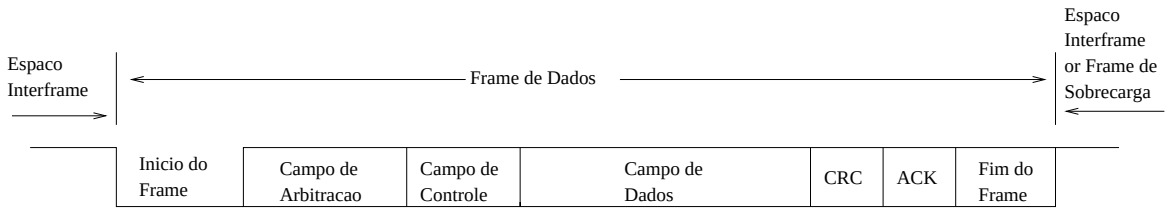


Figura 5: Frame de Dados

O “campo de arbitragem” é formado por 12 bits (11 bits do identificador mais um bit do RTR[2]) no caso do CAN 2.0A ou 32 bits (29 bits do identificador mais os bits SRR[3], IDE[4], RTR) no caso do CAN 2.0B.

O “campo de controle” define a quantidade de dados (bytes) a serem transmitidos e, no caso do CAN 2.0A, possui o bit IDE, usado para indicar qual formato do frame (padrão ou estendido / 2.0A ou 2.0B). Este campo é formado por 6 bits, tanto para a especificação CAN 2.0A quanto para a 2.0B, embora neste o bit IDE foi transferido para o “campo de arbitragem” e em seu lugar foi definido o bit reservado BR1.

O “campo de dados” contém os dados que serão transmitidos. Este campo pode conter de 0 a 8 bytes, que são transferidos do bit mais significativo até o menos significativo (MSB - *Most Significant Bit*). A quantidade de bytes transmitidos é definida através dos bits DLC3-DLC0 existentes no “campo de controle”, conforme Tabela 2.

Tabela 2: Codificação do Tamanho dos Dados. (FREESCALE, 2001)

DLC3	**DLC2**	**DLC1**	**DLC0**	**BYTES**
d	d	d	d	0
d	d	d	r	1
d	d	r	d	2
d	d	r	r	3
d	r	d	d	4
d	r	d	r	5
d	r	r	d	6
d	r	r	r	7
r	d	d	d	8
d = “dominante” r = “recessivo”				

O “campo CRC” é utilizado para verificação de erros na transmissão. Este campo é formado por 15 bits da sequência do CRC e um bit delimitador do CRC, bit recessivo. É importante notar que o CRC é gerado a partir dos bits dos campos “início do frame”, “arbitragem”,“controle” e “dados”. O CRC é calculado com os bits originais destes campos, ou seja, antes do controlador CAN adicionar os “bit-stuffing”.

O “campo ACK” é utilizado para que o nó receptor informe ao nó transmissor que recebeu uma mensagem corretamente. O transmissor envia dois bits recessivos e o receptor indica que conseguiu receber a mensagem mudando o primeiro bit para dominante e mantendo o segundo em recessivo.

[2]Remote Transmition Request bit
[3]Substitute Remote Request bit
[4]Identifier Extention bit

Finalmente o campo “fim do frame” indica que o final do frame transmitido. Este campo é formado por 7 (sete) bits recessivos.

4.2.1.2 Frame Remoto

Numa rede CAN os nós receptores podem indicar aos nós transmissores que desejam receber dados. Assim os nós transmissores podem atuar de forma reativa, ou seja, não precisam ficar enviando dados a todo momento de forma pró-ativa. O nó receptor faz isto enviando um identificador igual ao identificador do frame de dados desejado.

Um frame remoto é composto por seis campos: “início do frame” (SOF - *Start of Frame*), “campo de arbitragem” (*Arbitration field*), “campo de controle” (*Control field*), “campo CRC” (*Cyclic redundancy check*), “campo ACK” (*Acknowledge*) e “fim do frame” (EOF - *End of Frame*).

Como pode-se perceber o frame remoto é semelhante ao frame de dados, porém não possui o “campo de dados”. A descrição de cada frame é a mesma apresentada na Seção 4.2.1.1. Visto que o frame remoto não possui o “campo de dados”, o valor codificado nos bits do tamanho de dados, existente no “campo de controle”, é ignorado.

O bit RTR indica, através de sua polaridade, se o frame transmitido é um frame de dados (bit RTR dominante) ou um frame remoto (bit RTR recessivo).

A Figura 6 apresenta os campos do frame remoto.

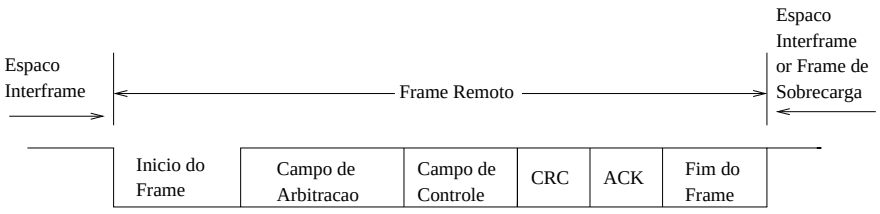


Figura 6: Frame Remoto

4.2.1.3 Frame de Erro

O Frame de Erro é utilizado para indicar a ocorrência de um estado de erro. Este frame é composto por dois campos: o campo “Flag de Erro” (*Error Flag*), que é formado pela sobreposição de todas as flags de erros dos nós no barrameto, e pelo campo “Delimitador de Error” usado para indicar a finalização de um Frame de Erro.

Existem dois tipos de “Flag de Erro”: a flag de Erro PASSIVA e a flag de Erro ATIVA. A flag de Erro PASSIVA consiste de uma sequência de 6 (seis) bits recessivos e a flag de Erro ATIVA consiste de uma sequência de 6 (seis) bits dominantes.

Um nó no estado “erro-ativo” que detectar a ocorrência de um estado de erro deverá indicar isto transmitindo uma flag de Erro ATIVA. Como esta flag é uma sequência de 6 (seis) bits iguais, ela viola a regra do *bit-stuffing*, ver Seção 4.2.3, com isto outros nós na rede detectarão este erro no barramento e também enviarão suas flags de Erro. A sobreposição de todos estes bits dominantes formará uma sequência de mínima de 6 (seis) e máxima de 12 (doze) bits.

Da mesma forma um nó em estado “erro-passivo” quando detectar um estado de erro deverá transmitir uma flag Erro PASSIVO.

A Figura 7 apresenta os campos do frame de erro.

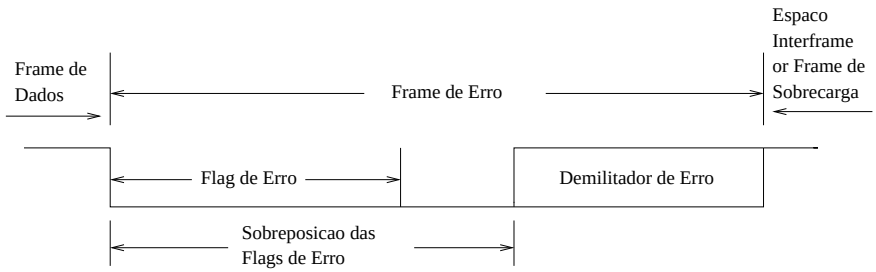


Figura 7: Frame de Erro

4.2.1.4 Frame de Sobrecarga

O frame de sobrecarga é gerado em duas situações: 1) quando por alguma condição interna o nó não é capaz de receber a próxima mensagem ou 2) se um dos 2 (dois) primeiros bits do espaço interframe é dominante. No primeiro caso o frame de sobrecarga funciona como um estado de espera no barramento.

Todos os nós somente poderão transmitir consecutivamente dois frames de sobrecarga.

A Figura 8 apresenta os campos do frame de sobrecarga.

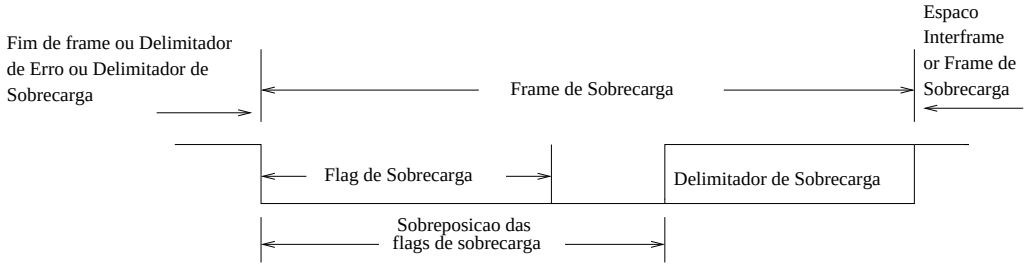


Figura 8: Frame de Sobrecarga

4.2.1.5 Espaçamento Interframe

Frames de dados e frames remotos são precedidos por um conjunto de bits chamados de “espaço interframe”. Frames de erro e frames de sobrecarga não são separados por “espaço interface”. O “espaço de interframe” é formado por 3 (três) bits recessivos. Sendo que os dois primeiros bits chamados de intervalo (*intermission*) e o último bit funcionando como delimitador de interframe.

4.2.2 Arbitragem

Toda vez que o barramento estiver livre qualquer nó poderá transmitir sua mensagem. Se dois ou mais nós começam a transmitir ao mesmo tempo, a arbitração é feita através de comparação “bit a bit” do campo identificador. A comparação inicia-se do bit mais significativo em direção ao menos significativo. O nó vencedor será o nó que tiver um bit dominante mais à esquerda (bits mais significativos). Se dois ou mais nós começarem com bits dominantes nas mesmas posições, a comparação passa para o próximo bit menos significativo, e assim até encontrar uma posição onde um tem bit dominante e o outro recessivo.

Durante a fase de arbitragem todos os nós que estão transmitindo comparam o nível do sinal transmitido com o nível do sinal no barramento, para saber se o valor que estão transmitindo é o valor presente no barramento. Caso um nó esteja transmitindo um bit

recessivo, mas se, ao analisar o barramento, verificar que o barramento está transmitindo um bit dominante, imediatamente este nó abandona sua transmissão.

Se um frame de dados e um frame remoto, ambos com o mesmo identificador, tentam transmitir ao mesmo tempo, o frame de dados prevalecerá sobre o frame remoto. Este mecanismo de arbitragem garante que não será perdido nem dados e nem tempo (FREESCALE, 2001).

A Figura 9 apresenta o processo de arbitragem realizado quando três nós tentam acessar ao mesmo tempo o barramento. Todos os nós começam com os bits iguais, porém o sexto bit mais significativo do Nó1 é recessivo (nível lógico "1") e no momento da transmissão deste bit ele perde o barramento para o Nó2 e Nó3, que possuem bits dominantes nesta posição. Da mesma forma, o décimo bit mais significativo do Nó2 é recessivo e no momento de sua transmissão este nó perde o barramento para o Nó3 que possui o décimo bit dominante.

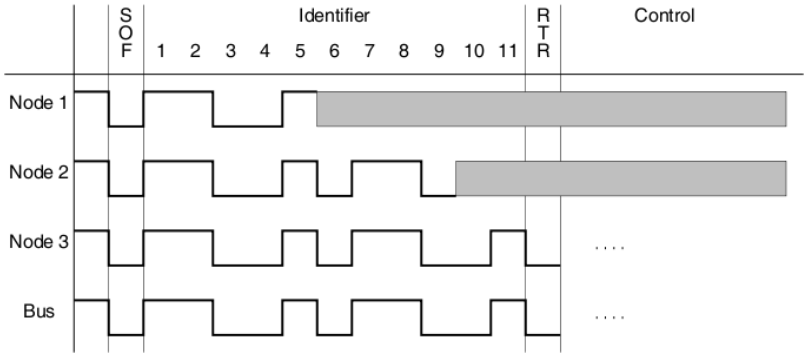


Figura 9: Arbitragem no CAN (JOHANSSON; TORNGREN; NIELSEN, 2005)

Este exemplo mostra que o barramento CAN implementa uma forma de arbitragem elegante, o que evita o problema de colisões (i.e. CSMA-CD) ou o problema de atrasos para envio de mensagens, como ocorre em barramentos do tipo passagem de bastão (*token passing*) devido ao tempo de circulação do *token* na rede.

4.2.3 Bit-stuffing

O protocolo CAN define que uma sequência de 5 (cinco) bits de mesma polaridade no frame deverá ser acrescida de um bit de polaridade oposta. Isto significa que se numa sequência existirem 6 (seis) bits de mesma polaridade (000000 ou 111111) no barramento é considerado em erro.

Como o *bit-stuffing* é utilizado, a quantidade de bits transmitidos no barramento não é a simples soma do tamanho dos bits transmitidos, mas depende conteúdo dos próprios bits transmitidos.

Além de servir como uma forma de detecção de erros o *bit-stuffing* também é utilizado para assegurar a sincronização entre todos os nós do barramento (CiA, 2004).

4.2.4 Interface física

A especificação CAN desenvolvida pela BOSCH (BOSCH, 1991) não cobre a camada física, porém isto é definido na especificação ISO-11898.

No CAN são definidos dois estados lógicos: dominante e recessivo. Na especificação ISO-11898 estes dois estados lógicos (bits) são representados por diferencial de tensão. A Figura 10 apresenta o diferencial de tensão utilizado para representar cada estado.

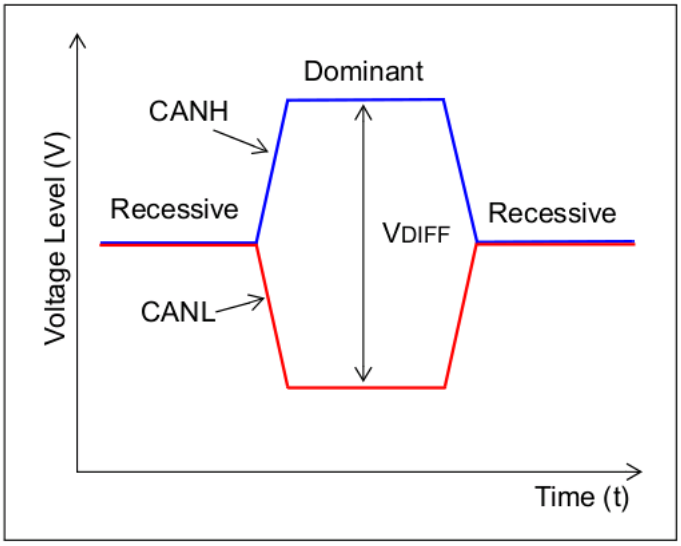


Figura 10: Barramento diferencial. (RICHARDS, 2002)

O barramento CAN é implementado por um par de fios, um fio é ligado ao pino CANL (*CAN Low*) e o outro ao pino CANH (*CAN High*) do *transceiver* utilizado. No estado recessivo a diferença de tensão entre estes dois sinais deverá ser inferior a 0,5V e no estado dominante deverá ser superior 0,9V. A tensão nominal é ∼2,5V para os sinais CANL e CANH quando em estado recessivo. Quando em estado dominante a tensão nominal é de ∼1,5V e ∼3,5V para CANL e CANH respectivamente (RICHARDS, 2002).

A ISO-11898-2 não define quais tipos de fios e conectores devem ser utilizados para interligar os nós no barramento CAN. Entretanto a especificação requer que os fios e os conectores utilizados sejam adequados para manter os níveis de tensão dentro dos limites permitidos.

A especificação também determina que sejam usados resistores terminadores de 120Ω em cada ponta do barramento, usados para casamento de impedância, evitando que o sinal seja refletidos de volta nos extremos do barramento.

4.3 O Protocolo TTP

O protocolo *Time-Triggered Protocol* (TTP) é um protocolo de comunicação utilizado para conectar módulos eletrônicos em sistemas de tempo real distribuídos tolerantes à falhas. Este protocolo foi desenvolvido pela Universidade Técnica de Viena desde 1993.

A implementação mais conhecida do protocolo TTP é o TTP/C, originalmente desenvolvido para atender os requisitos das aplicações automotivas SAE classe C. A especificação atual (1.4.3) tem como alvo sistemas de tempo real distribuídos com fortes requerimentos de segurança, disponibilidade e *composability*[5], que são necessários no campo automotivo, eletrônicos para aplicações espaciais e sistemas de controle e automação industrial.

O TTP/C é implementado segundo a arquitetura *Time-Triggered* (TTA - *Time-Triggered Architecture*). Esta arquitetura define a infra-estrutura computacional para implementação das aplicações, além de prover mecanismos e guias de como particionar uma aplicação

[5]Integração de dois ou mais subsistemas formando um sistema maior que mantem as características temporais destes subsistemas.

grande em subsistemas autônomos menores e bem definidos. A TTA decompõe um sistema embarcado grande em *clusters* e nós provendo uma base de tempo global tolerante a falhas cuja precisão é conhecida por todos os nós.

A unidade base da arquitetura TTA é um nó, que é composto de um processador com memória, sistema de entrada/saída, controlador de comunicação *time-triggered*, um sistema operacional e uma aplicação em software, tudo isso contido numa única unidade ou até mesmo numa única pastilha de silício. Os nós são interligados através de canais de comunicação replicados formando os *clusters*.

A comunição na arquitetura TTA é autônoma e realizada em divisões de tempo (*slots*) geradas através da arbitração TDMA, e as mensagens são previamente escalonadas e enviadas periodicamente. O TTP/C se encarrega da consistência das mensagens, isso significa que quando a aplicação recebe alguma mensagem ela pode confiar na integridade destes dados.

Cada nó TTP/C contém seus dados de controle do escalonador que armazena uma lista de descritor de mensagens personalizada (MEDL - *Message Descriptor List*) e especifica em que instante aquele nó deverá transmitir ou receber uma mensagem. A transmissão da mensagem não é ponto-a-ponto, mas sim por *broadcast*, ou seja, todos os nós recebem todos os dados transmitidos no barramento.

Como o acesso ao barramento é realizado por cada nó apenas no *slot* reservado para este, o acesso ao barramento é assegurado livre de colisões. Isto é reforçado pela existência da base de tempo global conhecida por todos os nós, pelo escalonamento de acesso ao barramento estático definido previamente por todos os participantes e pela existência do guardião do barramento que impede que determinado nó envie mensagens em outro *slot* senão aquele reservado para ele.

A base de tempo global é criada através da sincronização entre nós com relógios precisos, chamados de relógios mestres (*master clocks*). O fato de não utilizar um servidor central como base de tempo global contribui para o carácter tolerante a falhas do TTP/C.

Em relação à topologia de rede, o TTP/C pode ser implemento como rede em barramento ou em estrela, ou uma combinação destes dois tipos.

4.3.1 Tipos de Frames do TTP/C

O protocolo TTP/C define três tipos de frames (TTA-GROUP, 2003):

- **frame de início a frio (*cold start frame*)** - usado unicamente para inicialização da rede;

- **frame de dados C-State explícito** - o C-State é um frame de indica o estado interno do controlador de comunicação, indicando: tempo atual, frame sendo transmitido, modo de funcionamento atual do frame, vetor de associados (lista nós que estão em funcionamento), estas informações são necessárias ao nós que desejam se integrar ao cluster durante a inicialização do mesmo ou que desejam ser reintegrados ao cluster;

- **frame de dados C-State implícito** - Neste caso o C-State não é transmitido, mas o nó receptor pode detectar se seu C-State está válido para a mensagem transmitida, uma vez que o CRC será calculado com base nos dados do frame mais o C-State.

Um frame TTP/C é composto pelos campos: tipo do frame, requisição para mudança de modo, dados da aplicação, CRC e, dependendo do tipo do frame, o C-State. O frame

de dados pode carregar até 240 bytes (TTA-GROUP, 2004), mas até o momento que este trabalho foi escrito, a especificação da camada de compatibilidade que define o formato exato do frame não está disponível publicamente na última versão da especificação (TTA-GROUP, 2003).

4.3.2 Esquema de Acesso ao Meio

O protocolo TTP/C utiliza o esquema TDMA (*time-division multiple acces*) como estratégia de acesso ao médio físico. Em protocolos TDMA cada nó pode acessar o meio físico para transmissão de sua mensagem durante uma quantidade de tempo fixa, chamada de *slot* TDMA.

Um *slot* lógico do nó deve-se iniciar antes do *slot* TDMA, pois o nó precisa de uma pequena quantidade de tempo para preparar uma ação para o próximo *slot* (ler a tabela MEDL, inicializar o transceiver, etc). Esta pequena fase do *slot* lógico do nó, que antecede o *slot* TDMA, é chamada de *pre-send phase* (PSP), como pode ser visto na Figura 11.

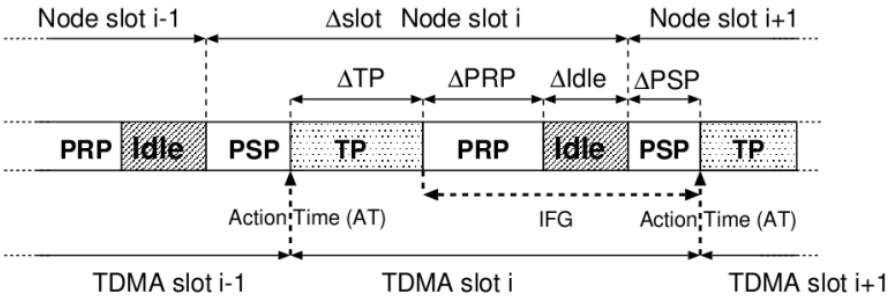


Figura 11: Temporização do slot. (TTA-GROUP, 2003)

Um *slot* TDMA inicia-se com a fase de transmissão (*TP - transmission phase*) e termina com o início da transmissão do nó sucessor. Como pode ser visto na Figura 11 o tamanho do *slot* TDMA e do *slot* do nó são iguais, a diferença é apenas o ponto onde cada um inicia. O início da fase de transmissão é chamada de tempo de ação (*AT - action time*) e o tempo após a fase de transmissão é necessário para que o nó processe os dados recebidos é chamado de *post receive phase* (PRP).

A seqüência periódica de *slots* TDMAs (ou *slots* lógicos do nó) é chamada de *round* TDMA. O tamanho dos *rounds* TDMAs são iguais, apenas o conteúdo e o tamanho dos frames enviados nos *slots* dos nós podem variar. A recorrência do padrão periódico dos *rounds* TDMAs é chamada de "ciclo do cluster". A Figura 12 apresenta o diagrama exemplificando este conjunto.

É permitido que dois ou mais nós compartilhem um determinado *slot* para melhorar a utilização da largura de banda da rede. Estes nós que compartilham o mesmo *slot* são chamados de nós multiplexados. Porém cada nó multiplexado será associado a um round TDMA, evitando assim a ocorrência de colisões.

4.4 O Protocolo Flexray

O Flexray[6] é um protocolo de comunicação Time-Triggered, desenvolvido pelo Consórcio Flexray, objetivando primariamente aplicações automotivas. Este protocolo permite a formação de uma rede com até 64 nós. A topologia da rede pode ser do tipo

[6]Consórcio Flexray: http://www.flexray.com

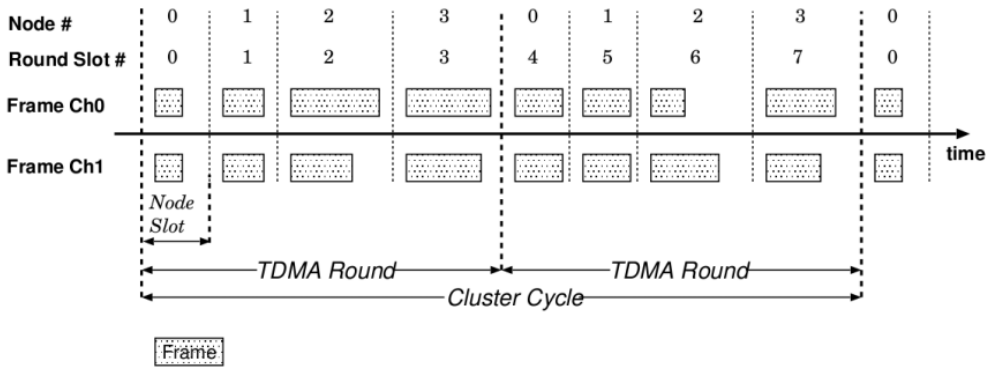


Figura 12: Ciclo do cluster. (TTA-GROUP, 2003)

barramento passivo, estrela ativa ou passiva, ou uma combinação destes tipos, chamada de rede híbrida.

