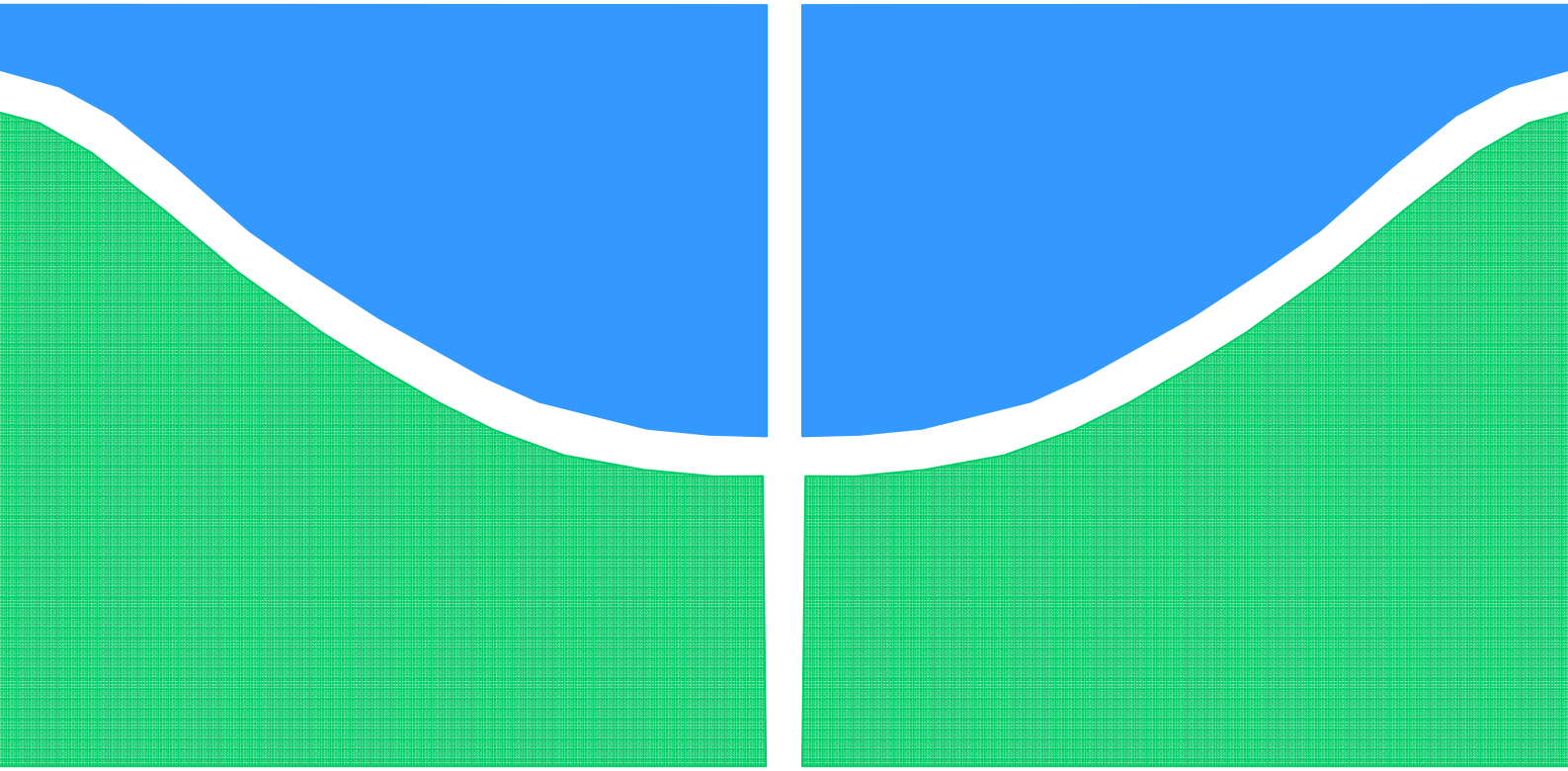

# TRABALHO DE GRADUAÇÃO

# SISTEMA DE AQUISIÇÃO MULTI-CAMADA BASEADO NA ARQUITETURA WOSA/XFS

**Diogo Caetano Garcia**
**Karen França de Oliveira**

**Brasília, agosto de 2006**

**UNIVERSIDADE DE BRASILIA**

FACULDADE DE TECNOLOGIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

# SISTEMA DE AQUISIÇÃO MULTI-CAMADA BASEADO NA ARQUITETURA WOSA/XFS

**Diogo Caetano Garcia**
**Karen França de Oliveira**

Relatório submetido como requisito parcial para obtenção
do grau de Engenheiro Eletricista

**Banca Examinadora**

Prof. Ícaro dos Santos, UnB/ Dep (Orientador) _____

Prof. Adson Ferreira da Rocha, UnB/ Dep _____

Prof.Ricardo Zelenovsky, UnB/ Dep _____

# DEDICATÓRIA

Aos meus avós e à minha querida madrinha Sônia (*in memorian*).

*Karen França de Oliveira*

À minha família.

*Diogo Caetano Garcia*

# AGRADECIMENTOS


A Deus, por todas as bênçãos e oportunidades maravilhosas que recebi.

Aos meus avós, pelo apoio incondicional que sempre me deram.

Aos meus pais e à minha irmã Rafaella, por todo o