O objetivo do consórcio Flexray era desenvolver um protocolo confiável, flexível e tolerante a falhas.

O taxa de transmissão de dados do Flexray é de 10Mbps e o meio físico utilizado pode ser fios de cobre ou fibra ótica. A especificação prevê a existência de dois canais independentes (canal A e B), porém pode-se criar redes Flexray utilizando-se apenas um canal. Um nó pode ser ligado a ambos ou a apenas um canal.

4.4.1 Ciclo de Comunicação

Um ciclo de comunicação do Flexray é formado por quatro segmentos: segmento estático (*static segment*), segmento dinâmico (*dynamic segment*), janela de símbolo (*symbol window*) e tempo ocioso da rede(*network idle time*). Na Figura 13 é apresentado o diagrama do ciclo de comunicação do Flexray.

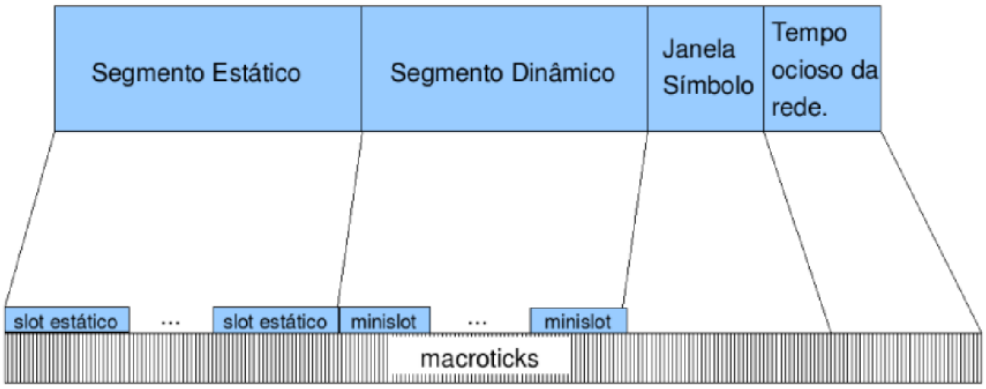


Figura 13: Ciclo de Comunicação do Flexray

O segmento estático é composto por vários intervalos de tempo (slots) de tamanhos iguais, utilizando um esquema de arbitração TDMA, que são usado para transmissão de frames escalonados estaticamente. Múltiplos slots podem ser alocados para o mesmo nó no mesmo ciclo de comunicação. Os espaços alocados para as mensagens escalonadas no segmento estático são sempre reservados para elas, portanto um slot reservado para uma mensagem estática (Time-Triggered) não poderá ser utilizado por uma mensagem

dinâmica (Event-Triggered) ou por outra estática, mesmo se naquele ciclo o slot estiver desocupado.

O segmento dinâmico é formado por *minislots* onde frames escalonados dinamicamente são enviados (NOLTE; HANSSON; BELLO, 2004). Para o sistema de *minislotting* funcionar cada mensagem precisa ter um identificador único. Cada nó possui um contador de slots, no início do segmento dinâmico este contador é zerado e a cada mensagem transmitida este contador é incrementado e um novo *minislot* dinâmico é detectado por todos os nós. O nó que possuir uma mensagem com identificador igual ao valor do contador de slots irá transmití-la no início deste *minislot*.

Se após um tempo Δ (com Δ menor que o tempo para transmitir uma mensagem) nenhum nó transmitir uma mensagem, então um novo minislot dinâmico é detectado e o contador de slot é incrementado. As mensagens de maior prioridade possuem identificador mais baixo e por isso são transmitidas primeiro. O tamanho de uma mensagem é variável (até 254 bytes), mas o tamanho do segmento dinâmico é fixo, portanto dependendo da quantidade de mensagens mais prioritárias e de seus tamanhos, uma mensagem menos prioritária terá que aguardar o próximo segmento dinâmico para ser transmitida.

A janela de símbolo é o período no qual um símbolo pode ser transmitido no barramento. A janela de símbolo pode ser usada para testes, em tempo de execução, da interconexão da camada física da rede com a ECU (BERWANGER; SCHEDL; TEMPLE, 2007).

A unidade de tempo base do Flexray é chamada de *macrotick* e é formada por um número variável de microticks. Os pulsos ("ticks") dos osciladores locais podem deslocar-se devido às tolerâncias de fabricação, envelhecimento do cristal e diferenças ambientais. Para que a duração do macrotick seja a mesma em todos os nós, apesar deste fatores citados, o número de microticks necessários para formar um macrotick precisa ser reajustado de tempo em tempo. Isto acontece durante o segmento *network idle time*, quando os nós calculam a sincronização de relógio e ajustam seus relógios para manterem-se em cadência uns com os outros (BOHM, 2005).

O protocolo Flexray também prevê a utilização do guardião do barramento (*bus guardian*), que protege o barramento contra comportamentos defeituosos, como por exemplo quando um nó começa a transmitir fora do seu slot pré-alocado. Se um nó começar a ter este mal comportamento, imediatamente o guardião do barramento irá bloqueá-lo evitando que ele atrapalhe as demais comunições no barramento.

4.4.2 Frame do Flexray

O frame do Flexray é formado por três segmentos: o segmento de cabeçalho (*header segment*), o segmento de carga útil (*payload segment*) e o segmento de "reboque" (*trailer segment*).

Na Figura 14 é apresentado o diagrama do frame do Flexray.

O segmento de cabeçalho ocupa cinco bytes e engloba os seguintes campo: Bit Reservado, Indicador de Preâmbulo da Carga Útil, Indicador de Frame Nulo, Indicador de Frame de Sincronismo, Indicador de Frame de Inicialização, Identificador do Frame, Tamanho da Carga Útil, CRC do Cabeçalho, Contador de Ciclos.

O segmento de carga útil poderá carregar de 0 a 254 bytes de dados (0 a 127 palavras de 2 bytes), como o campo Tamanho da Carga Útil contém apenas 7 bits, a quantidade de bytes enviados como carga útil é sempre o dobro do valor deste campo.

O segmento de "reboque" contém 24 bits e possui o CRC computado sobre o segmento de cabeçalho e sobre o segmento de carga útil.

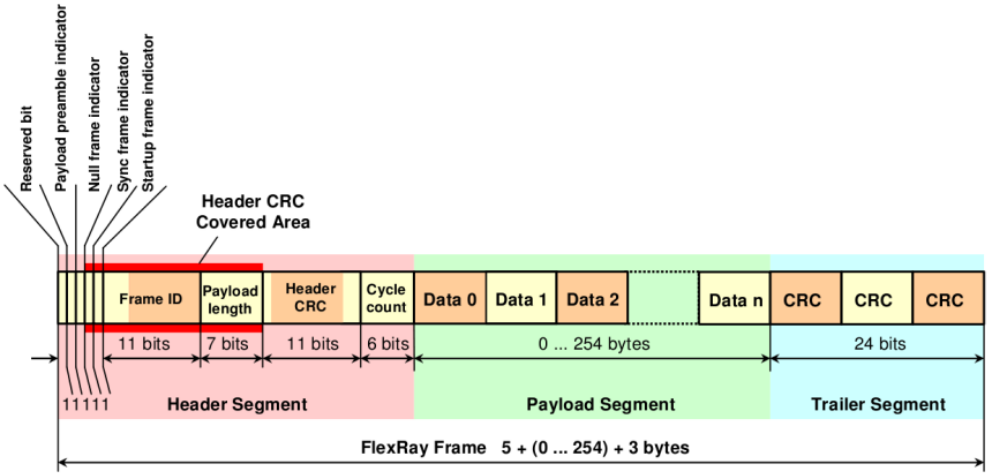


Figura 14: Frame do Flexray

O Identificador do Frame possui 11 bits, o que significa que um ciclo de comunicação poderia ser configurado para conter no máximo 2048 frames.

4.5 O protocolo FTT-CAN

O protocolo FTT-CAN (*Flexible Time Triggered on Controller Area Network*) foi criado com o objetivo de tornar possível a admissão dinâmica de mensagens numa rede em barramento, preservando as garantias temporais desta rede.

Segundo (ALMEIDA; FONSECA; FONSECA, 1998) somente sistemas de tempo real flexíveis possuem capacidade de operar em ambientes dinâmicos, onde uma completa especificação não é possível e onde os requisitos podem mudar durante o ciclo de vida do sistema.

Este protocolo utiliza como base o protocolo CAN, modificando a forma como o CAN faz seu controle de acesso ao meio. Na verdade esta modificação é mais temporal que física, ou seja, ele ordena as mensagens de forma a criar uma separação entre mensagens *event-triggered* (ET) e *time-triggered* (TT).

Para isso o FTT-CAN divide o tempo de acesso ao barramento em ciclos periódicos, chamados de Ciclos Elementares (*Elementary Cycles* - ECs), como pode ser visto na Figura 15.

Mensagens do tipo *time-triggered* (TT) são adequadas para transportar dados periódicos, enquanto que as *event-triggered* (ET) são adequadas para carregar dados esporádicos (ALMEIDA; PEDREIRAS; FONSECA, 2002). Infelizmente a maioria das redes de campo privilegia um determinado tipo de mensagem em detrimento a outra.

Exemplos de aplicações das mensagens *time-triggered* são: controle de movimento, controle de motor, controle de temperatura e controle de posicionamento. Como exemplo de aplicações para mensagens *event-triggered* pode-se citar: monitoramento de alarmes, diagnósticos e configurações (i.e. ajuste de *set-point*).

O protocolo FTT-CAN define duas janelas de tempo consecutivas, assíncrona e síncrona, que correspondem às duas fases deste protocolo. A fase assíncrona é usada para transportar mensagens do tipo *event-triggered*, sendo chamada de assíncrona porque as mensagens nesta fase podem ocorrer a qualquer instante em relação ao início do EC. Já a fase síncrona transporta mensagens do tipo *time-triggered* e possui este nome porque as

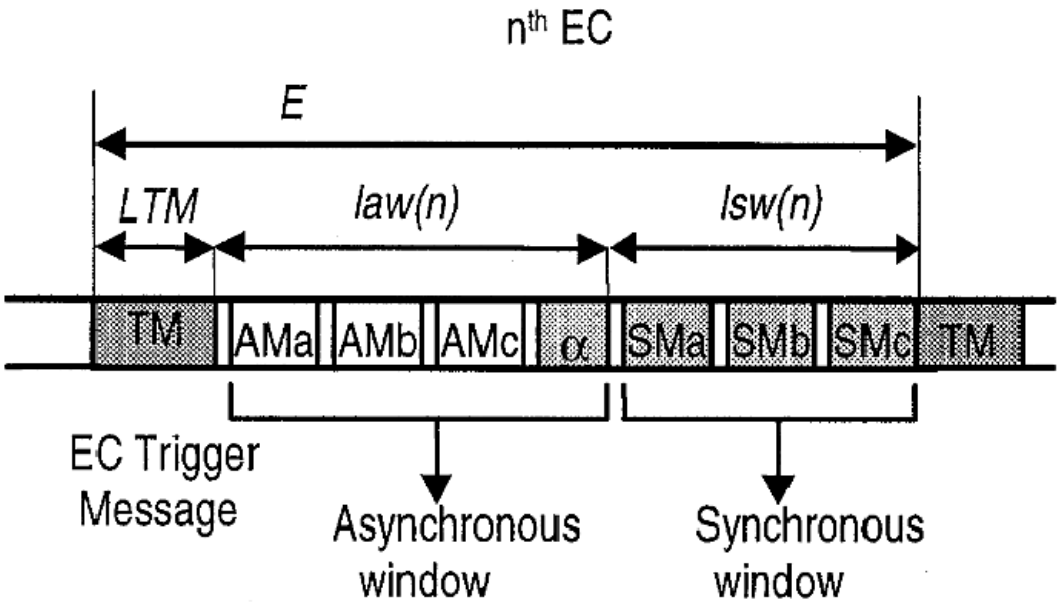


Figura 15: Elementary Ciclo do FTT-CAN (ALMEIDA; PEDREIRAS; FONSECA, 2002)

mensagens ocorrem em tempos sincronizados com o início do EC.

A duração de cada fase (ET e TT) deve ser ajustada de acordo com a aplicação em particular. No entanto a duração de cada Ciclo Elementar é fixa (*E* unidades de tempo). Cada Ciclo Elementar é composto pela *trigger message*(TM), uma fase síncrona e uma fase assíncrona. A TM apenas precisa carregar a indicação das mensagens síncronas que deverão ser produzidas durante a fase síncrona. Todos os nós decodificarão a TM, mas apenas aqueles que devem produzir as mensagens, indicadas na TM, começarão o envio na fase síncrona.

Os nós que possuem mensagens assíncronas na fila de envio ativam sua transmissão no início da janela assíncrona e o próprio MAC do CAN automaticamente resolve quaisquer disputa pelo barramento, usando sua política de arbitragem, ver Seção 4.2.2. Para saber quando começa cada fase (cada janela) os nós no barramento são sincronizados tomando como base o momento de início de cada EC, caracterizado pela recepção da *trigger message* (TM). A transmissão da TM gasta uma quantidade de tempo constante (*LTM*).

A duração da janela assíncrona da enésima (*n*) EC é definida por *law(n)* e a duração da janela síncrona correspondente é definida por *lsw(n)*. Como o tamanho do EC é fixo, pode-se facilmente deduzir que o tamanho destas duas janelas são inversamente proporcionais.

A razão pela qual a janela assíncrona precede a janela síncrona está relacionado ao tempo que os nós gastam para decodificar a mensagem TM. É a mensagem TM que define quais mensagens *time-triggered* deverão ser transmitidas no EC atual. Dependendo do microcontrolador utilizado (i.e. microcontrolares de 8 bits), a decodificação da TM poderá demorar um longo tempo. Colocando-se a janela de mensagens assíncronas antes das síncronas permite que a decodificação da TM seja realizada em paralelo com a transmissão de mensagens assíncronas, resultando assim no uso mais eficiente do barramento. Usando microcontroladores com mais poder de processamento (i.e. microcontroladores de 32 bits) este tempo de decodificação é insignificante, neste caso a janela síncrona poderá preceder a janela assíncrona.

Um isolamento temporal é utilizado para evitar que a transmissão de mensagens assíncronas sobreponha o início da janela síncrona. Para assegurar este isolamento as mensagens pendentes, que não possam ser transmitidas dentro da janela assíncrona, são removidas do *buffer* de transmissão do controlador de rede. Por causa disso pode ocorrer o aparecimento de um pequeno espaço ocioso no barramento, no final da janela assíncrona (α na Figura 15). No final da janela síncrona também poderá ocorrer um pequeno espaço ocioso, mas neste caso gerado pela variação do *bit-stuffing* usado na codificação das mensagens CAN.

O tráfego de mensagens *time-triggered* é escalonado *online* (com o barramento em funcionamento) e fica centralizado em um único nó, chamado de nó mestre. O protocolo FTT-CAN objetiva aplicações de tempo real baseadas em microcontroladores com baixo poder de processamento, tipicamente encontrados em sistemas embarcados distribuídos (PEDREIRAS, 2003). Portanto para reduzir a sobrecarga do protocolo foi utilizado uma técnica de escalonamento chamado "escalonador de planejamento" (*planning scheduler*) (ALMEIDA; PASADAS; FONSECA, 2001), que basicamente consiste em calcular a escalação das mensagens para os próximos "*n*" ECs.

Os serviços de comunicação do FTT-CAN são fornecidos para a aplicação por meio de dois subsistemas, o Sistema de Mensagem Síncronas (SMS) e o Sistema de Mensagens Assíncronas (AMS). Os serviços oferecidos pelo SMS são baseados no modelo "produtor-consumidor" e os do AMS são oferecidos através do paradigma *send-receive*.

O restante deste capítulo é organizado da seguinte forma: na Seção 4.5.1 é apresentado os tipos de mensagens definidas no protocolo FTT-CAN; a Seção 4.5.2 aborda o sistema SMS e a Seção 4.5.3 cobre o sistema AMS.

4.5.1 Tipos de mensagens do FTT-CAN

O protocolo FTT-CAN define os seguintes tipos de mensagens:

- **EC Trigger Message** [TM_MESG_ID];
- **Mensagens de Dados Síncronos** [DATA_MESG_ID];
- **Mensagens de Dados Assíncronos** [AM_DATA_MESG_ID];
- **Mensagens de Controle** [CONTROL_MESG_ID].

Os quatro bits mais significativos do campo de identificação do CAN são utilizados para definir os tipos de mensagens do FTT-CAN, como pode ser visto na Tabela 3.

Tabela 3: Identificação do tipo de mensagem. (PEDREIRAS, 2003)

0 [Sync]	0 [Master]	00	TM_MESG_ID
	1 [Slave]	10	DATA_MESG_ID
1 Async	000	CONTROL_MESG_ID (HP)	
	100	AM_DATA_MESG_ID (RT)	
	110	CONTROL_MESG_ID (LP)	
	111	AM_DATA_MESG_ID (NRT)	

Como pode ser visto na Tabela 3 há dois tipos de mensagem CONTROL_MESG_ID e AM_DATA_MESG, a razão da existência delas é que mensagens assíncronas podem ter diferentes criticidades temporais, portanto diferentes classes de tráfego.

Na Tabela 4 é apresentado os campos do frame da TM.

Tabela 4: Campos da TM

Tipo	**ID Mestre**	**Novo Plano**	**No. Seq.**	**Tam. Janela Síncrona**	**EC-Schedule**
Campo ID do CAN				Campo de Dados do CAN	
[b10..b7]	[b6..b4]	b3	[b2..b0]]	MSB	1 a 7 bytes
TM_MESG_ID	0 a 7	0,1	0 a 7	0 a 255	bitmap

A mensagem TM é identificada pelo campo **Tipo** contendo o valor TM_MESG_ID (valor "0000"). O campo **ID Mestre** permite a existência de até 8 mestres diferentes na rede, o que permite a implementação de sistemas redundantes. O **Novo Plano** é utilizado para indicar o início de um novo plano quando o Escalonador de Planejamento é utilizado. O campo **Número de Sequência** é incrementado pelo mestre a cada EC e permite detectar a omissão de até oito TM consecutivas. O **Tamanho da Janela Síncrona** informa a duração da janela síncrona, este tamanho tem uma resolução de $LSW/255$. Finalmente o campo **EC-Schedule** indica quais mensagens assíncronas deverão ser produzidas no EC. Este campo é codificado em forma de *bitmap*, ou seja, cada bit 1 significa que a mensagem correspondente deverá ser produzida. A posição dos bits correspondem exatamente a mensagem a ser produzida, assim bit0 = SM0, bit1 = SM1 e assim sucessivamente.

As Mensagens de Dados Síncronos são usadas para envio de dados periódicos, sempre que uma mensagem estiver mapeada no campo **EC-Schedule** ela será transmitida por algum dos nós na rede. A Tabela 5 apresenta os campos do frame desta mensagem.

Tabela 5: Campos da Mensagem de Dados Síncronos

Tipo	**TX_ND**	**ID da Mensagem**	**Dado da Mensagem**
Campo ID do CAN			Campo de Dados do CAN
[b10..b7]	b6	[b5..b0]	0 a 8 bytes
DATA_MESG_ID	0,1	0 a 63	Dep. Aplicação

O campo **Tipo** contém o valor DATA_MESG_ID indicando que esta é uma Mensagem de Dados Síncronos. O campo "transmite novo dado" (**TX_ND**) é usado como uma *flag* para indicar à aplicação se o dado sendo transmitido neste EC é igual ou diferente ao dado transmitido no EC anterior. Se ela estiver ativada significa que o dado é diferente e que a aplicação que consome este dado deverá atualizar sua cópia local, se estiver desativado significa que o dado é igual ao transmitido anteriormente e a aplicação poderá utilizar o dado antigo, eliminando a operação de leitura. O campo **ID da Mensagem** serve para identificar cada tipo de mensagem, no caso do FTT-CAN existem 64 tipos diferentes de mensagens. Por último, o campo **Dado da Mensagem** carrega até 8 bytes de dados úteis.

As Mensagens de Dados Assíncronos são utilizadas para envio de dados esporádicos, em geral, informações de eventos. Este tipo de mensagem é enviada diretamente pela aplicação durante a ocorrência de um evento sem interferência do nó mestre, o nó que

desejar enviar esta mensagem apenas precisa aguardar o início da janela assíncrona para fazê-lo. A Tabela 6 apresenta os campos do frame deste tipo de mensagem.

Tabela 6: Campos da Mensagem de Dados Assíncronos

Tipo	**Não Usa**	**ID da Mensagem**	**Dado da Mensagem**
Campo ID do CAN			Campo de Dados CAN
[b10..b7]	b6	[b5..b0]	0 a 8 bytes
AM_MESG_ID (RT,NRT)	_	0 a 63	Dep. Aplicação

A estrutura do frame desta mensagem é semelhante ao da Mensagem de Dados Síncronos, com a diferença que nesta não há o campo "transmite novo dado", pois trata-se de mensagens de eventos. Existem dois tipos de Mensagens de Dados Assíncronas, que são mapeadas para classes de tráfegos diferentes. As mensagens assíncronas de alta prioridade (**RT**) são aquelas com restrições temporais e têm prioridade na disputa pelo barramento, assim elas pertecem à classe de "tráfego de tempo real". Já as mensagens assíncronas de baixa prioridade (**NRT**) não têm tais restrições temporais e são enviadas segundo a política do melhor esforço (*best-effort*). Estas mensagens assíncronas de baixa prioridade pertencem à classe de "tráfego não tempo real".

As Mensagens de Controle também são do tipo assíncronas e são usadas para realizar o gerenciamento do sistema (e.g. download de software, diagnósticos).

Tabela 7: Campos da Mensagem de Dados Assíncronos

Tipo	**Não Usa**	**ID da Mensagem**	**Dado da Mensagem**
Campo ID do CAN			Campo de Dados CAN
[b10..b7]	b6	[b5..b0]	0 a 8 bytes
CONTROL_MESG_ID (HP,LP)	_	0 a 63	Dep. Aplicação

Assim como nas Mensagens de Dados Assíncronos as Mensagens de Controle podem ser de dois tipos: mensagens de alta prioridade (**HP**) e de baixa prioridade (**LP**). As Mensagens de Controle de alta prioridade possuem a maior prioridade entre todas a mensagens assíncronas, como pode ser visto na Tabela 3. Estas mensagens são utilizada para operações de gerenciamento em tempo crítico, como mudança de requisição da tabela STR (ver Seção 4.5.2). As mensagens de baixa prioridade são utilizada para operações de controle que não envolvam restrições temporais, como por exemplo diagnóstico remoto, atualização de software e configurações de *set-point*.

4.5.2 SMS

O SMS (*Synchronous Messaging System*) é utilizado para enviar mensagens *time-triggered*, também chamadas de síncronas pelo fato de estarem síncronizadas com as ECs. No início de cada EC há a *trigger message* (TM) que contém no campo de dados o mapa de bits específicando quais mensagens deverão ser produzidas/consumidas pelos nós da rede.

Todos os nós da rede decodificam a TM e aqueles que deverão produzir ou consumir alguma mensagem síncrona aguardam o início da janela síncrona para iniciarem suas

transmissões/recepções. Eles tentarão transmitir em intervalos de tempo precisos em relação ao início da janela síncrona, o que configura a formação de *slots* de tempo para a transmissão das mensagens. As eventuais disputas pelo acesso ao barramento dentro destes *slots* de tempo são resolvidas pelo protocolo MAC nativo do CAN.

O nó mestre possui uma tabela com os atributos temporais das mensagens assíncronas (*Synchronous Requirements Table - SRT*). Cada registro nesta tabela descreve as características da mensagem, conforme a definição:

$$SRT \equiv \{SM_i(DLC_i\ C_i\ Ph_i\ P_i\ D_i\ Pr_i),\ i = 1 \ldots N_s\}$$

Onde *DLC* é o tamanho em bytes (de 0 a 8), *C* é o tempo máximo de transmissão (levando em conta os *stuff bits* do CAN, como visto na Seção 4.2.3), *Ph* é a fase relativa, ou seja, a parte de que ponto em relação ao início das transmissões das ECs a mensagem começará a ser enviada. O *P* é o período, *D* é o *deadline* e *Pr* é a prioridade da tarefa. Os atributos *Ph*, *P* e *D* são valores inteiros múltiplos de *E* (duração da EC). N_s é o número de mensagens síncronas (registros da SRT).

O escalonador utiliza a tabela SRT para realizar o escalonamento das mensagens síncronas para cada EC. O resultado da escalação das mensagens é inserido na área de dados da TM (codificada como mapa de bits) da EC correspondente. Como o Escalonador de Planejamento é utilizado, qualquer mudança realizada na SRT em tempo de execução só terá efeito na próxima criação dos planos.

Visando aumentar a flexibilidade e resolver o problema do atraso, gerado pela criação de planos, os autores em (MARTINS; FONSECA, 2001) implementaram um co-processador usando FPGA que varre a STR e realiza o escalonamento das mensagens de cada EC. Esta solução reduz a latência necessária para que as modificações realizadas nos atributos das tarefas tenham efeito, melhorando o tempo de resposta do FTT-CAN.

O fato do escalonamento das mensagens no protocolo FTT-CAN ser realizado baseando-se nos atributos da SRT e não nos identificadores das mensagens permite que várias politicas de escalonamento sejam implementadas, como por exemplo: Rate-Monotonic (RM), Deadline-Monotonic (DM), Earliest-Deadline First (EDF), Least-Laxity First (LLF) entre outras.

O uso do EDF torna possível a utilização de quase 100% da largura de banda (janela síncrona) disponível, porém 100% de utilização não é obtido porque as mensagens no barramento CAN não podem ser “preemptadas” (PEDREIRAS, 2003).

4.5.3 AMS

O protocolo FTT-CAN suporta a transmissão de mensagens assíncronas utilizadas para comunicação *event-triggered*, que são manipuladas pelo AMS (*Asynchronous Messaging System*). Este subsistema utiliza o processo de arbitragem nativo do CAN para realizar a transmissão das mensagens, neste caso o ID da mensagem é utilizado como prioridade da mensagem.

No entanto todos os nós obedecem a definição de só tentar enviar as mensagens assíncronas durante a janela assíncrona, iniciada logo após a recepção da TM. Se um nó não conseguir enviar sua mensagem na janela assíncrona atual, ele deverá aguardar o início da próxima janela assíncrona.

Caso a carga de dados síncronos na rede esteja alta, a janela assíncrona será encurtada, portanto a capacidade de comunicação através da AMS será diminuída. Para garantir uma largura de banda mínima para as mensagens assíncrona, pode-se estabelecer o tamanho

máximo para a duração da janela síncrona (*LSW*).

Um impacto negativo pode acontecer durante a fase assíncrona, caso os nós não estejam precisamente sincronizados com o início da janela assíncrona. Neste caso pode acontecer de um nó que transmite uma mensagem menos prioritária estar sincronizado com o início da janela e outro nó que transmite uma mensagem mais prioritária estar atrasado em relação ao início da janela, neste caso a mensagem menos prioritária será transmitida primeiro e a mais prioritária terá que aguardar a mensagem atual terminar sua transmissão.

Em (ALMEIDA; PEDREIRAS; FONSECA, 2002) os autores sugerem uma técnica para resolver este problema de bloqueio. A idéia é que os nós comecem a tentar transmitir as mensagens assíncronas assim que detectarem o início de uma mensagem TM. Assim todos os nós ficarão bloqueados aguardando o fim da TM (começo da janela assíncrona) quando poderão disputar ao mesmos tempo para enviar as mensagens assíncronas.

As mensagens assíncronas não podem se estender além do fim da janela assíncrona, do contrário poderão bloquear as mensagem síncronas. Para evitar que isso aconteça as mensagens que não couberem perfeitamente dentro da janela assíncrona deverão ser removidas do *buffer* de transmissão e o barramento ficará ocioso até o fim da janela assíncrona. Esta técnica garante que a janela síncrona não será atrasada, porém tem um impacto negativo, pois diminui a largura de banda disponível.

A transmissão das mensagens assíncronas acontecem explicitamente através da requisição da aplicação. Uma aplicação que desejar transmitir uma mensagem deverá chamar a função não bloqueante *AMS_send*. A mensagem será enfileirada no *buffer* de transmissão, sendo ordenada primeiro pela prioridade (de acordo com o identificador da mensagem) e segundo pelo instante da requisição (FCFS).

As mensagens assíncronas são recebidas pela aplicação através da chamada da função bloqueante *AMS_receive*, esta função pode aguardar por uma mensagem específica ou por qualquer mensagem. O AMS enfileira as mensagens recebidas da rede até elas serem consumidas pela aplicação através da chamada *AMS_receive*.

Outros fatores também impactam no comportamento temporal do tráfego síncrono no FTT-CAN. Por exemplo, o SMS manipula o tráfego síncrono com controle autónomo, ou seja, a transmissão e recepção de mensagens é realizada exclusivamente pela interface de rede sem qualquer intervenção do software da aplicação (ALMEIDA; PEDREIRAS; FONSECA, 2002).

Os serviços oferecidos pelo SMS seguem o modelo produtor-consumidor e são disponibilizados para a aplicação através das chamadas *SMS_produce* para escrever uma mensagem no *buffer* da interface de rede e *SMS_consume* para ler uma mensagem recebida pela interface de rede.

Além destes serviços também são oferecidos serviços para gerenciar a tabela SRT, como *SRT_add*, *SRT_remove* e *SRT_change*. Estes serviços permitem controlar de forma automática a admissão, remoção e modificação de mensagens na rede de forma a assegurar as garantias temporais do sistema.

5 AUTOSAR E A REUSABILIDADE

Segundo (VINCENTELLI, 2000) o reuso de componentes é fator chave para diminuir o tempo de desenvolvimento de novos produtos e para diminuição do custo de produção. Ainda segundo o mesmo autor a metodologia usada para projetar sistemas complexos, como os da indústria automotiva, precisa começar em alto nível de abstração para ser eficiente.

Na prática não é o que acontece, pois os projetos de sistemas embarcados em geral são desenvolvidos com um nível de abstração muito próximo ao nível de implementação. Assim o compartinhamento de componentes entre projetos diferentes torna-se difícil ou impossível, pois pequenas modificações exigirão o desenvolvimento de um componente totalmente novo.

Com o desenvolvimento do padrão AUTOSAR a indústria automotiva poderá pela primeira vez focar-se nos requisitos e definições dos produtos e não em sua implementação. Fabricantes concorrentes poderão compartilhar componentes básicos, mas essenciais para a segurança do motorista, e desenvolver apenas itens que agregam valor e/ou proporcionam conforto ao usuário.

O padrão AUTOSAR será apresentado a seguir, na Seção 5.1.

5.1 O padrão AUTOSAR

5.1.1 Visão geral do AUTOSAR

O consórcio AUTOSAR foi criado em 2003 pelas fabricantes automotivas, fornecedores de componentes automotivos e desenvolvedoras de ferramentas (softwares CAD, IDE, etc) com o objetivo inicial de padronizar uma arquitetura de software para as Electronic Control Units (ECUs) (PAPERS, 2004). Na Figura 16 é apresentado os benefícios entre as partes envolvidas no consórcio.

A substituição das soluções proprietárias por um padrão aberto traz vários benefícios.

Benefícios comum a todos os participantes:

- Aumento do reuso de software;
- Aumento da flexibilidade de projeto;
- Regras de projeto claras para integração;
- Redução do custo de desenvolvimento de software e serviços;
- Foco na proteção, inovação e funções competitivas.

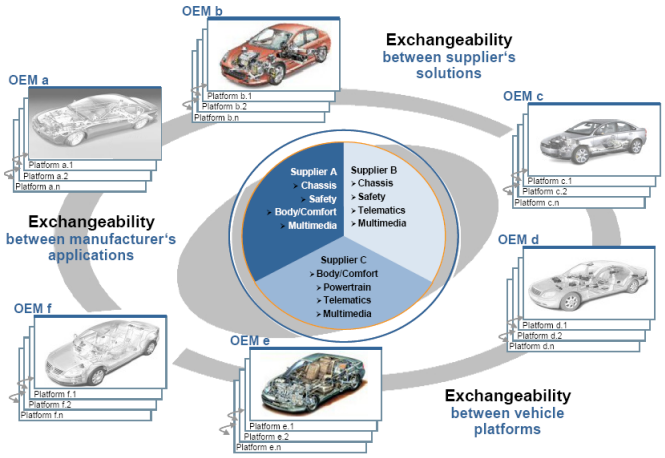


Figura 16: Benefícios do AUTOSAR (PAPERS, 2006)

Benefícios específicos para as fabricantes:

- Funções de natureza competitiva podem ser desenvolvidas separadamente;
- As últimas inovações compartilhas estarão acessíveis;
- Processos padronizados para adaptação.

Benefícios específicos para os fornecedores:

- Redução de profileração de versões;
- Desenvolvimento compartilhado entre fornecedores;
- Aumento da eficiência em desenvolvimento funcional;
- Novo modelo de negócios;
- Preparação para o aumento do volume de software que está por vir.

Mas os esforços dessa iniciativa não estão limitados apenas à infra-estrutura de software. O foco se expande à especificação de interfaces funcionais compatíveis e uma metodologia de desenvolvimento coerente, baseada em modelos e formatos para troca de dados (PARTNERSHIP, 2006).

Os componentes de software dos sistemas automotivos em geral são fabricados pelas montadoras, fornecedores ou por uma companhia de software independente (PELZ et al., 2005). Isto gera alguns problemas:

- a troca do microcontrolador em geral acarreta várias mudanças no software;
- como cada solução é proprietária, raramente é possível a comunicação entre aplicações das montadoras e dos fornecedores.

Para resolver estes problemas o AUTOSAR propõe duas soluções. A primeira é a definição de uma camada de abstração do hardware (HAL - *Hardware Abstration Layer*). Uma HAL pode ser definida como: "todo software que é diretamente dependente de um hardware específico, mas que cria uma camada padrão para outro software dependente

deste" (YOO; JERRAYA, 2003). Assim para que um determinado sistema que implemente o padrão AUTOSAR funcione em outro microcontrolador basta que a HAL seja portada para este microcontrolador.

A segunda solução adotada foi a criação de um barramento padronizado em software chamado "Virtual Functional Bus", este barramento permite abstrair a camada de comunicação entre aplicações rodando numa mesma ECU ou em ECUs distintas. Os detalhes técnicos sobre o funcionamento destas tecnologias do AUTOSAR estão apresentados na Seção 5.1.2.

5.1.2 O VFB do AUTOSAR

A grande inovação do padrão AUTOSAR é sem dúvida o *Virtual Functional Bus* (VFB). É ele quem permite desacoplar as aplicações da infra-estrutura. Uma aplicação é constituída por um ou mais "AUTOSAR Software Component" (SW-C), que podem ser atômicos ou compostos.

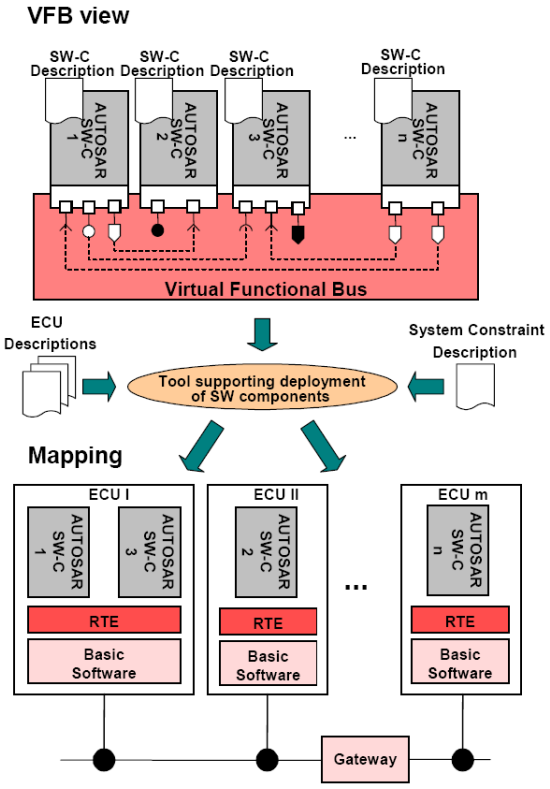


Figura 17: Visão do VFB (PAPERS, 2006)

Na parte superior da Figura 17 é apresentado o conceito do VFB, ele disponibiliza métodos e serviços padronizados para comunicações entre as SW-Cs. O VFB abstrai o meio físico de comunicação entre as SW-Cs, portanto não importa se a comunicação é intra-ECU ou inter-ECU e nem qual barramento físico estão utilizando, a forma como elas se conectam ao barramento permanecerá inalterada.

O VFB utiliza o conceito de portas para conectar os SW-Cs, serviços e *device-drivers*. Portas são pontos de interação bem definidos através de interfaces que fornecem ou requerem dados ou serviços, assim as portas podem ser do tipo *require* (R-port) ou *provide* (P-port).

O padrão AUTOSAR define dois modos diferentes para comunição, o modo Client-Server e o Sender-Receiver. No paradigma Client-Server a comunicação pode ser síncrona ou assíncrona, o cliente poderá ficar bloqueado aguardando a resposta de sua requisição ou poderá continuar com seu processamento e ser notificado quando a resposta chegar. Já no paradigma Sender-Receiver a comunicação é realizada sempre de forma assíncrona.

A Figura 18 apresenta estes modos de comunicação.

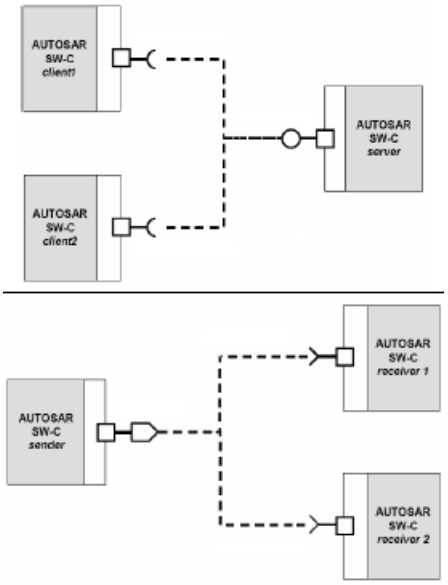


Figura 18: Comunicação Client-Server/Sender-Receiver (PAPERS, 2004)

5.1.3 A arquitetura de software do AUTOSAR

A arquitetura de software do AUTOSAR é composta por três camadas. A camada superior é formada pela camada de Aplicação, que é composta pelos SW-C executando localmente ou de forma distribuída. A camada intermediária é formada pelo Runtime Environment (RTE), é nesta camada que o VFB é implementado. Finalmente, a camada de baixo é a Basic Software (BSW). Na BSW é onde está implementada a abstração do hardware (HAL). A Figura 19 apresenta as camadas do AUTOSAR e suas subdivisões.

A camada de Aplicação contém os AUTOSAR software components (SW-Cs) que são mapeados para cada ECU, segundo seus requisitos específicos. Mas a implementação dos SW-C é independente do microcontrolador, da ECU e também da localização física de outros componentes no sistema (PAPERS, 2006).

A camada RTE funciona como um elo de ligação entre a camada de Aplicação e a BSW. A RTE disponibiliza para a camada de Aplicação, via VFB, todos os serviços e funcionalidades implementados na camada BSW.

A camada BSW é subdividida nas seguintes camadas:

- Camada de Serviços;
- Camada de Abstração da ECU;
- Camada de Abstração do Microcontrolador;
- Camada de Drivers Complexos.

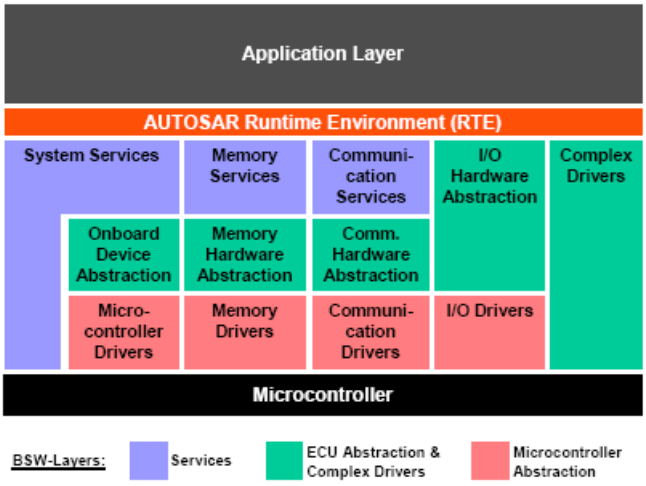


Figura 19: Camadas de software do AUTOSAR (PAPERS, 2006)

A camada de Serviços provê serviços do sistema, como gerenciamento de memória, protocolos de diagnósticos, gerenciamento de energia e sistema operacional. Com excessão do sistema operacional, todos estes serviços são independentes do hardware. A camada de Abstração da ECU abstrai o acesso aos periféricos e outros recursos da ECU para a camada superior. Apesar dessa camada ser específica para cada ECU ela é independente do microcontrolador utilizado. Para conseguir esta independência do microcontrolador existe a camada de Abstração do Microcontrolador que disponibiliza para a camada de Abstração da ECU vários drivers para acesso a I/O, ADC, SPI e outros. Finalmente a camada de Drivers Complexos é disponível para que as fabricantes possam implementar dispositivos que não podem ser padronizados pela especificação AUTOSAR, como aqueles que envolvem aplicações de tempo real ou que lhes dêem vantagens competitivas.

5.2 Críticas e sugestões ao padrão AUTOSAR

Em geral as principais fontes bibliográficas citam apenas os benefícios do padrão AUTOSAR, deixando de lado os seus pontos fracos. O autor desta dissertação acredita que é importante ressaltar os pontos que o padrão AUTOSAR precisa ser melhorado e propor sugestões para que estas lacunas sejam analisadas por outros pesquisadores.

Segundo (SCHREINER; SCHORDAN; GÖSCHKA, 2009) o *middleware* convencional do AUTOSAR tende a ser pesado para os sistemas embarcados utilizados no setor automotivo. Isto se deve principalmente as várias camadas de software utilizadas no sistema operacional e ao sistema de controle de mensagens que tenta ser flexível o bastante para permitir a utilização de múltiplas interfaces de redes e o roteamento de diferentes tipos de protocolos através delas, o que permite por exemplo a comunicação entre o barramento CAN e Flexray de forma transparente. Realmente as fabricantes procuram ao máximo a redução de custos, o que envolve a escolha de microcontroladores com baixo poder de processamento e quantidade de memória reduzida. A utilização de microcontroladores de 8 e 16 bits, que são mais baratos que os de 32 bits, é muito comum neste setor.

Outro problema que impede que o sistema AUTOSAR seja mais leve é a granularidade das configurações, que são definidas apenas em nível de aplicação. Por outro lado um nível de granularidade menor poderia deixar a configuração do AUTOSAR ainda mais

complexa, e pequenas diferenças entre implementações de fabricantes distintos poderiam atrapalhar a interoperabilidade entre os dois sistemas.

O padrão AUTOSAR também deixa de fora a especificação de segurança com relação a comunicação com o mundo externo. A comunicação com todas as ECUs do automóvel pode ocorrer através do uso das ferramentas de diagnósticos ou mesmo através das interfaces wireless (WiFi) e Bluetooth existentes no painel de instrumentos de alguns veículos modernos. Além disso a atualização de firmware do automóvel, que hoje é realizada nas oficinas especializadas, provavelmente passará a ser realizada remotamente (como acontece nos celulares), economizando custos para as fabricantes e para os proprietários. Isto abre as portas para novos tipos de ataques e crimes que podem por em risco a vida do proprietário do automóvel e de sua família.

Portanto a especificação deve se antecipar em relação a estes problemas e propor formas de evitar que estes tipos de falhas aconteçam. Medidas simples de proteção como a autenticação do firmware antes da execução do seu conteúdo poderiam auxiliar no processo segurança, mas apenas isso não seria suficiente se o sistema não for pensado para ser seguro desde sua concepção.

Outra sugestão que o autor deste trabalho deixa para o consórcio AUTOSAR é a criação de *middleware* comum para utilização por todas as fabricantes e fornecedores automotivos, como acontece com o Linux e as várias empresas que o utilizam. Apenas a especificação não é suficiente para garantir total compatibilidade entre as diferentes implementações. Assim eles poderiam criar uma base de código padrão que seria utilizado e melhorado por todos que façam parte do projeto ou até mesmo por desenvolvedores independentes (caso o projeto fosse de código fonte aberto).

6 ESTRATÉGIA PARA INSERÇÃO DINÂMICA DE NODOS NO FTT-CAN

A inserção dinâmica de nodos significa a inclusão de uma placa numa rede com a mesma já em funcionamento, sem com isto atrapalhar o funcionamento desta rede. Neste trabalho o autor desenvolve uma estratégia para inserção de nós numa rede FTT-CAN, implementa esta rede e analisa o funcionamento da mesma em relação a suas características de tempo real.

Neste contexto qualquer nó que for adicionado à rede com ela em funcionamento não deverá corromper ou provocar atrasos nas messagens que estão sendo transmitidas nela e acima de tudo deverá integrar-se perfeitamente nesta rede. Analisando estes dados será possível concluir se uma rede FTT-CAN de tempo real com suporte a inserção dinâmica de nós (*hot-plugging*) é adequada para aplicações de tempo real críticas (*hard realtime*) ou de tempo real leve (*soft realtime*).

Embora muitos *transceivers* CAN suportem *hot-plugging* há alguns anos (CORRIGAN, 2006), a literatura técnica e acadêmica explorando as possibilidades deste assunto é muito escassa.

Além do suporte a inserção física pelo *transceiver*, para que um sistema de hot-plugging funcione numa rede FTT-CAN, inicialmente o nós mestre precisa suportar admissão online de mensagens. Este tema é tratado na Seção 6.1 deste capítulo.

Este protocolo de admissão de mensagens precisa ser bem definido entre os nós escravos e o mestre. Na Seção 6.2 é apresentado o protocolo que foi implementado para alcançar este objetivo.

6.1 Admissão online de novos nós na rede FTT-CAN

A admissão online de mensagens já foi tratada por outros pesquisadores (FONSECA et al., 2000), mas estas eram realizadas com o nó já conectado fisicamente à rede. Na pesquisa realizada por aqueles autores o nó que desejasse transmitir mensagens na rede deveria enviar uma solicitação ao mestre central (mestre) e este usaria os dados da solicitação para criar o plano de escalonamento de mensagens.

O plano de escalonamento é um meio termo entre o escalonamento dinâmico de mensagens e o escalonamento estático. No escalonamento dinâmico o nó mestre precisa atualizar a sua tabela a cada ciclo de alocação das mensagens a serem transmitidas na rede. No escalonamento estático esta tabela é imutável e é criada antes mesmo da rede entrar em funcionamento (*offline*). Já no plano de escalonamento o que existe são planos de escalas (tabelas) que definem quais mensagens deverão ser admitidas no plano atual *i*, enquanto o escalonador cria o próximo plano (*i + 1*), que será executado futuramente.

O funcionamento do plano de escalonamento original é apresentado na Figura 20.

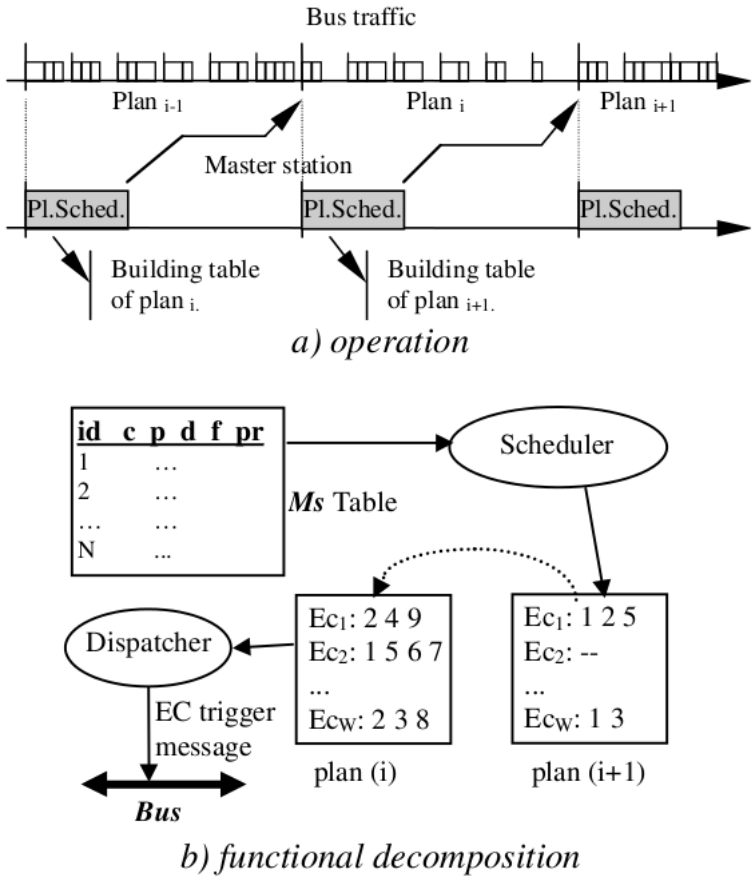


Figura 20: Plano de Escalonamento Original. (ALMEIDA et al., 1999)

Através do plano de escalonamento é possível criar redes mais flexíveis e dinâmicas de forma mais simples do que usando o escalonamento dinâmico, porém mesmo o plano de escalonamento poderá gerar atrasos para a rede FTT-CAN. Para resolver este problema, em (FONSECA et al., 2000) foi implementado um co-processador em hardware exclusivamente para desempenhar a função do escalonador de planos.

A vantagem que o plano de escalonamento gera em termo de flexibilidade é descompensado pela necessidade de um hardware especial, necessário para evitar que a criação

do próximo plano atrapalhe a despacho correto das mensagens atuais.

Visando resolver este problema de forma simples, o autor desta dissertação procurou formas de criar uma escalonamento simples, mas que não gerasse atrasos na rede FTT-CAN.

Assim, uma nova abordagem que elimina a criação de um novo quadro de escalonamento foi desenvolvida e desta forma simplificou a implementação de uma rede mais flexível e sem o *overhead* normalmente encontrado nas redes de escalonamentos dinâmicos e na de plano de escalonamento.

Esta abordagem é apresentada na Seção 6.1.1.

6.1.1 Simplificando o plano de escalonamento

No plano de escalonamento original enquanto as mensagens do plano atual estão sendo despachadas o escalonador está ocupado calculando o próximo plano de escalonamento. Qualquer demora ou atraso na criação do novo plano poderá significar uma falha na sincronização do início da *Trigger Message* do próximo ciclo elementar (EC). Quando isso ocorre, os períodos das mensagens do próximo EC não poderão ser cumpridos.

Para reduzir as chances deste problema acontecer este trabalho simplifica a criação da tabela de escalonamento, primeiro utilizando um algoritmo de distribuição linear dos períodos das mensagens e em seguida reduzindo os planos de escalonamentos a um plano único.

As mensagens de todos os nós são colocadas numa lista interna (chamada de plano único de escalonamento) no nó central que calcula o momento em que cada mensagem será transmitida no barramento usando como critério o período da mensagem.

Na Figura 21 é apresentada a lógica de funcionamento do plano único de escalonamento.

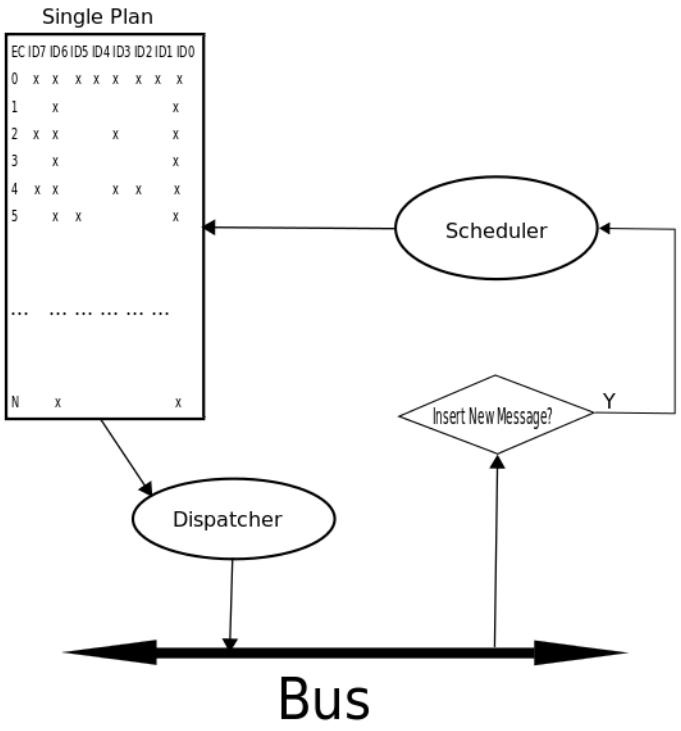


Figura 21: Plano Único de Escalonamento

Quando o nó mestre é ligado sua lista do plano único de escalonamento (*single plan*) não contém nenhuma mensagem a ser transmitida.

À medida que nós escravos são inseridos na rede, eles deverão enviar solicitações de inserção de sua mensagem síncrona. Esta solicitação de inserção de nova mensagem é usada para criar o plano único de escalonamento.

Sempre que uma mensagem de solicitação de inserção é enviada na rede, o nó mestre deverá analisar esta solicitação e chamar o escalonador para inseri-la na tabela do plano único de escalonamento. Cada mensagem deverá possuir um identificador distinto, isto significa que para cada identificador será reservado um *slot* com seu ID em todos ciclos elementares (ECs), mesmo que a mensagem tenha um período que não necessite ocupar todos os ECs do plano de escalonamento.

A separação de fase síncrona do FTT-CAN em *time-slots* foi uma melhoria proposta por (ATAIDE, 2010) que reduz a probabilidade de inversão de prioridade entre as mensagens síncronas. No protocolo original todas as mensagens síncronas tentavam transmitir ao mesmo tempo no início da fase síncrona, porém o deslonamento das mensagens causado pelo *bit-stuffing* era negligenciado.

A separação em *time-slots* também supre a necessidade da mensagem informar sua prioridade, uma vez que o próprio ID identifica a prioridade da mensagem e em qual *time-slot* ela será transmitida.

No caso da rede usada para teste, cada ciclo elementar é transmitido a cada 5ms, então o período mínimo de transmissão da mensagem é de 5ms e este período precisa também ser múltiplo de 5ms. Portanto temos:

$$P_m \geq 5\text{ms}$$
$$P_m \ \% \ 5\text{ms} = 0$$

Onde P_m é o período da mensagem a ser transmitida no barramento.

O plano de escalonamento utilizado neste projeto Steer-by-Wire é composto de 40 ciclos elementares, portanto o período máximo que uma tarefa poderá ter é de 200ms:

$$5\text{ms} \leq P_m \leq 200\text{ms}$$

Na implementação atual a fase síncrona poderá ter até 8 mensagens, cada uma poderá possuir um tamanho de até 8 bytes. Esta limitação pode ser contornada aumentando o tamanho da TM (*Trigger Message*) para 2 ou mais bytes, não excedendo os 8 bytes que é a quantidade máxima de bytes que uma mensagem CAN pode carregar, portanto o limite máximo de mensagens é de 64 mensagens síncronas (*Time-Trigger*).

O escalonador rodando no nó mestre recebe os dados da mensagem a ser inserida na rede: ID da nova mensagem, período e quantidade de bytes a serem transmitidos. Se a mensagem possuir o período igual a 1 (= 5ms) o escalonador deverá percorrer de 1 em 1 as 40 posições do plano de escalonamento ativando o ID desta mensagem. Caso a mensagem tivesse um período igual a 2 (= 10ms) o escalonador deveria percorrer de 2 em 2 as posições da tabela, portando a mensagem seria ativada apenas 20 vezes.

Esta estratégia reduz bastante a sobrecarga que um escalonador terá para alocar as novas mensagens de forma dinâmica, porém o resultado do escalonamento não é o mais eficiente em termo de distribuição das tarefas ao longos dos ECs existens no plano de escalonamento.

Um exemplo contendo o plano único de escalonamento com as mensagens de diferentes períodos é apresentado na Figura 22.

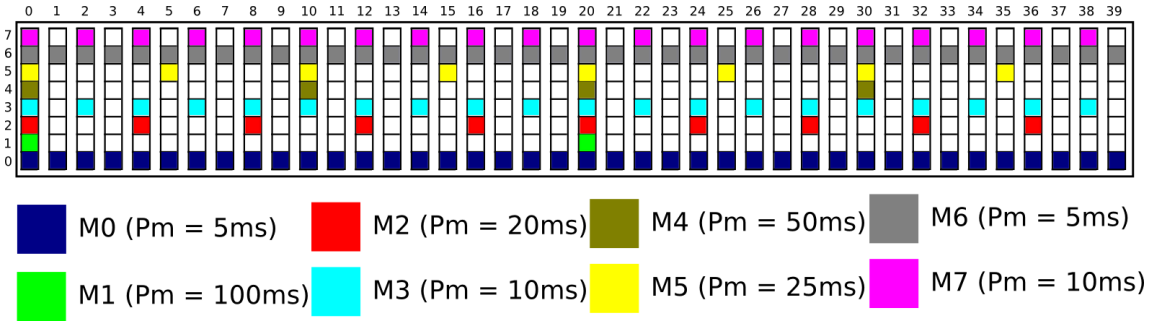


Figura 22: Ocupação Plano Único

Como pode ser visto na Figura 22 a estratégia adotada para escalonamento ocupa todos os *time-slots* dos ECs 0 e 20, enquanto mantém os ECs 1, 3, 7, 9, 11, 17, 19, 21, 23, 27, 29, 31, 33, 37, 39 com apenas 2 *time-slots* utilizados.

Esta ineficiência do algoritmo para ocupar de forma mais uniforme os ECs do plano não é tão ruim quanto à primeira vista pode parecer. Como o protocolo FTT-CAN ajusta o início do fase síncrona conforme a quantidade de mensagens síncronas existentes no ciclo elementar, sobra mais largura de banda para as mensagens assíncronas.

6.2 Estratégia implementada para autodeteção

Quando um novo nó é fisicamente inserido na rede ele deve inicialmente enviar uma mensagem assíncrona com o ID 0x55 e com os seguintes campos: ID da mensagem que deseja ser transmitida na rede; período da mensagem que este nó enviará; quantidade de bytes que a mensagem contém. O formato da mensagem usado para autodeteção dos nós no barramento FTT-CAN é apresentado na Figura 23.

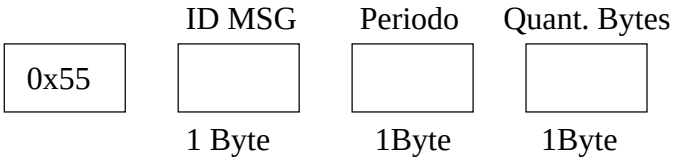


Figura 23: Formato da mensagem de detecção de nó

O nó mestre recebe esta mensagem assíncrona e deverá usar as informações existentes nos campos: período e quantidade de bytes para fazer o novo plano de escalonamento das mensagens. A quantidade de bytes da mensagem é utilizada para calcular o tempo que a mensagem ficará ocupando o barramento quando for transmitida.

Após computar a solicitação o nó mestre deverá responder no próximo EC com uma mensagem assíncrona de ID 0xAA com os campos: ID do nó que solicitou a inserção na rede e o código de resultado da solicitação. O formato da mensagem de reconhecimento de inserção de mensagem é apresentado na Figura 24.

O significado de cada código de resultado retornado pelo mestre é apresentado na Tabela 8.

Caso o nó necessite enviar mais de uma mensagem ele deverá repetir o processo de inserção de mensagem na rede. Isto é necessário porque apenas uma mensagem pode ser inserida por vez no plano de escalonamento do nó mestre.

A mensagem não precisa definir sua prioridade, pois na nova abordagem do protocolo

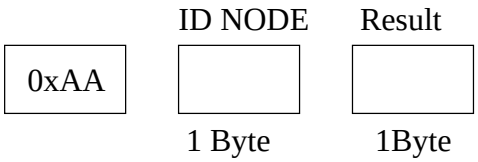


Figura 24: Formato da mensagem de resultado de inserção

Tabela 8: Código dos resultados da solicitação de inserção na rede

Código	**Significado**
0x00	Mensagem inserida com sucesso
0x01	Sem time-slot disponível
0x02	Período inválido
0x03	Quantidade de bytes inválida

do FTT-CAN desenvolvida por (ATAIDE, 2010) as mensagens *Time-Triggered* ficam isoladas em seus devidos *time-slots* num padrão semelhante ao TDMA. Portanto uma mensagem não poderá ocupar a posição de outra como poderia ocorrer no protocolo FTT-CAN original. Nesta nova abordagem o ID da mensagem indica sua posição na fase assíncrona, sendo as posições são definidas em ordem crescente de ID, ou seja, as mensagem com ID menor são transmitidas primeiro.

A implementação e validação deste protocolo de inserção dinâmica de nós (*hot-plugging*) é apresentada no Capítulo 7 deste trabalho. No mesmo capítulo também são apresentados todos os detalhes da implementação de um sistema Steer-By-Wire, utilizado para demonstrar a viabilidade prática da proposta.

7 IMPLEMENTAÇÃO E VALIDAÇÃO DA PROPOSTA

O objetivo deste trabalho foi implementar e validar um sistema de *hot-plugging* para a rede FTT-CAN. Assim um novo nó pode ser adicionado à rede FTT-CAN e automaticamente comunicar com o nó mestre da rede para definir suas mensagens periódicas de tempo real (*Time-Trigger*) que serão enviadas pela rede. Mensagens esporádicas/assíncronas (*Event-Triggered*) não precisarão ser definidas, pois poderão disputar o barramento na fase assíncrona de todos os ECs.

Foi desenvolvida uma camada de abstração da API OSEK sobre o RTAI/uClinux e as APIs de comunicação para uso das aplicações. Esta camada de abstração foi necessária para criar um sistema compatível com a especificação AUTOSAR.

As APIs de comunicação foram implementadas de forma a usar o protocolo FTT-CAN e manter uma comunicação transparente para as aplicações rodando em nível mais alto. Isto significa que as aplicações utilizam a API de comunicação sem ter conhecimento do protocolo implementado pela camada de abstração.

Como forma de demonstrar o funcionamento da inserção dinâmica de nós foi desenvolvido um protótipo que simula um sistema *Steer-by-Wire* rodando sobre a rede FTT-CAN e que permite a inclusão de novos nós na rede sem atrapalhar o funcionamento da mesma. Este sistema adquire o ângulo de rotação do volante, obtido através de um joystick volante (*steering wheel joystick*), e envia para outras placas na rede FTT-CAN encarregadas de processar e fazer a atuação sobre as rodas.

O restante deste capítulo está organizado da seguinte forma: primeiro é apresentado na Seção 7.1 são apresentado os passos necessários para a implementação da camada de abstração do AUTOSAR OS sobre o RTAI/Linux; na Seção 7.3 o hardware utilizado neste projeto; e finalmente na Seção 7.4 é apresentado o protótipo da rede *steer-by-wire* implementado neste trabalho para demonstração de funcionamento.

7.1 Implementação do AUTOSAR OS

O Sistema Operacional do AUTOSAR é baseado no OSEK/VDX, pois este já é amplamente utilizado na indústria automotiva e tem sido usado em várias classes de ECUs. O fato do AUTOSAR ser baseado no OSEK implica que os sistemas legados serão prontamente suportados.

Portanto para integrar o FTT-CAN no AUTOSAR é importante utilizar um sistema operacional compatível com a espeficicação OSEK.

A integração do protocolo FTT-CAN com o AUTOSAR poderia ter sido desenvolvida de uma das duas formas:

1. Implementação do driver FTT-CAN sobre algum sistema OSEK open-source;

2. Implementação de uma camada de abstração das API OSEK sobre o RTAI.

Neste trabalho optou-se por implementar uma camada com as APIs do OSEK rodando sobre o RTAI. Esta decisão simplificou o desenvolvimento do projeto, uma vez que o FTT-CAN já estava portado e funcionando corretamente sobre o RTAI.

No entanto a decisão de implementar uma camada OSEK sobre o RTAI implicou em alguns desafios que não necessariamente seriam encontrados caso optasse pelo desenvolvimento do driver FTT-CAN sobre alguma implementação *Open-Source* do sistema operacional OSEK. Para o desenvolvimento da camada de abstração OSEK sobre o RTAI foi necessário entender o que cada função da API OSEK faz e como ela realiza tal função.

A parte mais difícil na implementação da camada abstração foi fazer com que as funções do RTAI comportassem de forma a imitar as funções do OSEK, conforme explicado na Seção 7.1.1.

Além dos requisitos do padrão OSEK, o AUTOSAR OS define uma série de requisitos que devem ser seguidos. Este requisitos são listados no Apêndice A desde trabalho.

7.1.1 Camada de abstração OSEK OS sobre o RTAI

A criação de uma camada de abstração sobre o RTAI consistiu na implementação de uma API compatível com a especificação 3.2 do OSEK, que internamente chama uma ou mais funções nativas do RTAI. Uma melhor alternativa seria a criação de um *skin* RTAI ADEOS, porém o RTAI ADEOS ainda não foi portado para o microcontrolador Coldfire MCF5282. Por este motivo optou-se pela implementação de uma camada HAL mais simples, convertendo apenas as funções OSEK em chamadas de funções nativas do RTAI.

Com a implementação da API OSEK sobre o RTAI é possível portar facilmente aplicações nativas de outras implementações OSEK. Com isto aplicações legadas serão suportadas e também aplicações que antes rodavam em ECUs com microcontroladores mais simples poderão rodar sobre qualquer processador que suporte o RTAI/Linux.

O desenvolvimento da camada de abstração do OSEK foi facilitado pela existência de várias implementações OSEK *open-source* disponíveis na Internet. Serviram como base para este projeto: FreeOSEK [1], PICOS18 [2] e ToppersOSEK [3]. Verificando o que cada função faz internamente, foi possível delimitar qual a melhor estratégia usar na implementação das chamadas do RTAI.

7.1.1.1 Conversão da API OSEK para o RTAI

Algumas chamadas de serviços do OSEK puderam ser mapeadas diretamente para chamadas equivalentes do RTAI, outras chamadas tiveram que ser implementadas usando duas ou mais chamadas do RTAI. Outras chamadas não puderam ser mapeadas para um conjunto de chamadas do RTAI e por isso tiveram que ser implementadas como uma função completamente nova no RTAI.

Na Tabela 9 é apresentado uma listagem de algumas funções que puderam ser mapeadas diretamente para funções equivalente no RTAI.

Na Tabela 10 é apresentado uma listagem de algumas funções que foram mapeadas para duas ou mais funções do RTAI.

Na Tabela 11 é apresentado uma listagem de algumas funções que não possuiam equivalentes no RTAI e tiveram que ser completamente implementadas.

[1] http://openosek.sourceforge.net

[2] www.picos18.com

[3] http://www.toppers.jp/en/index.html

Tabela 9: Funções mapeadas diretamente para o RTAI

Função OSEK	**Função RTAI**
ActiveTask	rt_task_init
CancelAlarm	stop_rt_timer
ChainTask	rt_task_suspend
ClearEvent	rt_sem_delete
GetTaskState	rt_get_task_state
Schedule	rt_task_yield
TerminateTask	rt_task_delete

Tabela 10: Funções mapeadas para duas ou mais funções RTAI

Função OSEK	**Função RTAI**
GetAlarm	start_rt_timer + next_period
SetAbsAlarm	rt_set_periodic_mode + start_rt_timer
SetEvent	rt_sem_init + rt_sem_signal
SetRelAlarm	rt_set_oneshot_mode + rt_sleep_until
WaitEvent	rt_sem_init + rt_sem_wait

Tabela 11: Funções que tiveram que ser completamente implementadas no RTAI

Função OSEK	**Função RTAI**
GetActiveApplicationMode	-
GetAlarmBase	-
GetEvent	-
GetResource	-
GetTaskID	-
ReleaseResource	-
StartOS	-
ShutdownOS	-

7.1.2 Camada de abstração OSEK COM sobre FFTCAN/RTAI

Como a camada de comunicação do OSEK/AUTOSAR é complexa para uma aplicação que utilizará apenas uma interface de rede e se conectará a apenas um tipo de rede, ela foi simplificada para atender as necessidades deste projeto e com isto diminuir o *overhead* do sistema.

Uma vez que o conceito da rede CAN não lida com o endereço de *Sender/Receiver*, este é assumido como o próprio ID da mensagem transmitida/recebida na rede. Assim uma aplicação que deseja enviar dados para outros nós na rede simplesmente chamará uma função para este fim e o driver FTT-CAN se encarrega de enviar a mensagem no barramento CAN.

Já a comunicação Cliente/Servidor precisa ser implementada em nível de aplicação. Uma aplicação cliente enviará uma mensagem do tipo *Sender/Receiver* para o aplicação servidora rodando em outro nó. A aplicação servidora então enviará uma mensagem de resposta com os dados solicitados.

A função encarregada de enviar uma mensagem na especificação OSEK/VDX é chamada de SendMessage e definida como:

```
StatusType SendMessage(message, data, length)
```

Onde message é o ID da mensagem a ser transmitida; data é o conteúdo da mensagem e length é a quantidade de bytes existentes no conteúdo da mensagem.

Esta função foi implementada criando-se uma mensagem padrão do CAN e enviando-a através das chamadas das funções nativas do driver FTT-CAN.

```
int SendMessage (int message, char *data, int length)
{
  int ret = 0;
  SIMPLE_MESSAGE msg;

  msg.id = message;

  if ( data && length > 0 && length <= 8)
    memcpy(&(msg.data[0]), data, length);
  else {
    printk("Error creating message\n");
    return -1;
  }
  msg.length = length;
  msg.flags = MSG_FLAG_STD;
  msg.code_buf = MSG_CODE_TX;

  ret = set_Buffer(&msg);
  if (ret != 0) {
    printk("No more FlexCAN buffer available!\n");
    return ret;
  }

  /* Verify kind of msg (Sync/Async) */
  if (msg.id > 8)
    TransmitAsyncMsg(msg.id);
  else
    TransmitSyncMsg(msg.id);
  return 0;
}
```

A estrutura SIMPLE_MESSAGE é descrita a seguir:

```
typedef struct {
        /* ID of CAN message */
        unsigned long int id;
        /* Flag of message: Standard, Extended or Remote Frame. */
        unsigned char flags;
        /* Code to indicate if will be a Send or Receive Message. */
```

```
        unsigned char code_buf;
        /* The data of message. */
        unsigned char data[8];
        /* Length of message to be sent. */
        unsigned short int length;
} SIMPLE_MESSAGE;
```

A especificação OSEK/VDX também especifica uma função para receber messagens chamada ReceiveMessage e definida como:

```
StatusType ReceiveMessage(message, data, length)
```

Onde message é o ID da mensagem que deseja-se receber; data é o buffer onde a mensagem será armazenada e length é a quantidade de byte a serem lidos.

```
int ReceiveMessage (int *message, char *data, int *length)
{
  int ret = 0;
  SIMPLE_MESSAGE msg;

  msg.id = message;

  /* Verify type of message to be read */
  if( msg.id > 8)
    ReceiveAsyncMsg(msg.id);
  else
    ReceiveSyncMsg(msg.id);

  ret = get_Buffer(&msg);
  if (ret != 0) {
    printk("Message cannot be received!\n");
    return ret;
  }

  if (msg.length > 0 && length <= 8)
    memcpy(data, &(msg.data[0]), msg.length);
  else {
    printk("Error receiving message\n");
    return -1;
  }

  return 0;
}
```

A implementação destas funções foi necessária para facilitar o porte de aplicações nativas do OSEK COM. De fato a especificação OSEK COM não exige a existência de um sistema operacional compatível com OSEK OS, mas como neste caso ele existe, o desenvolvedor poderá utilizar o mecanismo de notificação através de ativação de tarefa. Isto facilita não só o porte de aplicações OSEK COM, mas também o porte das aplicações OSEK COM implementadas sobre um sistema operacional compatível com o OSEK OS.

7.2 Implementação do protocolo de inserção dinâmica nodos

Para implementar a comunicação seguindo o protocolo de *hot-plugging* (apresentado na Seção 6.2), inicialmente o nó escravo deverá definir uma mensagem assíncrona com ID 0x55 contendo: o ID da nova mensagem síncrona que ele pretende transmitir; o período desta mensagem e a quantidade de bytes que a mesma terá. As linhas de código a seguir ilustram a criação da mensagem 0x55:

```
insertMSG.rtEntityID = 0x55;
insertMSG.data[0] = 0x01; /* ID MSG = 2 */
insertMSG.data[1] = 1; /* Period = 1 => 5ms */
insertMSG.data[2] = 2; /* New NODE will transmit 2 bytes */
insertMSG.size = 3; /* How many bytes to transmit */
insertMSG.type = PRODUC_VAR | ASYNC_VAR;
add_RtEntityOnQueue (&insertMSG);
```

A variável `insertMSG` é do tipo `REALTIME_ENTITY`. A `REALTIME_ENTITY` é uma fila contendo as mensagens que o nó deverá transmitir.

O nó mestre deverá inicialmente configurar seu *buffer* de mensagens para filtrar pela mensagem de ID 0x55, assim quando uma mensagem com este ID chegar, uma interrupcão será levantada. O nó mestra configura seu buffer para filtra pela mensagem de ID 0x55 através do trecho de código:

```
hotplugMSG.id = 0x55; /* ID 0x55 indicates hot-plug a node */
hotplugMSG.flags = MSG_FLAG_STD;
hotplugMSG.code_buf = MSG_CODE_RX;
hotplugMSG.length = 2;
hotplugMSG.msg_buf = 0;
set_Buffer(&hotplugMSG);
```

Quando esta mensagem é recebida a rotina de tratamento de interrupcão copia os dados da mensagem para as variáveis `IDMSG`, `PERIOD`, `SIZENEWMSG` respectivamente. O nó mestre deverá analisar estes dados durante a execução de sua *thread* (`TM_Thread`). Em seguida deverá retornar uma mensagem síncrona com ID 0xAA reportando o resultado da inserção da nova mensagem.

Este pedaço de código é uma versão simplificada do código que é executado na `TM_Thread`:

```
if(newnodeinserted == 1){
        int offset = PERIOD;
        int end = MAXEC / PERIOD;
        int size = SIZENEWMSG;

        if(offset >= MAXEC)
                ackInsertion.data[1] = INVALPERIOD;

        if(size > 8)
                ackInsertion.data[1] = INVALSIZE;

        for(i = 0; i < end; i = i + offset)
```

```
                vectorTM[i] = vectorTM[i] | 1 << IDMSG;

        set_Buffer(&ackInsertion);
        newnodeinserted = 0;
}
```

O tempo necessário para inserir as mensagens no plano único de escalonamento (`vectorTM`) é inversamente proporcional ao período da mensagem. Para inserir uma mensagem com período de 5ms, o que gera um total de 40 iterações no laço FOR, o necessário menos de 48μs como pode ser visto na Figura 25.

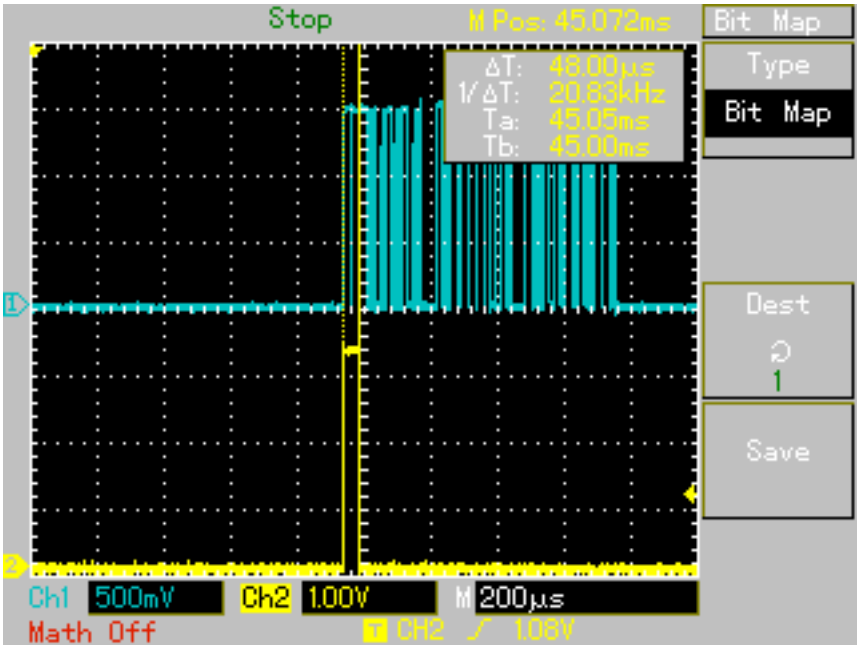


Figura 25: Tempo necessário para inserir uma mensagem com período de 5ms

Para inserir uma mensagem com período igual a 10ms foi necessário 36μs e para inserir uma mensagem com período igual a 100ms foi necessário 32μs.

A mensagem `ackInsertion` é criada da mesma forma que a mensagem 0x55 apresentada anteriormente. Ela contém 2 bytes: o ID da mensagem que tentou ser inserida e o código de sucesso/erro da operação.

O nó escravo é notificado do recebimento da mensagem de ACK (0xAA) e deverá remover a mensagem 0x55 de sua fila de mensagens assíncronas. Esta operação é executada na *thread* `ReceiveAsyncMsg_Task` do nó escravo.

Quando o nó escravo recebe a mensagem de ACK, ele não tem tempo hábil para remover o mensagem 0x55 que está em transmissão no ciclo elementar atual, pois a mensagem já se encontra em processo de transmissão no controlador CAN. Para resolver o problema de repetição da transmissão repetitiva das mensagens 0x55 e 0xAA o nó mestre deverá ignorar o envio da segunda mensagem 0x55, caso esta contenha os mesmo dados da mensagem anterior.

Este problema do envio do segundo 0x55 e a validação do funcionamento deste sistema de inserção dinâmica de nodos é apresentada na Seção 7.5.

7.3 Hardware base do projeto

Nesta seção serão apresentados os equipamentos utilizados para o desenvolvimento do protótipo do sistema *Steer-by-Wire* implementado neste trabalho. Para implementação da rede FTT-CAN foram utilizadas quatro placas M5282Lite, descritas na Seção 7.3.1. Para implementação da função de volante foi utilizado um joystick volante Logitech Momo Racing com interface USB, este equipamento está descrito na Seção 7.3.3.

Como as placas M5282Lite utilizadas não possuem interface USB Host (elas funcionam apenas como USB Device) foi utilizada uma placa MBED da NXP rodando o RTOS NuttX[4], esta placa é apresentada na Seção 7.3.2. Para comunicação da placa MBED com a placa M5282Lite foi utilizado o barramento SPI, assim elas puderam estabelecer uma comunicação bidirecional.

7.3.1 Placa M5282Lite

O hardware principal usado neste trabalho foi a placa de desenvolvimento M5282Lite da empresa Axiom. Esta placa foi escolhida por possuir os recursos necessários para a implementação da rede baseada no protocolo FTT-CAN e principalmente ser uma placa de baixo custo.

A placa M5282Lite está ilustrada na Figura 26.

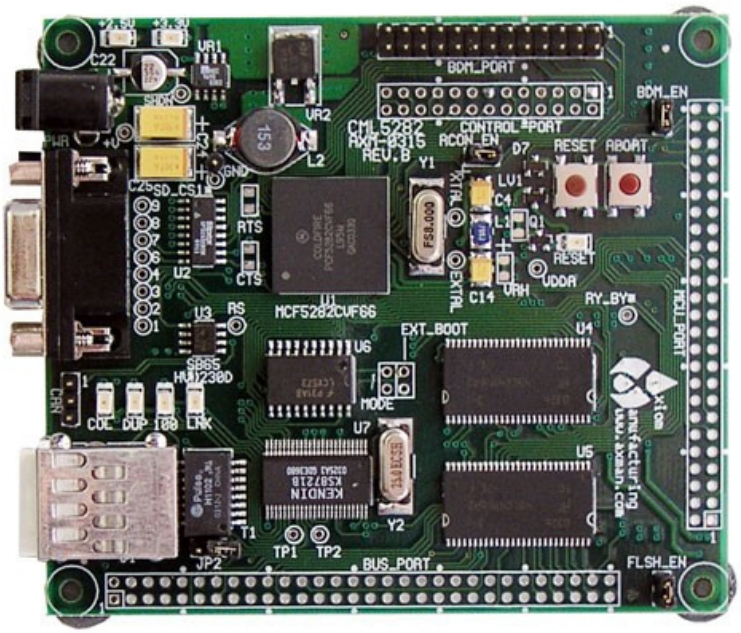


Figura 26: Placa de desenvolvimento M5282Lite

Esta placa possui um microcontrolador Coldfire MCF5282 de 66MHz, possuindo 16MB de memória RAM, 2MB de memória flash, além de interface de rede Ethernet e barramento CAN. O microcontrolador Coldfire MCF5282 está descrito na Seção 7.3.1.1.

A implementação do RTAI que roda sob o uClinux neste microcontrolador foi desenvolvida por Bernhard Kuhn da empresa Lineo. O RTAI funciona como um *nano-kernel* RTOS entre o hardware e o uClinux. As tarefas de tempo real são executadas diretamente pelo RTAI e quando sobra tempo ocioso, este executa o uClinux como se fosse uma tarefa de baixa prioridade

[4]http://nuttx.sourceforce.net

O driver do FTT-CAN enquadra-se na categoria de tarefa de tempo real sendo executada sobre o RTAI. O driver acessa diretamente os registradores do controlador CAN do MCF5282, controlando o momento exato em que determinadas mensagens deverão ser enviadas através do barramento.

As funções de envio e recebimento de mensagens do FTT-CAN foram utilizadas como um conjunto de API da camada COM do AUTOSAR OS. Portanto um desenvolvedor familiar com as APIs do OSEK poderá utilizar o protocolo FTT-CAN de forma transparente.

7.3.1.1 Microcontrolador Coldfire MCF5282

O MCF5282 é um microcontrolador RISC de 32bits que possui um núcleo Coldfire versão 2 (V2), com 24 linhas de endereços e 32 linhas de dados, rodando a 66MHz.

Este microcontrolador possui uma grande quantidade de periféricos integrados, como por exemplo:

- 2 Kbytes de memória *cache* interna, 64 Kbytes de SRAM *dual port* interna, 512 Kbytes de *flash* interna
- MAC Ethernet 10/100
- Módulo CAN 2.0B (FlexCAN), com *buffer* para 16 mensagens
- Controlador de SRAM, 8 *chip selects* programáveis com suporte a flash paginada
- Oito canais de conversores analógico/digitais com precisão de 10 bits
- Três UARTs com DMA, I^2C, I/Os genéricos
- Oito timers de 16 bits para captura, comparação e PWM
- Quatro timers periódicos com interrupção para alarme e watchdog
- Debug via BDM e suporte a JTAG
- Operação em faixa industrial (-40C a +85C)
- Unidade de multiplicação e acumulação (eMAC) para aplicações de processamento de sinais
- 59 MIPS (Dhrystone 2.1) a 66MHz, executando da flash interna
- 3,3V de alimentação para o núcleo mas com I/O com suporte a 5V

Estas características tornam este microcontrolador adequado para aplicações industriais e automotivas. Outros fatores que influenciaram na escolha deste microcontrolador foram o fato de possuir um bom suporte ao uClinux e o fato de possuir um porte estável da interface de tempo real RTAI.

A Figura 27 apresenta a estrutura interna do MCF5282.

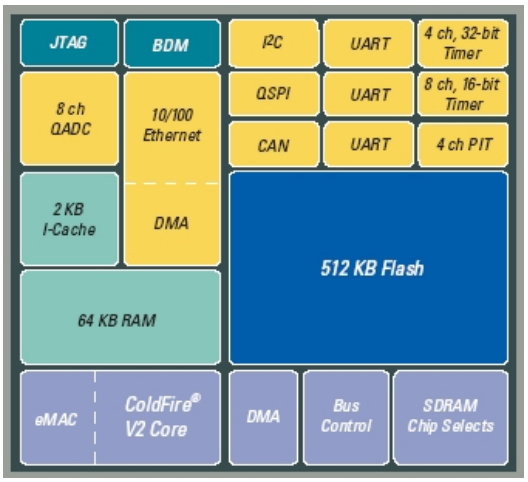


Figura 27: Estrutura Interna do Coldfire

7.3.2 Placa de desenvolvimento MBED

A placa MBED foi utilizada para interfaceamento com o joystick volante Logitech Momo Racing, uma vez que a placa M5282Lite não possui a função de USB Host. Esta placa foi desenvolvida como parte de um projeto independente por dois funcionários da empresa ARM, chamado MBED[5]. O objetivo deles era criar uma placa que permitisse o desenvolvimento rápido de protótipos, para isso eles criaram uma IDE online em AJAX, com vários drivers disponíveis, onde o desenvolvedor pode criar seu programa, compilar e baixar o firmware gerado. Para resolver o problema da interface de programação da placa eles criaram um *bootloader* que monta a memória Flash do microcontrolador através da USB como se fosse um *pen-driver*. Então basta o desenvolvedor salvar neste *pen-driver* o firmware que ele desenvolveu online e reiniciar a placa para seu programa ser executado.

Embora o desenvolvimento usando a IDE online possa acelerar o processo de desenvolvimento, uma vez que não é necessário instalar as ferramentas no sistema operacional e nem realizar nenhuma configuração de variáveis de ambiente, neste projeto foi utilizado a forma convencional de programação instalando o conjunto de ferramentas de compilação GNU GCC.

A placa MBED é apresentada na Figura 28.

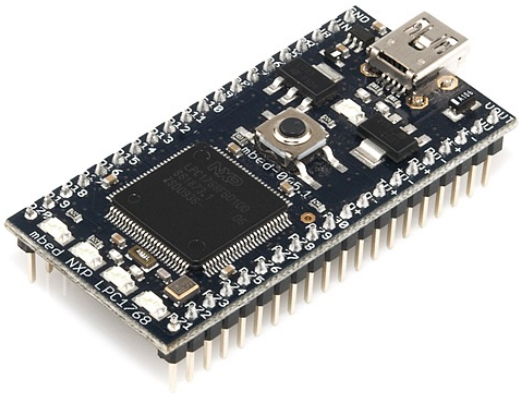


Figura 28: Placa de desenvolvimento MBED

[5]http://www.mbed.org

A placa MBED utiliza o microcontrolador LPC1768 da NXP rodando a 96MHz com 64KB de RAM e 512KB de Flash. O microcontrolador LPC1768 está descrito na Seção 7.3.2.1.

Esta placa utiliza o formato DIP 40 pinos, o que facilita sua montagem sobre um *protoboard*, permitindo que o desenvolvedor ligue outros componentes eletrônicos para interfaceamento direto com a placa.

7.3.2.1 Microcontrolador NXP LPC1768

O LPC1768 é um microcontrolador RISC de 32 bits baseado no núcleo ARM Cortex-M3 rodando a 96MHz.

Este microcontrolador possui internamente vários periféricos, listados a seguir:

- 64 Kbytes de memória SRAM e 512 Kbytes de memória *flash*;
- Unidade de Proteção de Memória (MPU);
- Controlador de DMA de propósito geral (GPDMA) de 8 canais;
- Ethernet com controlador de DMA dedicado;
- Controlador USB 2.0 *full-speed* configurável para função Device, Host ou OTG (*On The Go*);
- Quatro UARTs, sendo uma com suporte nativo a RS485;
- Controlador CAN 2.0B de dois canais;
- Dois controladores SPI;
- Três interfaces I^2C;
- Interface I^2S (*Inter-IC Sound*);
- Conversor Analógico/Digital (ADC) de 12 bits amostrando até 200KHz;
- Conversor Digital/Analógico (DAC) de 10 bits;
- Quatro temporizadores/contadores que podem gerar requisições de DMA;

7.3.3 Joystick Volante Logitech Momo Racing

No desenvolvimento do protótipo *Steer-by-Wire* foi utilizado o volante com *force-feedback* Logitech Momo Racing. Através deste volante seria possível o motorista ter um feedback melhor sobre o terreno onde o carro está sendo conduzido.

A volante Momo Race é apresentado na Figura 29.

Em sistemas *Steer-by-Wire* o volante com *force-feedback* é muito importante, pois torna o controle da direção mais realístico. Por exemplo, quando a roda do carro colide com algum obstáculo o volante deve vibrar e forçar as mãos do motorista na direção horizontal equivalente à da força atuando sobre a roda.

Infelizmente não é possível ligar o joystick volante diretamente na placa M5282Lite, pois esta não possui interface USB host. Foi necessário encontrar uma placa com interface USB host para funcionar como ponte entre o joystick e a M5282Lite. A placa escolhida

Figura 29: Joystick Volante Logitech Momo Racing

foi a MBED (descrita na Seção 7.3.2), por ser uma placa de baixo custo e por possuir as interfaces necessárias para esta aplicação.

A placa MBED não tem recursos suficientes para rodar Linux (nem uClinux), então foi utilizado outro sistema operacional, na verdade um RTOS chamado NuttX. Se esta placa suportasse o Linux bastaria ativar a compilação do driver do joystick Momo Racing para poder utilizá-lo. Como ela não suporta, foi necessário criar um driver para que o joystick volante Logitech Momo Racing funcionasse no NuttX.

Primeiramente foi necessário entender como protocolo do joystick volante funciona para então criar o driver.

O protocolo de *force-feedback* utilizado no joystick Logitech Momo Racing chama-se I-Force. Este protocolo foi criado pela empresa Immersion Corporation, uma empresa que surgiu como fruto da parceira entre o Laboratório de Displays Avançados e Percepção Espacial do centro de pesquisa Ames da NASA e do centro de pesquisa de Design da Universidade de Stanford (NASA, 1997).

Neste protocolo a força de rotação que pode atuar sobre o joystick (i.e. pender para direita ou esquerda) é mapeada num único byte (0 - 255). O valor médio 128 significa que há um equilíbrio entre a força esquerda e direita, neste caso o joystick permanecerá na posição onde se encontrava anteriormente. A medida que o valor decresce (< 128) a força incrementa sua intensidade para a direita. A medida que o valor aumenta (> 128) a força incrementa sua intensidade para a esquerda.

A Figura 30 exemplifica este funcionamento do joystick volante com *force-feedback*.

Analisando o log de comunicação da USB (através do módulo usbmon do Linux) e verificando o driver open-source do Linux, concluiu-se que o byte que codifica a força do joystick *force-feedback* é o terceiro byte (de um total de 7 bytes) enviados ao *Endpoint 1 Out* do dispositivo. A sequência dos bytes enviados é a seguinte:

0x51 0x08 **0xFF** 0x7F 0x80 0x00 0x00

Onde os bytes “0x51 0x08” são usados para indicar que se trata de um comando para definir a força aplicada ao joystick; O valor 0xFF (255) é o valor que codifica a força aplicada ao joystick volante (eixo X), neste caso significa intensidade máxima forçando

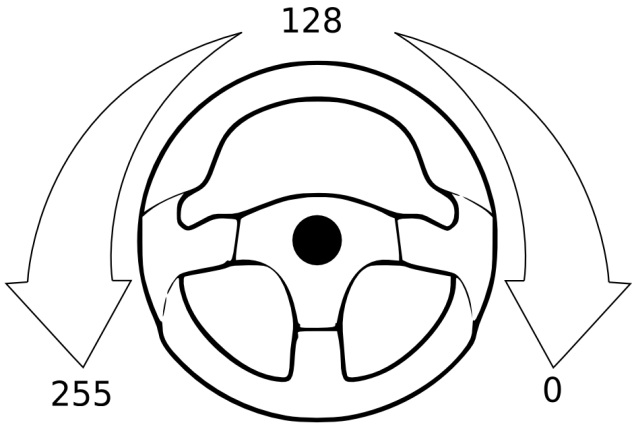


Figura 30: Atuação da força de rotação sobre o volante - protocolo I-Force

para a esquerda; O valor 0x7F é ignorado para o joystick volante, mas nos joysticks com dois eixos (X e Y) serve para codificar a força aplicada ao eixo Y; E finalmente os valores "0x80 0x00 0x00" são constantes em todas as transmissões.

Além da transmissão dos bytes usados para definir a força que movimenta o joystick volante para a esquerda e para a direita, foi identificado o comando *autocenter*, usado para manter o volante sempre forçando para a posição central. Ele é codificado pelo terceiro byte (eixo X) e quarto byte (eixo Y). A sequência dos bytes enviados para o comando autocenter são:

0xFE 0x0D **0x0F 0x0F** 0x80 0x00 0x00

Onde os bytes "0xFE 0x0D" são usado para indicar o comando *autocenter*; O valor 0x0F indica magnitude de 100%, a magnitude pode variar de 0x00 = 0% de magnitude (sem força tendendo para o centro) a 0x0F = 100% de magnitude (força máxima tendendo a manter o joystick volante na posição central).

O driver implementado executa a seguinte sequência:

- Executa uma requisição GET_DESCRIPTOR sobre a interface HID USB do dispositivo, para obter o Descriptor HID;
- Executa o *parsing* do Descriptor retornado para validar se o dispositivo é um joystick e quais eventos ele reporta;
- Executa uma requisição SET_REPORT sobre o *Endpoint 1 Out*[6] para desativar o autocenter do joystick;
- Executa uma requisição GET_REPORT sobre a interface HID para ler a posição do joystick;
- Executa uma requisição SET_REPORT sobre o *Endpoint 1 Out*[7] para definir a força de atuação sobre o joystick.

[6] A sequência de 7 bytes enviada é: [0xFE, 0x0D, 0x00, 0x00, 0x80, 0x00, 0x00], onde os 2 primeiros 0x00 significam magnitude 0, ou seja, autocenter desativado

[7] A sequência de 7 bytes enviada é: [0x51, 0x08, 0x80, 0x7F, 0x80, 0x00, 0x00], onde o primeiro 0x80 é a força 128 enviada ao joystick

As dificuldades encontras no desenvolvimento deste driver foram relativas principalmente ao hardware da placa MBED. Primeiro foi necessário soldar dois resistores *pull-down* de 15KΩ nos pinos USB_D+ e USB_D- uma vez que a placa não vinha com estes resistores. Depois foi necessário configurar o pino USB_PWRD como GPIO, pois a placa não possui circuito externo para alimentar a USB e para detectar que um dispositivo foi inserido no barramento. Estas modificações não estão documentadas nem no manual da placa e nem no guia de referência do microcontrolador.

7.4 Implementação da rede Steer-by-Wire

O projeto da rede Steer-by-Wire foi implementado utilizando uma parte real e outra parte virtual. A parte real inclui o volante com force-feedback e as placas com a rede FTT-CAN, já a parte virtual é compreendida pela aplicação de visualização e controle de força das rodas. Esta decisão foi tomada para simplificar o desenvolvimento da parte mecânica do projeto.

Para interfaceamento com o joystick foi utilizada uma placa MBED com o microcontrolador LPC1768, visto que as placas M5282Lite não possuem portas USB Host. Esta placa e este interfaceamento foram apresentados nas Seção 7.3.2 e Seção 7.3.3, respectivamente.

A aplicação que simula as rodas foi desenvolvida usando a biblioteca gráfica GTK2, mas como esta biblioteca só suporta nativamente rotações em ângulos múltiplos de 90 graus, foi utilizada as funções da biblioteca **cairo** para realizar a parte de rotação.

A rotação de cada roda em torno do próprio eixo foi realizada com as seguintes funções do cairo:

```
cairo_translate (cr, 200, 200);
cairo_rotate (cr, angle);
cairo_translate (cr, -200, -200);
```

O computador que executa a aplicação de simulação das rodas recebe a posição destas através da porta serial (conversor UBS/Serial) */dev/ttyUSB0*. As posições são enviadas pelo nó de controle da roda que funciona numa placa M5282Lite. O conversor USB/Serial é ligado na serial desta placa para realizar a aquisição das posições.

Na aplicação de simulação das rodas também há um componente `slider` usado para atuar sobre o volante force-feedback. Isto permite simular forças externas atuando sobre as rodas. A tela da aplicação de simulação das rodas é apresentada na Figura 31.

A rede Steer-by-Wire desenvolvida é composta por 3 nós principais: 1) volante; 2) central de controle; 3) rodas e um nó auxiliar 4) setas. Inicialmente apenas o nó central de controle precisa ser ligado para que a rede comece a funcionar. Este nó será responsável pela detecção e inserção dos demais nós na rede.

O diagrama funcional do componentes principais da rede Steer-by-Wire pode ser visto na Figura 32.

A placa M5282Lite do nó Volante comunica com o volante force-feedback Momo Racing através da serial da placa MBED, uma vez que a M5282Lite não possui a função de USB Host. Neste caso a placa MBED atua como uma ponte entre o Momo Racing e a placa M5282Lite. Esta comunicação é bidirecional, uma vez que o nó Volante pode ler a posição do joystick e atual sobre o controle de forças deste.

Quando os nós Volante e Rodas são adicionados na rede, o nó mestre deverá coordenar o envio de mensagens destas e o sistema Steer-by-Wire entrará em funcionamento. O nó

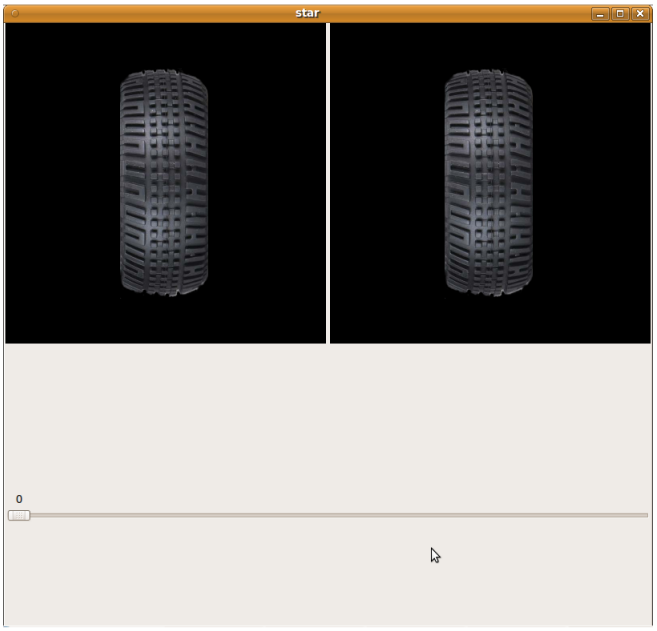


Figura 31: Tela do simulador das rodas

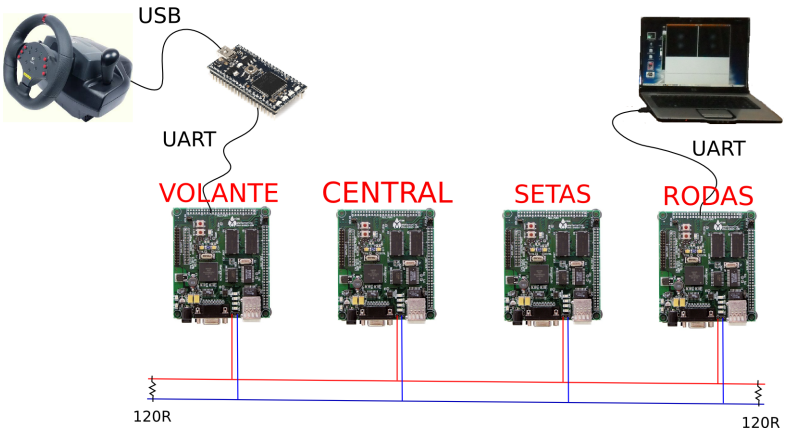


Figura 32: Diagrama da Rede Steer-by-Wire

volante começa a ler a posição do ângulo de rotação do joystick através da serial da placa MBED e transmitir estes valores através da rede FTT-CAN. Então o nó responsável pelas rodas irá ler estes valores transmitidos na rede pelo nó volante e transmitir, através da serial da placa M5282Lite, para a aplicação de visualização que está em execução no computador.

Através da aplicação de visualização das rodas o usuário poderá também atuar sobre o force-feedback do joystick, para isso basta mover o componente slider da aplicação para esquerda ou direita para que o volante se mova na mesma direção.

A Tabela 12 lista as mensagens que são transmitidas por cada nó na rede Steer-by-Wire:

Tabela 12: Mensagens transmitidas na rede

Nó	**Nome da Msg**	**Tipo**	**Período (ms)**
Volante	SteerWheelAngle	Sincrona	5
Volante	Dir_Set	Assincrona	-
Roda	WheelForce	Sincrona	5
Central	TM	Sincrona	5

Todos os nós transmitem suas mensagens sincronizados com o início de cada ciclo elementar (EC) usando como base de sincronismo a Trigger Message (TM) conforme explicado na Seção 4.5.

A foto com a rede Steer-by-Wire ligada com os 4 nós pode ser vista na Figura 33.

Figura 33: Rede Steer-by-Wire com os quatro nós ligados

7.5 Validação do Sistema de Inserção Dinâmica de Nós

A validação do sistema de inserção dinâmica de nós da rede FTT-CAN foi realizado ligando inicialmente apenas o nó mestre no barramento, em seguida as demais placas foram inseridas no barramento. Foi utilizado o CANanalyzer da fabricante IXX AT para obter a amostragem das mensagens sendo transmitidas na rede.

Em geral o sistema de inserção dinâmica de nodos comportou-se de forma esperada, com o novo nó sendo inserido corretamente. Na Figura 34 é apresentada a inserção do nó responsável por controlar o Volante, que transmite a mensagem SteerWheelAngle, usando o protocolo de *hot-plugging* apresentado neste trabalho.

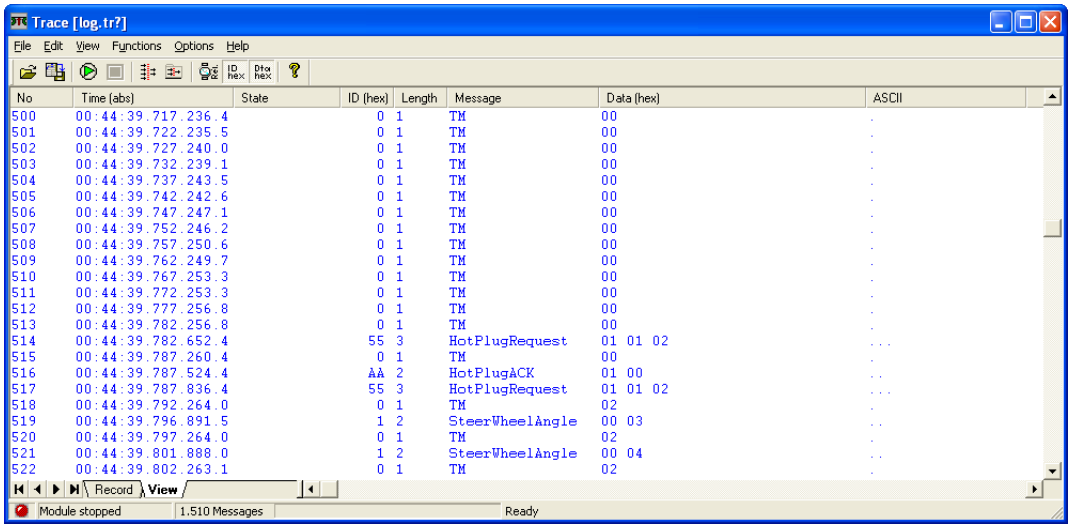


Figura 34: Inserção da Mensagem SteerWheelAngle na rede FTT-CAN

Como pode ser visto a mensagem 0x55 que é usada para solicitar a inserção do nó na rede FTT-CAN acabou sendo transmitida duas vezes. Isto acontece porque no momento que o nó escravo está recebendo a mensagem de ACK (0xAA) ele já disparou o envio da nova mensagem assíncrona 0xAA.

Nos testes iniciais isso gerava um estado de inconsistência na rede, onde uma sucessão de transmissões 0x55 e 0xAA, pois no momento em que o nó mestre respondia à mensagem 0x55 outra mensagem 0x55 já havia sido transmitida pelo nó escravo.

Foram feitas várias tentativas de remover a mensagem do *buffer* do controlador CAN quando a mensagem 0xAA fosse detectada, porém elas não tiveram efeito. Provavelmente seria necessária uma solução mais drástica, como tentar executar um *reset* no controlador imediatamente após detectar o recebimento da mensagem 0xAA.

Porém este tipo de solução seria complicado de implementar, pois o controlador precisaria voltar rapidamente para o estado posterior ao envio da mensagem 0x55 e assim se manter sincronizado com a Trigger Message.

Felizmente este problema foi resolvido de uma forma mais simples. O nó mestre detecta que a nova mensagem possui o mesmo conteúdo que a mensagem anterior e ignora esta nova requisição.

Esta solução não resolve a questão do envio do segundo 0x55, mas pelo menos evita que o nó escravo e mestre entrem num estado inconsistente enviado inúmeras mensagens 0x55 e 0xAA.

Algumas vezes a inserção do novo nó gerou problemas de erro lógico na rede, estes erros impediram o funcionamento correto do sistema de *hot-plugging* do nó escravo da

rede. Um destes problemas pode ser visto na Figura 35.

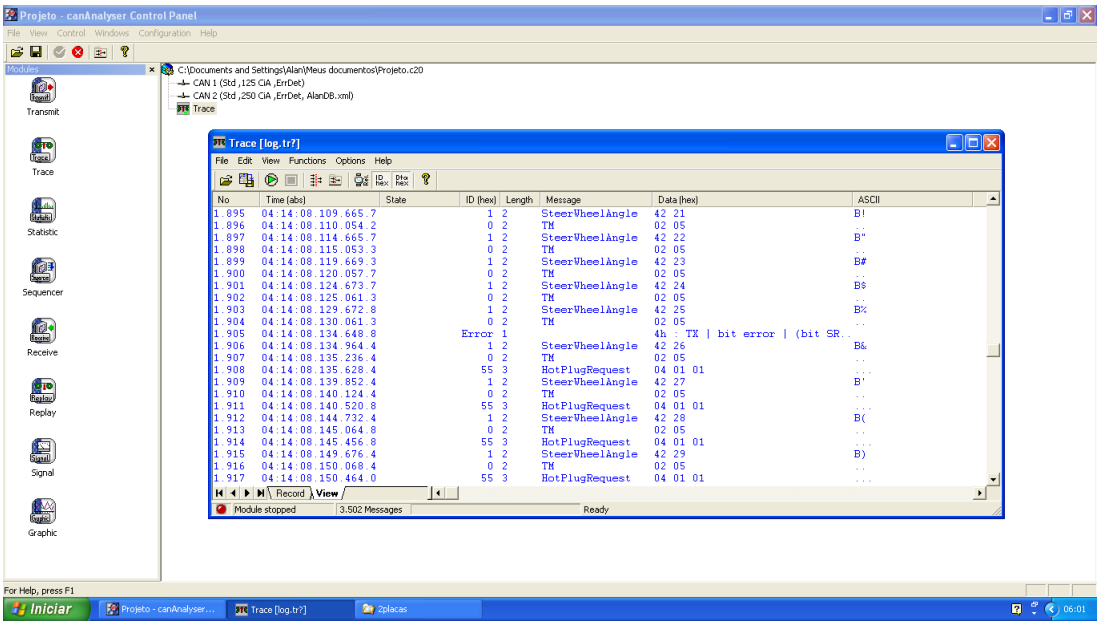


Figura 35: Erro durante a inserção do nó na rede FTT-CAN

Embora este problema não seja tão frequente (ele ocorreu em menos de 5% dos testes), ele serve para indicar que este sistema de inserção dinâmica de nodos não poderia ser utilizado numa aplicação *hard real-time*, onde uma falha colocaria em risco a vida de uma pessoa.

Mais analises precisam ser realizadas para descobrir se esta falha é um problema isolado deste *transceiver* ou se ocorrerá em todos os demais modelos existentes no mercado.

Ainda sim, este fato não impede que a rede FTT-CAN e o sistema de *hot-plugging* sejam utilizados em aplicações de tempo real que não tenham tais restrições de criticidade.

7.5.1 Medidas dos tempos de transmissão da rede

A tabela com os tempos de transmissão mínimo, máximo e o jitter de transmissão das mensagens com período de 5ms podem ser visto na Tabela 13.

Tabela 13: Tabela dos tempos de transmissão da rede a 5ms

Nó	Nome da Msg	Quant Bytes	Tempo Mínimo (us)	Tempo Máximo (us)	Jitter (us)
Volante	SteerWheelAngle	2	4987	5016	29
Volante	Dir_Seta	1	N/A	N/A	N/A
Roda	WheelForce	1	4995	5008	13
Central	TM	1	4992	5005	13

Como pode ser visto na Tabela 13 o maior Jitter apresentado foi de 29 microsegundos.

Os histogramas para as mensagens TM, WheelForce, SteerWheelAngle, podem ser vistos nas Figuras 36, 37, 38, nesta ordem.

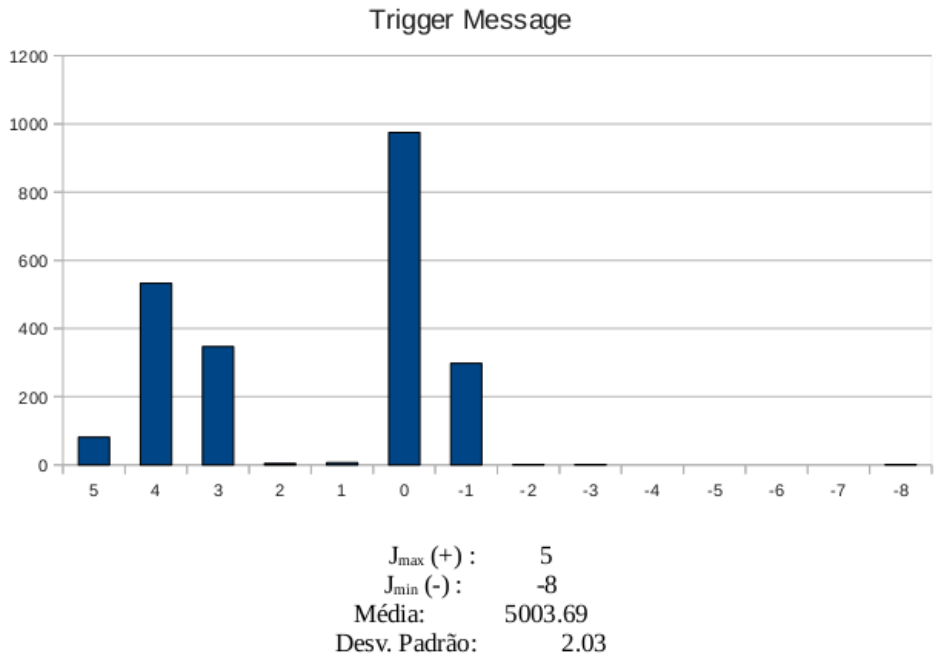
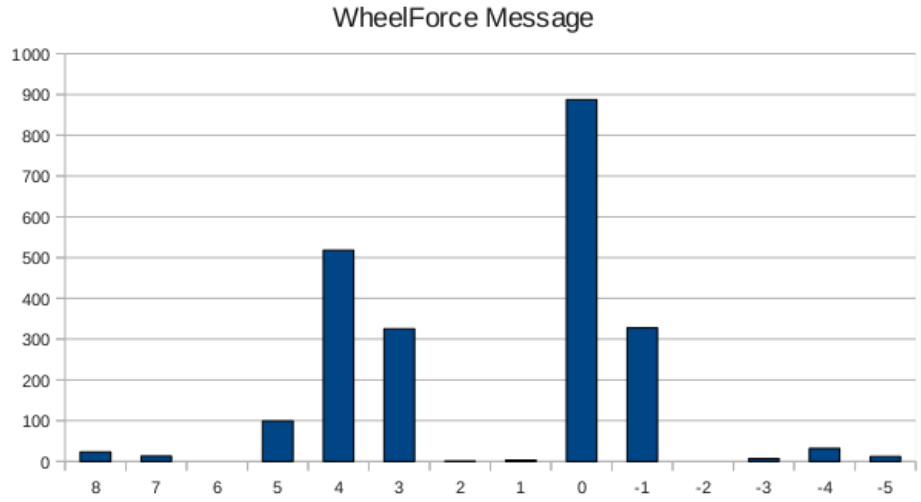
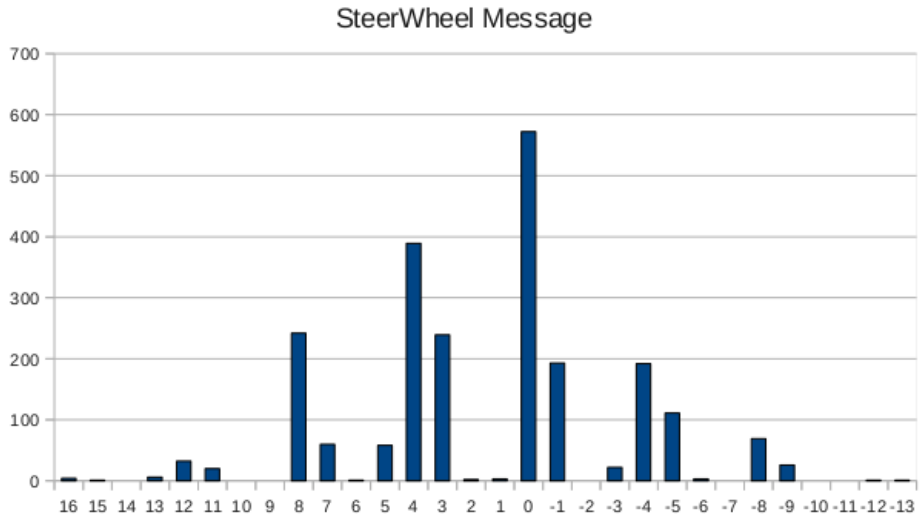


Figura 36: Histograma dos Jitters da TM à 5ms



J_{max} (+) :	8
J_{min} (-) :	-5
Média:	5003.69
Desv. Padrão:	2.33

Figura 37: Histograma dos Jitters da WheelForce à 5ms



J_{max} (+) :	16
J_{min} (-) :	-13
Média:	5001.46
Desv. Padrão:	4.52

Figura 38: Histograma dos Jitters da SteerWheelAngle à 5ms

A tabela com os tempos de transmissão mínimo, máximo e o jitter de transmissão das mensagens com período de 10ms podem ser visto na Tabela 14. Observe que a TM continua a ser transmitida a cada 5ms.

Tabela 14: Tabela dos tempos de transmissão da rede a 10ms

Nó	**Nome da Msg**	**Quant Bytes**	**Tempo Mínimo (us)**	**Tempo Máximo (us)**	**Jitter (us)**
Volante	SteerWheelAngle	2	9984	10016	32
Volante	Dir_Seta	1	N/A	N/A	N/A
Roda	WheelForce	1	9994	10008	14
Central	TM	1	4992	5008	16

Como pode ser visto na Tabela 14 o maior jitter apresentado foi 32 microsegundos.

Os histogramas para as mensagens TM, WheelForce, SteerWheelAngle, podem ser vistos nas Figuras 39, 40, 41, nesta ordem.

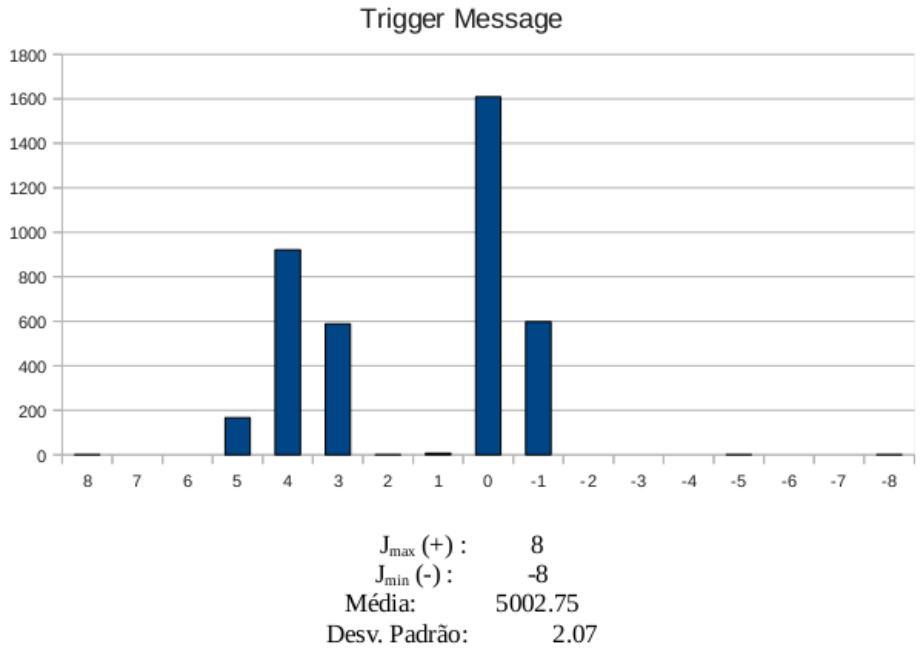
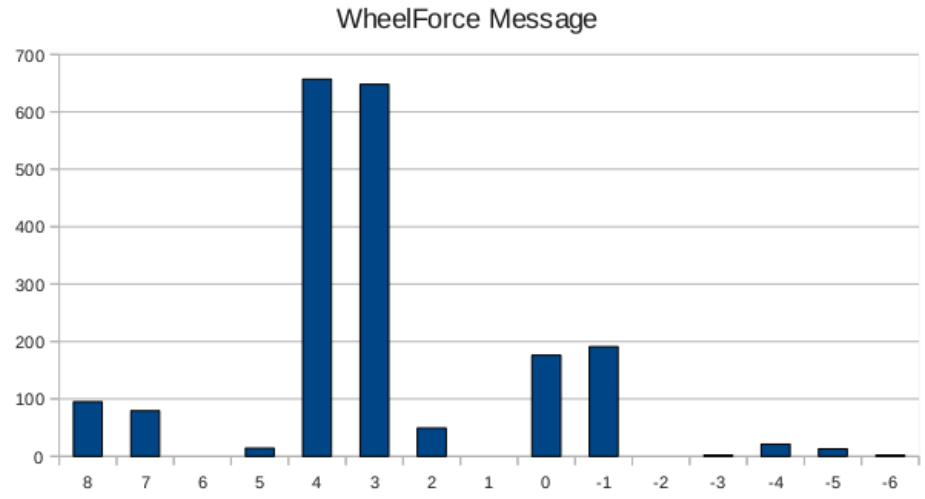
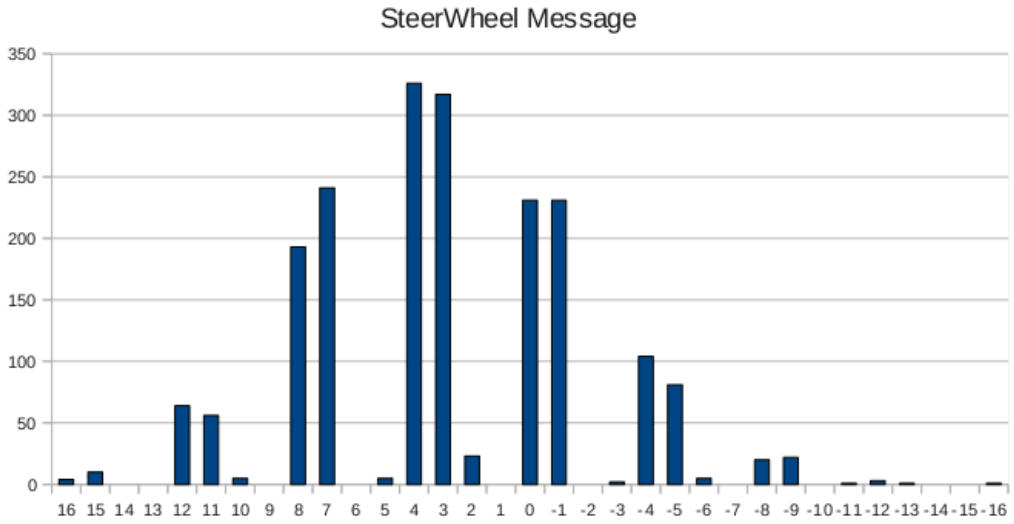


Figura 39: Histograma dos Jitters da TM à 5ms



J_{max} (+) : 8
J_{min} (-) : -6
Média: 10008.07
Desv. Padrão: 2.39

Figura 40: Histograma dos Jitters da WheelForce à 10ms



J_{max} (+) : 16
J_{min} (-) : -16
Média: 10008.06
Desv. Padrão: 4.74

Figura 41: Histograma dos Jitters da SteerWheelAngle à 10ms

A tabela com os tempos de transmissão mínimo, máximo e o jitter de transmissão das mensagens com período de 20ms podem ser visto na Tabela 15. Observe que a TM continua a ser transmitida a cada 5ms.

Tabela 15: Tabela dos tempos de transmissão da rede a 20ms

Nó	**Nome da Msg**	**Quant Bytes**	**Tempo Mínimo (us)**	**Tempo Máximo (us)**	**Jitter (us)**
Volante	SteerWheelAngle	2	19987	20019	32
Volante	Dir_Seta	1	N/A	N/A	N/A
Roda	WheelForce	1	19993	20012	19
Central	TM	1	4991	5010	19

Como pode ser visto na Tabela 15 o maior jitter apresentado foi 32 microsegundos.

Os histogramas para as mensagens TM, WheelForce, SteerWheelAngle, podem ser vistos nas Figuras 42, 43, 44, nesta ordem.

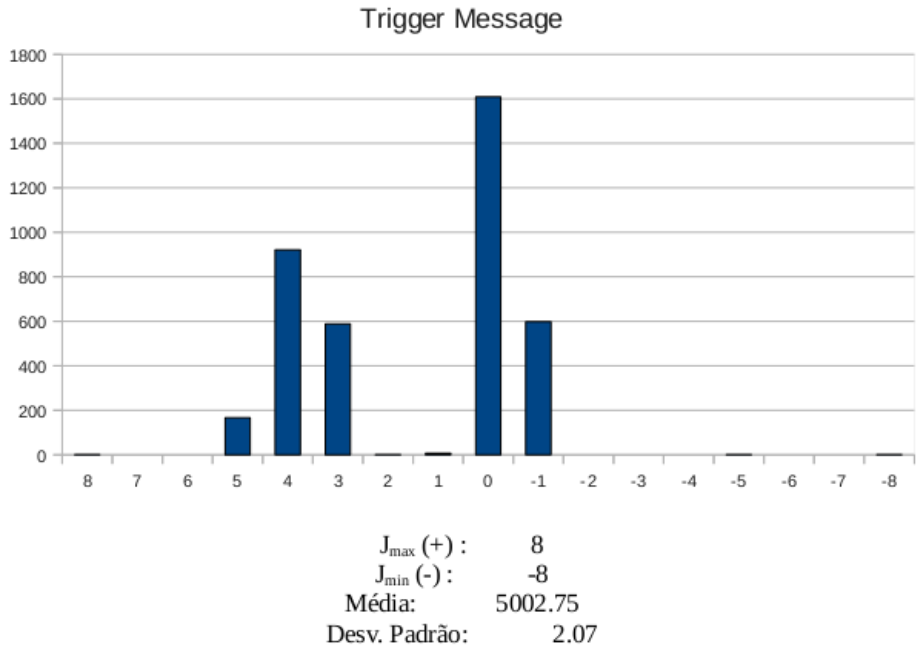
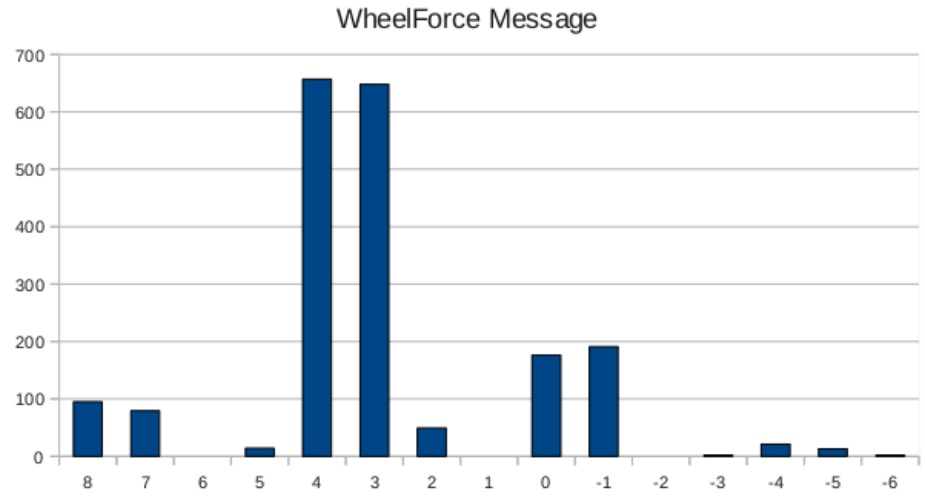
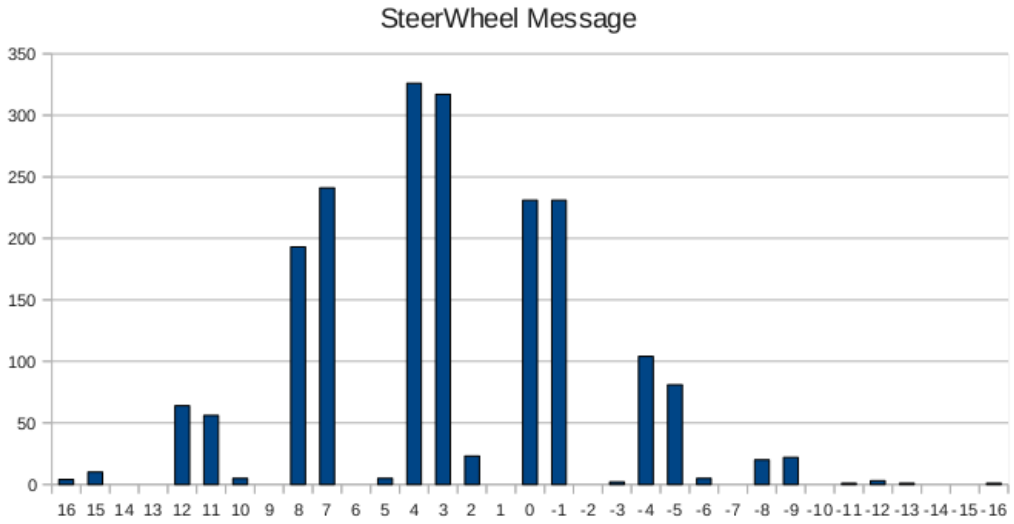


Figura 42: Histograma dos Jitters da TM à 5ms



J_{max} (+) : 8
J_{min} (-) : -6
Média: 10008.07
Desv. Padrão: 2.39

Figura 43: Histograma dos Jitters da WheelForce à 20ms



J_{max} (+) : 16
J_{min} (-) : -16
Média: 10008.06
Desv. Padrão: 4.74

Figura 44: Histograma dos Jitters da SteerWheelAngle à 20ms

Analisando os tempos das transmissões das mensagens observa-se que a mudança do período teve pouca influência sobre o *Jitter* das mensagens. A maior fonte de *Jitter* foi o tamanho da mensagem.

Portanto para evitar o aumento do *Jitter* nas mensagens de tempo real transmitidas pelo FTT-CAN deve-se dar preferência às mensagens com poucos bytes.

8 CONCLUSÃO

O padrão AUTOSAR trará uma flexibilidade nunca antes alcançada pela indústria automotiva. Fabricantes poderão trabalhar de forma colaborativa, cada uma desenvolvendo parte de um projeto e depois poderão integrar sem modificações as partes desenvolvidas obtendo um sistema funcional.

Para o consumir final os resultados deste consórcio também serão benéficos, pois este poderá usar em seu veículo equipamentos desenvolvidos por fabricantes de outras marcas, uma vez que, em princípio, eles irão interoperar.

O AUTOSAR ainda possui problemas, que deverão ser solucionados com o tempo, como por exemplo a não existência de um parâmetro para definição de um *deadline* para execução das tarefas (RAJNAK; KUMAR, 2007), a especificação prevê apenas o uso do pior caso para a execução de uma tarefa (WCET[1]).

Com base no estudo apresentado neste trabalho, o autor acredita que a proposta de uma solução automotiva baseada no AUTOSAR, que combine a flexibilidade e garantias FTT-CAN com um sistema de *hot-plugging* para a rede, será de grande interesse para a indústria automotiva, permitindo o desenvolvimento de ECUs com maior reusabilidade e com características *plug-and-play*.

O fato das redes automotivas atuais serem baseadas no protocolo CAN torna esta solução altamente interessante, uma vez que as fabricantes não precisarão investir em outro tipo de tecnologia para ter os benefícios de uma rede de tempo real com garantias temporais e flexibilidade oferecidos pelo protocolo FTT-CAN.

Uma dica para outros pesquisadores que queiram continuar o desenvolvimento deste trabalho é a utilização o RTOS Open Source Artic Core (www.arccore.com) que implementa as API do padrão Autosar. Infelizmente este sistema não existia quando este trabalho de mestrado foi iniciado. Com certeza usando o Artic Core grande parte do trabalho seria simplificado sem a necessidade de “reinventar a roda”.

[1] Worst Case Execution Time

REFERÊNCIAS

ALBERT, A. Comparison of Event-Triggered and Time-Triggered Concepts with Regard to Distributed Control Systems. **Embedded World 2004**, [S.l.], n.17, p.235– 252, Feb. 2004.

ALBERT, A.; GERTH, W. Evaluation and Comparison of the Real-Time performance of CAN and TTCAN. In: CAN IN AUTOMATION CONFERENCE, ICC, 9., 2003, Munich. **Anais...** [S.l.: s.n.], 2003. p.05–08.

ALMEIDA, L.; FONSECA, J. A.; FONSECA, P. Flexible Time-Triggered Communication on a Controller Area Network. **RTSS'98**, [S.l.], v.35, n.1, 1998.

ALMEIDA, L.; FONSECA, P.; FONSECA, J. A.; MAMMERI, Z. Scheduling and Clock Synchronization in CAN-based Distributed Systems. In: INTERNATIONAL CAN CONFERENCE, 6., 1999. **Anais...** [S.l.: s.n.], 1999.

ALMEIDA, L.; PASADAS, R.; FONSECA, J. Using the planning scheduler to improve flexibility in real-time fieldbus network. **IFAC - Iternational Federation of Automatic Control**, [S.l.], v.7, 2001.

ALMEIDA, L.; PEDREIRAS, P.; FONSECA, J. The FTT-CAN Protocol - Why and How. **IEEE Transactions on Industrial Electronics**, [S.l.], v.49, n.6, p.1189– 1201, Dec. 2002.

ATAIDE, F. H. **Proposta De Melhoria De Tempo De Resposta Para O Protocolo Ftt-Can - Estudo De Caso Em AplicaÃ§Ã£o Automotiva**. 2010. Dissertação (Mestrado em Ciência da Computação) — Universidade Federal do Rio Grande do Sul.

BABAOGLU, O.; MARZULLO, K. **Consistent Global States of Distributed Systems**: fundamental concepts and mechanisms. [S.l.: s.n.], 1993.

BECHENNEC, J.-L.; BRIDAY, M.; FAUCOU, S.; TRINQUET, Y. Trampoline An Open Source Implementation of the OSEK/VDX RTOS Specification. In: ETFA, 2006. **Anais...** IEEE, 2006. p.62–69.

BERWANGER, J.; SCHEDL, A.; TEMPLE, C. **Get ready as FlexRay hits the road**. http://www.eetasia.com/ART_8800459095_590626_NT_6372564f.HTM.

BIRRELL, A.; NELSON, B. Implementing remote procedure calls. **ACM Transactions on Computer Systems (TOCS)**, [S.l.], n.2, p.39–59, Feb. 1984.

BOHM, P. **Introduction to FlexRay and TTA**. 2005.

BOSCH, R. **CAN specification version 2.0**. http://www.motsps.com/csic/techdata/refman/can2spec.pdf.

CALHA, M. J. B. **A Holistic Approach Towards Flexible Distributed Systems**. 2006. Tese (Doutorado em Ciência da Computação) — University of Aveiro, Aveiro, Portugal.

CiA. **CAN Physical Layer**. http://www.can-cia.org/fileadmin/cia/pdfs/technology/physical_layer.pdf.

CONSORTIUM, O. **OSEK/VDX Operating System**. [S.l.]: OSEK/VDX Consortium, Copyright 2005.

COOK, J. A.; KOLMANOVSKY, I. V.; MCNAMARA, D.; NELSON, E. C.; PRASAD, K. V. Control, Computing and Communications: technologies for the twenty-first century model t. In: IEEE, 2007. **Proceedings...** IEEE, 2007. v.95, p.334–355.

CORRIGAN, S. **Improved CAN network security with TIâs SN65HVD1050 transceiver**. http://focus.ti.com/lit/an/slyt249/slyt249.pdf.

CRISTIAN, F.; FETZER, C. The Timed Asynchronous Distributed System Model. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.10, n.6, 1999.

DENUTO, J. V.; EWBANK, S.; KLEJA, F.; A., C. **LIN Bus and its Potential for use in Distributed Multiplex Applications**. 2001.

DIRK, J. OSEK/VDX HISTORY AND STRUCTURE. **University of Karlsruhe**, [S.l.], p.214–217, Nov 1998.

ESKICIOGLU, M. R. A Comprehensive Bibliography of Distributed Shared Memory. **SIGOPS Oper. Syst. Rev.**, [S.l.], v.30, p.71–96, Jan. 1996.

ESTEVEZ, A.; PRIANO, F.; PÃ©REZ, M.; GONZÃ¡LEZ, J.; MORALES, D.; RODA, J. Different Strategies to Develop Distributed Objects Systems at University of La Laguna. In: HIGH PERFORMANCE COMPUTING AND NETWORKING, 2000. **Anais...** Springer Berlin / Heidelberg, 2000. p.551–554. (Lecture Notes in Computer Science).

FLEXRAY, C. **FlexRay Communications System Protocol Specification Version 2.0**. [S.l.]: FlexRay Consortium, 2004.

FONSECA, J.; MARTINS, E.; ALMEIDA, L.; PEDREIRAS, P.; NEVES, P. Flexible Time-Triggered Protocol for CAN New Scheduling and Dispatching Solutions. **???????**, [S.l.], n.?????, p.??, Jan. 2000.

FREESCALE. **CAN Bosch Controller Area Network (CAN) Version 2.0 Protocol Standard**. http://www.freescale.com/files/microcontrollers/doc/data_sheet/BCANPSV2.pdf.

JOHANSSON, K. H.; TORNGREN, M.; NIELSEN, L. Vehicle Applications of Controller Area Network. In: **Handbook of Networked and Embedded Control Systems**. [S.l.: s.n.], 2005. p.741–766.

KALINSKY, D. A Modern Standard for Super-Small Kernels. **Real-Time Magazine**, [S.l.], n.4, p.22–24, Jan. 1999.

KOPETZ, H. Real-time in distributed real time systems. **Proceedings of the 5th IFAC Workshop on Distributed Computer Control Systems**, Sabi, South Africa, 1983.

KOPETZ, H. A Comparison of CAN and TTP. **Annual Reviews in Control**, [S.l.], v.24, p.177–188, 2000.

LEEN, G.; HEFFERNAN, D. Expanding Automotive Electronic Systems. **IEEE Computer**, [S.l.], v.35, n.1, p.88–93, 2002.

MARTINS, E.; FONSECA, J. Improving flexibility and responsiveness in FTT-CAN with scheduling coprocessor. In: FET01 (IFAC CONF. FIELDBUS TECHNOLOGY), 2001. **Proceedings...** [S.l.: s.n.], 2001. p.61–67.

MELLIAR-SMITH, P. M.; MOSER, L. E.; AGRAWALA, V. Broadcast protocols for distributed systems. **IEEE Transactions on Parallel and Distributed Systems**, [S.l.], v.1, p.17–25, Feb. 1990.

NASA. **Force Feedback Joysticks**. http://technology.arc.nasa.gov/success/successfiles/ForceFeed.pdf.

NAVET N.; SONG, Y. S.-L. F. W. C. Trends in Automotive Communication Systems. **Proceedings of the IEEE**, [S.l.], v.93, n.6, p.1204–1223, June 2005.

NILSSON, J. **Real-Time Control Systems with Delays**. 1998. Tese (Doutorado em Ciência da Computação) — Department of Automatic Control, Lund Institute of Technology, Lund, Sweden.

NOLTE, T.; HANSSON, H.; BELLO, L. L. **Implementing Next Generation Automotive Communications**. www-users.cs.york.ac.uk/ neil/ERTSI/camera/nohalo-ertsi-cr.pdf.

PAPERS, S. T. **Automotive Open System Architecture-An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E Architectures**. [S.l.]: SAE International, 2004.

PAPERS, S. T. **Achievements and Exploitation of the AUTOSAR Development Partnership**. [S.l.]: SAE International, 2006.

PARTNERSHIP, A. D. **Methodology**. [S.l.]: AUTOSAR Consortium, 2006.

PEDREIRAS, P. B. R. **Supporting Flexible Real-Time Communication on Distributed Systems**. 2003. Tese (Doutorado em Ciência da Computação) — University of Aveiro, Aveiro, Portugal.

PELZ, G.; OEHLER, P.; FOURGEAU, E.; GRIMM, C. **Advances in Design and Specification Languages for SoCs**. [S.l.]: Kluwer Academic Publishers, 2005. p.293–305.

RAJNAK, A.; KUMAR, A. Computer-aided architecture design & optimized implementation of distributed automotive EE systems. In: DAC '07: PROCEEDINGS OF THE 44TH ANNUAL CONFERENCE ON DESIGN AUTOMATION, 2007, New York, NY, USA. **Anais...** ACM Press, 2007. p.556–561.

RICHARDS, P. **A CAN Physical Layer discussion**. ww1.microchip.com/downloads/en/AppNotes/00228a.pdf.

RUFF, M. Evolution of local interconnect network (LIN) solutions. **Vehicular Technology Conference, 2003. VTC 2003-Fall. 2003 IEEE 58th**, [S.l.], v.5, p.3382–3389 Vol.5, Oct. 2003.

SANGORRÃN, D.; GONZÃ¡LEZ, M.; PÃ©REZ, H.; GUTIÃ©RREZ, J. J. Managing Transactions in Flexible Distributed Real-Time Systems. In: ADA-EUROPE, 2010. **Anais...** [S.l.: s.n.], 2010. p.251–264.

SCHREINER, D.; SCHORDAN, M.; GÖSCHKA, K. M. Component Based Middleware-Synthesis for AUTOSAR Basic Software. In: ISORC, 2009. **Anais...** IEEE Computer Society, 2009. p.236–243.

TAKADA, H.; NAKAMOTO, Y.; TAMARU, K. The ITRON Project: overview and recent results. In: INTL. CONFERENCE ON REAL-TIME COMPUTING SYSTEMS AND APPLICATIONS (RTCSA), 5., 1998. **Anais...** [S.l.: s.n.], 1998. p.3–10.

TANNENBAUM, A. **Distributed Systems Principles and Paradigms**. [S.l.]: Prentice Hall, 2002.

TINDELL, K.; BURNS, A.; WELLINGS, A. Calculating Controller Area Network (CAN) Message Response Times. **IEEE Computer**, [S.l.], v.35, n.1, 1996.

TORNGREN, M. A perspective to the Design of Distributed Real-time Control Applications based on CAN. , Munich, 2003.

TTA-GROUP. **Time-triggered protocol TTP/C, high-level specification document, Protocol Version 1.1**. 2003.

TTA-GROUP. **TTP Frequently Asked Questions**. 2004.

UPLAP, R.; OKHADE, M.; GHANEKAR, P.; NARAYAN, S.; SABLE, A. LIN Protocol - Technology Review and Demonstration in Power Window Application. **SAE International**, [S.l.], 2004.

VINCENTELLI, A. S. **Automotive Electronics**: trends and challenges. www-cad.eecs.berkeley.edu/Respep/Research/asves/paper2000/ASV_convergence00.pdf.

WILWERT, C.; NAVET, N.; SONG, Y.; SIMONOT-LION, F. **Design of automotive X-by-Wire systems**. [S.l.]: HAL - CCSd - CNRS, 2005.

YOO, S.; JERRAYA, A. A. Introduction to Hardware Abstraction Layers for SoC. In: DATE '03: PROCEEDINGS OF THE CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE, 2003, Washington, DC, USA. **Anais...** IEEE Computer Society, 2003. p.10336.

ZAHIR, A. OIL - OSEK Implementation Language. **IEE UK**, [S.l.], p.1–3, Jan. 1999.

APÊNDICE A ESPECIFICAÇÃO DO AUTOSAR

São apresentados a seguir os requisitos necessários aos sistemas operacionais compatíveis com o AUTOSAR OS.

A.1 Núcleo do Sistema Operacional

OS001: O Sistema Operacional deverá prover uma API compatível com o OSEK OS;

OS242: O Sistema Operacional não deverá permitir *callbacks* de Alarm nas classes de escalabilidade 2, 3 e 4;

OS301: O Sistema Operacional deverá possibilitar o incremento de um contador como uma ação alternativa à expiração do alarme;

OS304: Se em uma chamada SetRealAlarm o parametro "incremento" for definido como zero, o serviço deverá retornar `E_OS_VALUE` tanto no estado padrão como no extendido;

OS299: O Sistema Operacional deverá prover os serviços DisableAllInterrupt(), EnableAllInterrupt(), SuspendAllInterrupts(), ResumeAllInterrupts() antes da chamada StartOS() e depois da chamada ShutdownOS().

A.2 Tabelas de Escalonamento

OS002: O Sistema Operacional deverá incluir um mecanimo para processamento de cada ponto que expirou em uma tabela "por vez"(in turn);

OS007: O Sistema Operacional deverá permitir multiplas tabelas de escalonamento sejam processadas paralelamente;

OS005: O Sistema Operacional deverá prover um serviço para iniciar o processamento de uma tabela de escalonamento ao primeiro ponto de expiração (the first expiry point) no contador (O primeiro ponto de expiração deverá ser processado quando o valor do contador casar com o valor específicado pelo serviço);

OS347: O Sistema Operacional deverá prover um serviço para iniciar o processa-

mento de uma tabela de escalonamento ao primeiro ponto de expiração em um tempo relativo ao contador (O primeiro ponto de expiração deverá ser processado quando o valor do contador casar com o valor que o contador possuia no momento que o serviço foi chamado mais um valor específicado);

OS006: O Sistema Operacional deverá prover um serviço para cancelar o processamento da tabela de escalonamento imediatamente em qualquer ponto enquanto a tabela de escalonamento estiver em execução;

OS191: O Sistema Operacional deverá prover um serviço para troca de processamento de uma tabela de escalonamento para outra tabela de escalonamento (a troca é realizada ao final da tabela de escalonamento atual);

OS009: Se `PERIODIC = FALSE`, o Sistema Operacional deverá parar o processamento da tabela de escalonamento depois que o último ponto de expiração tenha ocorrido;

OS194: Se `PERIODIC = TRUE`, o Sistema Operacional deverá repetir o processamento da tabela de escalonamento do seu início, depois que o período tenha expirado;

OS012: O Sistema Operacional deverá permitir que o processamento da tabela de escalonamento seja controlado por um contador em software;

OS192: O Sistema Operacional deverá permitir que o processamento da tabela de escalonamento controlado por um contador em hardware;

A.3 Sincronização com tempo global

OS013: O Sistema Operacional deverá prover a habilidade para sincronização da tabela de escalonamento para uma base de tempo global;

OS199: O Sistema Operacional deverá prover serviços para serem chamados pelo usuário para prover o Sistema Operacional com o tempo global atual;

OS206: Se `SYNC_STRATEGY = SMOOTH` e um novo tempo global é provido, o Sistema Operacional deverá ajustar cada "span" de tempo entrem tempos de expiração adjacentes, seja incrementando ou decrementando;

OS200: Se `SYNC_STRATEGY = SMOOTH` e a sincronização da tabela de escalonamento do tempo global foi concluída, o Sistema Operacional deverá limitar o ajuste do tempo entre pontos de expiração adjacentes para uma faixa de valores configurada estaticamente (que é obtido por: `MAX_INCREASE` / `MAX_DECREASE` (OS310), se a tabela de escalonamento for síncrona ou `MAX_INCREASE_ASYNC` / `MAX_DECREASE_ASYNC`, se a tabela de escalonamento for assíncrona);

OS352: Se `SYNC_STRATEGY = HARD` e um novo tempo global é provido, o Sistema Operacional deverá ajustar o tempo ao final da tabela de escalonamento para o novo tempo global;

OS201: Se um serviço iniciar uma tabela de escalonamento configurada com `SYNC_STRATEGY = HARD` é chamada e o tempo global ainda não foi provido, o Sistema Operacional deverá iniciar esta tabela de escalonamento de forma síncrona depois que o tempo global for provido;

OS227: O Sistema Operacional deverá prover um serviço para consultar se uma tabela de escalonamento inicializada em modo assíncrono está sincronzada com um tempo global, deverá comparar com o limiar (*threshold*) `PRECISION` do OS310;

A.4 Suporte para Monitoramento da Pilha

OS067: Se uma Tarefa ou Rotina de Tratamento de Interrupção (ISR) Categoria 2 exceder seu uso da pilha, o Sistema Operacional deverá ser capaz de detectar isto como uma falha;

OS068: Se uma falha na pilha é detectada pelo monitoramento da pilha, o Sistema Operacional deverá chamar o serviço ShutdownOS() com o estado `E_OS_STACKFAULT`;

A.5 Aplicação-SO

OS016: O Sistema Operacional deverá prover serviços para determinar a Aplicação-SO em execução neste momento (um identificador único deverá ser alocado para cada aplicação);

OS017: O Sistema Operacional deverá prover um serviço para determinar a qual Aplicação-SO um dado Recurso, Tarefa, ISR, Contador, Alarme ou Tabela de Escalonamento pertence;

OS256: O Sistema Operacional deverá prover um serviço para determinar quais Aplicações-SO tem permissão para usar os IDs de uma Tarefa, ISR, Recurso, Contador, Alarm ou Tabela de Escalonamento nas chamadas da API;

OS258:O Sistema Operacional deverá prover um serviço para determinar a qual Aplicação-SO pertence um erro específico, gerado pela chamada da Task/Category 2 ou pela ISR/Aplicação;

A.6 Proteção dos Recursos

Proteção de Memória

OS198: O Sistema Operacional deverá previnir o acesso de escrita às suas próprias seções de dados e à sua própria pilha por outras Aplicações-SO não confiáveis;

OS026: O Sistema Operacional deverá previnir a tentativa de acesso de leitura por outras Aplicações-SO não confiáveis;

OS086: O Sistema Operacional deverá permitir que uma Aplicação-SO tenha acesso de leitura/escrita às suas próprias seções de dados privados;

OS207: O Sistema Operacional deverá previnir o acesso de escrita às seções de dados da Aplicação-SO por outras Aplicações-SO não confiáveis;

OS196: O Sistema Operacional deverá permitir a uma Task/Category 2 ISR o acesso de leitura/escrita à sua própria pilha privada;

OS208: O Sistema Operacional deverá previnir o acesso de escrita à pilha privada da Tasks/Category 2 ISRs de uma aplicação não confiável por toda as outras Tasks/ISRs na mesma Aplicação-SO;

OS355: O Sistema Operacional deverá previnir o acesso de escrita a todas a pilhas privadas das Tasks/Category 2 ISRs de uma Aplicação-SO por outras Aplicações-SO não confiáveis;

OS087: O Sistema Operacional deverá permitir a uma Task/Category 2 ISR o acesso de leitura/escrita às suas próprias seções de dados privados;

OS195: O Sistema Operacional deverá previnir o acesso de escrita às seções de dados privados da Task/Category 2 ISR de uma aplicação não confiável a todas as outras Tasks/ISRs na mesmoa Aplicação-SO;

OS356: O Sistema Operacional deverá previnir o acesso de escrita a todas as seções de dados privados da Task/Category 2 ISR de uma Aplicação-SO por outras Aplicações-SO não confiáveis;

OS027: O Sistema Operacional poderá prover a uma Aplicação-SO a habilidade de proteger suas próprias seções de código contra execução por Aplicações-SO não confiáveis;

OS081: O Sistema Operacional deverá prover a habilidade de prover biblioteca de código compartilhado às seções que são executáveis por todas as Aplicações-SO;

OS209: O Sistema Operacional deverá permitir o acesso, de leitura/escrita, ao espaço de periféricos mapeados em memória, pelas Aplicações-SO confiáveis;

OS083: O Sistema Operacional deverá prevenir a escrita, ao espaço de periféricos mapeados em memória, pelas Aplicações-SO não confiáveis (incluindo as leituras que tenham como efeito colateral a escrita a determinas areas de memória);

OS044: Se uma violação de acesso a memória é detectado, o Sistema Operacional deverá chamar o `Protection Hook` com o código de estado `E_OS_PROTECTION_MEMORY`;

Proteção Temporal

OS028: Em uma aplicação não confiável o Sistema Operacional deverá aplicar proteção temporal para todas Task/Category 2 ISR;

OS089: Se nenhuma Aplicação-SO é configurada OU se estiver executando uma Aplicação-SO confiável, o Sistema Operacional deverá ser capaz de aplicar proteção temporal para todas as Task/Category 2 ISRs.

OS210: Quando uma Task/Category 2 ISR alcanca seu tempo de execução estabelecido (*budget*), o Sistema Operacional deverá chamar o `Protection Hook` com `E_OS_PROTECTION_TIME`;

OS033: Se uma Task/Category 2 ISR carrega um Recurso do OSEK e excede o tempo de *Lock* do Recurso, o Sistema Operacional deverá chamar o `Protection Hook` with `E_OS_PROTECTION_LOCKED`;

OS037: Se uma Task/Category 2 ISR desativa uma interrupção e excede o tempo de `Lock` da Interrupção, o Sistema Operacional deverá chamar o `Protection Hook` com `E_OS_PROTECTION_LOCKED`;

OS048: O Sistema Operacional deverá limitar o número de ocorrências de interrupções dentro de uma linha do tempo (*timeframe*) configurada (se necessário desativando a fonte de interrupção e reativando as interrupções no próximo *timeframe*);

OS337: O Sistema Operacional deverá prover um serviço para desativar um fonte de interrupção específicada;

OS338: O Sistema Operacional deverá prover um serviço para requisitar a ativação de uma fonte de interrupção específicada;

OS064: Se o limite de tempo estabelecido (*budget*) de uma tarefa é alcancado o Sistema Operacional deverá chamar o `Protection Hook` com `E_OS_PROTECTION_RATE`;

Proteção dos Serviços

OS051: Se um endereço inválido (endereço que não pertence à aplicação) é passado ao um serviço do Sistema Operacional, o mesmo deverá retornar o código de status `E_OS_ILLEGAL_ADDRESS`;

Chamada de Serviço feita de um contexto errado

OS008: Se uma Aplicação-SO faz uma chamada de serviço de um contexto errado E não está dentro da Categoria 1 ISR, o Sistema Operacional não deverá executar a ação requisitada (a chamada de serviço não terá efeito) e deverá retornar `E_OS_CALLEVEL` ou um valor inválido do servico, a não ser que a chamada seja feita de dentro de uma Category 1 ISR;

Serviços com comportamento indefinido

Tarefas terminam sem chamar um TerminateTask() or ChainTask();

OS052: Se uma tarefa retorna de sua função de entrada sem executar a chamada TerminateTask() ou ChainTask(), o Sistema Operacional deverá terminar a tarefa (e chamar o `PostTaskHook` se configurado);

OS069: Se a tarefa retorna de sua função de entrada sem executar a chamada TerminateTask() ou ChainTask() E o `ErrorHook` é configurado, o Sistema Operacional deverá chamar a `ErrorHook` (isto é feito não importa se a tarefa causa outros erros, ex. `E_OS_RESOURCE`) com o status `E_OS_MISSINGEND` antes da tarefa deixar o estado `RUNNING`;

OS070: Se uma tarefa retorna da função de entrada sem executar a chamada TerminateTask() ou ChainTask() e continua carregando Recursos do OSEK, o Sistema Operacional deverá liberá-los;

OS239: Se uma tarefa retorna da função de entrada sem executar a chamada TerminateTask() ou ChainTask() e as interrupções continua desativadas, o Sistema Operacional deverá habilita-las;

Chamada PostTaskHook durante ShutdownOS()

OS071: Se a `PostTaskHost` está configurada, o Sistema Operacional não deverá chamar a *hook* se ShutdownOS for chamado;

Tasks/ISRs chamando EnableAllInterrupts/ResumeAllInterrupts/ResumeOSInterrupts sem a correspondente desativada

OS092: Se EnableAllInterrupts() / ResumeAllInterrupts() / ResumeOSInterrupts() forem chamadas e nenhuma correspondente DisableAllInterrupts() / SuspendAllInterrupts() / SuspendOSInterrupts() foi executada antes, o Sistema Operacional não deverá executar este serviço;

Tasks/ISRs chamando funções do Sistema Operacional quando DisableAllInterrupts/SuspendAllInterrup
foram chamadas

OS093: Se interrupções estão desativada e quaisquer serviços do SO, excluindo os serviços de interrupções, são chamados fora da rotina de *hook*, então o Sistema Operacional deverá retornar `E_OS_DISABLEDINT`;

Restrições de Serviços para Aplicações-SO não confiáveis

OS054: O Sistema Operacional deverá ignorar chamadas a ShutdownOS() por Aplicações-SO não confiáveis;

Chamadas de Serviços a Objetos em Diferentes Aplicações-SO

OS056: Se um identificador objeto-SO é o parâmetro de um serviço do sistema

e os direitos de acesso não tenham sido definidos em tempo de configuração para a Task/Category 2 ISR que esta chamando, o serviço do sistema deverá retornar `E_OS_ID`.

Protegendo o Hardware usado pelo SO

OS058: Se suportado pelo Hardware, o Sistema Operacional deverá executar as Aplicações-SO em modo não privilegiado;

OS096: Se suportado pelo hardware, o Sistema Operacional não deverá permitir que Aplicações-SO não confiáveis tenham acesso ao registradores de controle gerenciado pelo Sistema Operacional e o Sistema Operacional deverá restringir as Aplicações-SO confiáveis de acessar os registradores que não gerenciados por elas;

OS245: Se uma exceção de instrução ocorrer (ex. divisão por zero) o Sistema Operacional deverá chamar o *hook* de proteção com `E_OS_PROTECTION_EXCEPTION`;

Provendo "Funções Confiáveis"

OS097: O Sistema Operacional deverá prover um mecanismo para chamar um função confiável de uma Aplicação-SO (confiáveis ou não-confiável);

OS100: Se a função confiável chamada não está configurada, o Sistema Operacional deverá chamar o `ErrorHook` com `E_OS_SERVICEID`;

OS099: O Sistema Operacional deverá oferecer às Aplicações-SO um serviço para checar se uma região de memória é acessivel para escrita/leitura/execução por uma Task/Category 2 ISR e também retornar informações se a região de memória é parte do espaço da pilha;

A.7 Proteção de Erros

OS211: O Sistema Operacional deverá chamar a *Hook* de proteção com as mesmas permissões que o Sistema Operacional possui;

OS106: O Sistema Operacional deverá executar uma das seguintes reações dependendo do valor de retorno da *hook* de proteção:

- Matar a Task/Category 2 ISR falha OU
- Matar a Aplicação-SO falha OU
- Matar a Aplicação-SO falha e reinicia-la OU
- Chamar ShutdownOS();

OS217: Se a *Hook* de proteção não foi configurada e um erro de proteção ocorre, o Sistema Operacional deverá chamar ShutdownOS();

OS243: Se a reação é matar a Task/Category 2 ISR e nenhuma Task or ISR pode ser associada ao erro, a Aplicação-SO em execução deverá ser morta;

OS244: Se a reação é matar a Aplicação-SO falha e nenhuma Aplicação-SO pode ser associada ao erro, ShutdownOS deverá ser chamado;

OS108: Se o Sistema Operacional matar uma tarefa, a tarefa deverá terminar imediatamente (o `PostTaskHook` da tarefa não deverá ser chamado), todos os recursos alocados deverão ser liberados e chamar EnableAllInterrupts() / ResumeOSInterrupts() / ResumeAllInterrupts() se necessário;

OS109: Se o Sistema Operacional matar uma rotina de serviço de interrupcão, ele deverá limpar as requisições daquela interrupção, abortar a rotina de serviço de interrupção e liberar todos os recursos que a rotina de serviço de interrupção tenha alocado e chamar EnableAllInterrupts() / ResumeOSInterrupts() / ResumeAllInterrupts() se necessário;

OS110: Se o Sistema Operacional matar uma Aplicação-SO, ele deverá matar todas as Task/ISR associadas aquela aplicação, cancelar todos os alarmes, parar as tabelas de escalonamento e desativar as fontes de interrupção associadas aquela aplicação;

OS315: Se o Sistema Operacional matar uma Aplicação-SO (ou Task/Category 2 ISR) as mensagens locais associadas permanecerão em seus estados atuais;

OS111: Quando o Sistema Operacional reinicia uma Aplicação-SO ele ativa o `RESTARTTASK` configurado;

A.8 Escalabilidade do Sistema

OS240: Se uma implementação de uma classe de escalabilidade mais baixa suporta características de classes mais altas então as interfaces para estas características precisam cumprir com a especificação da classe mais alta;

OS241: O Sistema Operacional deverá suportar os recursos de acordo com a classe de escalabilidade configurada;

OS327: O Sistema Operacional deverá usar sempre modo extendido nas classes de escalabilidade 3 e 4;

A.9 Funções de Hook

OS060: Se uma *hook* de inicialização (*startup hook*) específica da aplicação é configurada para uma Aplicação-SO `<App>`, o Sistema Operacional deverá chamar `StartupHook_<App>` na inicialização do SO;

OS226: O Sistema Operacional deverá executar uma *hook* de inicialização específica da aplicação com os direitos de acesso da Aplicação-SO associada;

OS236: Se ambas, *hooks* de inicialização específica do sistema e específica da aplicação estão configuradas, o Sistema Operacional deverá chamar a *hook* de inicialização específica do sistemas antes de chamar a *hook* de inicialização específica da aplicação;

ShutdownHook

OS112: Se uma *hook* de encerramento (*shutdown hook*) é configurada para uma Aplicação-SO `<App>`, o Sistema Operacional deverá chamar `ShutdownHook_<App>` no encerramento do SO;

OS225: O Sistema Operacional deverá executar uma *hook* de encerramento específica da aplicação com os direitos de acesso do Aplicação-SO associada;

OS237: Se ambas, uma *hook* de encerramento específica do sistema e específica da aplicação, estão configuradas, o Sistema Operacional deverá chamar a *hook* de encerramento específica do sistema depois da *hook* de encerramento específica da aplicação;

Error Hook

OS246: Quando um erro ocorre E uma *hook* de erro específica da aplicação está configurada para Aplicação-SO `<App>` falha, o Sistema Operacional deverá chamar a *hook* de error `ErrorHook_<App>` específica da aplicação depois da *hook* de erro específica do sistema será chamada;

OS085: O Sistema Operacional deverá executar uma *hook* de erro específica da aplicação com os mesmos direitos de acesso da Task/Category 2 ISR que resultou na *hook* de erro sendo chamada;

A.10 Especificação da Configuração

OS113: A linguagem de configuração deverá permitir a seleção de um processador;

OS329: Os arquivos de configuração usados deverão usar a versao "3.0"da linguagem OIL;

Objeto do Sistema: `OS`

OS214: A linguagem de configuração deverá permitir ao usuário definir no máximo uma *hook* de proteção (`BOOLEAN PROTECTIONHOOK`);

OS259: A linguagem de configuração deverá permitir a seleção da classe de escalabilidade. Este atributo é usado para verificação cruzada (*cross checking*) e usa o mecanismo OIL AUTO (atributo no Objeto do Sistema » OS: ENUM WITH_AUTO [SC1,SC2,SC3,SC4] SCALABILITYCLASS = AUTO);

OS307: A linguagem de configuração deverá permitir selecionar se o monitoramento

da pilha será ativado ou não (`BOOLEAN STACKMINITORING`). O monitoramento da pilha aplica-se às Tasks e Category 2 ISRs.

Objeto do Sistema: `Application`

OS114: A linguagem de configuração deverá permitir ao usuário definir uma ou mais Aplicações-SO, até o número máximo especifico da implementação;

OS254: A linguagem de configuração deverá permitir ao usuário definir funções confiáveis disponíveis para outras Aplicações-SO;

Configurando Aplicações Confiáveis

OS115: A linguagem de configuração deverá prover ao usuário a habilidade de definir cada Aplicação-SO como sendo confiável;

OS124: A linguagem de configuração deverá permitir ao usuário definir ao máximo uma Startup-Hook para uma Aplicação-SO;

OS125: A linguagem de configuração deverá permitir ao usuário definir ao máximo uma Shutdown-Hook para uma Aplicação-SO;

OS213: A linguagem de configuração deverá permitir ao usuário definir ao máximo uma *Error-Hook* para uma Aplicação-SO;

Tarefa de Reinício

OS120: A linguagem de configuração deverá permitir ao usuário definir opcionalmente uma tarefa de uma Aplicação-SO as *Re-start Task* da mesma Aplicação-SO;

Objetos do SO

OS116: A linguagem de configuração deverá prover ao usuário a habilidade de associar várias tarefas para exatamente uma Aplicação-SO;

OS221: A linguagem de configuração deverá prover ao usuário a habilidade de associar várias ISRs a exatamente uma Aplicação-SO;

OS231: A linguagem de configuração deverá prover ao usuário a habilidade de associar vários Alarmes a exatamente uma Aplicação-SO;

OS230: A linguagem de configuração deverá prover ao usuário a habilidade de associar vários escalas (*schedules*) a exatamente uma Aplicação-SO;

OS234: A linguagem de configuração deverá prover ao usuário a habilidade de associar vários contadores a exatamente uma Aplicação-SO;

OS248: A linguagem de configuração deverá prover ao usuário a habilidade de asso-

ciar vários recursos a exatamente uma Aplicação-SO;

OS253: A linguagem de configuração deverá prover ao usuário a habilidade de associar várias mensagens a exatamente uma Aplicação-SO;

Objeto do Sistema `SCHEDULETABLE`

OS141: A linguagem de configuração deverá permitir ao usuário definir várias escalas (*schedules*);

OS145: A linguagem de configuração deverá permitir ao usuário definir o contador que controlará a tabela de escalonamento;

OS143: A linguagem de configuração deverá permitir ao usuário definir as ações ("ativar uma tarefa" ou "definir um evento" com `OFFSET` ticks/nanosegundos relativo ao início do período) de uma escala (*schedule*);

OS144: A linguagem de configuração deverá permitir ao usuário definir o PERIODO em ticks/nanosegundos de uma tabela de escalonamento (*schedule table*);

OS249: A linguagem de configuração deverá permitir ao usuário definir qual Aplicação-SO tem acesso à tabela de escalonamento;

OS310: A linguagem de configuração deverá permitir ao usuário definir a sincronização da tabela de escalonamento;

OS335: A linguagem de configuração deverá permitir ao usuário definir se uma tabela de escalonamento deverá ser iniciada automaticamente com um deslocamento relativo ao início do sistema operacional;

Objeto do Sistema: `TASK`

OS119: A linguagem de configuração deverá permitir ao usuário definir um pior caso do tempo de execução, em nanosegundos, para cada tarefa;

OS185: A linguagem de configuração deverá permitir ao usuário definir o tempo máximo permitido (*budget*) para cada tarefa;

OS188: A linguagem de configuração deverá permitir ao usuário definir um tempo de bloqueio de uma interrupção (*Interrupt Lock Time*) e/ou um tempo de bloqueio de um recurso (*Resource Lock Time*) em nanosegundos para um recurso do OSEK;

OS325: A linguagem de configuração deverá agrupar todos parâmetros relacionados à proteção temporal de uma tarefa (OS119, OS185, OS188) em um atributos do tipo ENUM;

OS250: A linguagem de configuração deverá permitir ao usuário definir quais Aplicações-SO terão acesso a tarefa;

Objeto do Sistema `ALARM`

OS251: A linguagem de configuração deverá permitir ao usuário definir quais Aplicações-SO terão acesso a alarme;

OS302: A linguagem de configuração deverá permitir ao usuário definir incrementar um contador como ação quando um alarme expirar;

Objeto do Sistema `RESOURCE`

OS252: A linguagem de configuração deverá permitir ao usuário definir quais Aplicações-SO terão acesso a determinado recurso;

Objeto do Sistema `COUNTER`

OS317: A linguagem de configuração deverá permitir ao usuário definir quais Aplicações-SO terão acesso a contador;

OS255: A linguagem de configuração deverá permitir ao usuário definir o tipo de um contador (SOFTWARE ou HARDWARE);

OS331: A linguagem de configuração deverá permitir ao usuário definir o tipo de unidade do contador (TICKS ou NANOSECONDS);

Objeto do Sistema `MESSAGE`

OS316: A linguagem de configuração deverá permitir ao usuário definir quais Aplicações-SO terão acesso a mensagem;

Objeto do Sistema: `ISR`

OS222: A linguagem de configuração deverá permitir ao usuário definir o pior caso do tempo de execução, em nanosegundos, para cada ISR;

OS223: A linguagem de configuração deverá permitir ao usuário definir a taxa máxima de chegada (em nanosegundos) para cada ISR;

OS229: A linguagem de configuração deverá permitir ao usuário definir um tempo de bloqueio da interrupção (*Interrupt Lock Time*) e/ou um tempo de bloqueio de recurso (*Resource Lock Time*) em nanosegundos de um recurso OSEK;

OS326: A linguagem de configuração deverá agrupar todos parâmetros relacionados à proteção temporal de uma ISR em um ENUM;

A.11 Geração do SO

OS172: O gerador deverá prover ao usuário a habilidade de ler a informação de um arquivo de configuração selecionável;

OS173: O gerador deverá prover ao usuário a habilidade de executar checagem de consistência da configuração atual;

OS050: Se serviços de proteção são requeridos e STATUS não é igual a EXTENDED (onde todos manipuladores de erro associados são providos), a checagem de consistência deverá exibir um erro;

OS045: Se características de proteção são configuradas junto com ISR Category 1 do OSEK, a checagem de consistência deverá exibir uma advertência;

OS320: Se os atributos configurados não casam com a classe de escalabilidade configurada (ex. definir um limite de execução de tempo em Tasks ou Category 2 ISRs e selecionar classe de escalabilidade 1) a checagem de consistência deverá exibir um erro;

OS175: Se o mesmo OS-object é associado a mais de uma Aplicação-SO, a checagem de consistência deverá exibir um erro;

OS311: Se uma Task ou Category 2 ISR não pertence a exatamente uma Aplicação-SO a checagem de consistência deverá exibir um erro;

OS177: Se uma fonte de interrupção que é usada pelo Sistema Operacional é associada a uma Aplicação-SO, a checagem de consistência deverá exibir um erro;

OS233: Se uma tarefa de reinicio é configurada e uma *hook* de proteção é desativada, a checagem de consistência deverá exibir um advertência;

OS303: Se `INCREMENTCOUNTER` é configurado como ação da expiração do alarme E o alarme é controlado diretamente ou indiretamente por este contador, a checagem de consistência deverá exibir uma advertência;

OS328: Se STATUS é `STANDARD` e `SCALABILITYCLASS` é SC3 ou SC4, a checagem de consistência deverá exibir um erro;

OS334: Se uma tabela de escalonamento não tem ponto de expiração no *offset* zero, a checagem de consistência deverá exibir uma advertência;

OS343: Se `SCALABILITYCLASS` é SC3 ou SC4 E uma tarefa é referenciada dentro de um objeto "tabela de escalonamento" E a Aplicação-SO da "tabela de escalonamento" não tem acesso a tarefa, a checagem de consistência deverá exibir um erro;

OS344: Se `SCALABILITYCLASS` é SC3 ou SC4 E uma tarefa é referenciada dentro de um objeto alarme E a Aplicação-SO do alarme não tem acesso a tarefa, a checagem de consistência deverá exibir um erro;

OS345: Se `SCALABILITYCLASS` é SC3 ou SC3 E uma tarefa é referenciada dentro de um objeto mensagem E a Aplicação-SO desta mensagem não tem acesso à tarefa, a checagem de consistência deverá exibir um erro;

Gerando o Sistema Operacional

OS179: Se a checagem de consistência do arquivo de leitura de entrada das configurações não está livre de erros, então o gerador não deverá gerar/configurar o sistema operacional;

OS336: O gerador deverá gerar a tabela de vetores de interrupção;

APÊNDICE B CÓDIGO FONTE DO STEER-BY-WIRE

O código fonte deste trabalho esta disponível na página do projeto no site SourceForge.

Nome do projeto no SourceForge: ftt2drive

Link: http://sourceforge.net/projects/ftt2drive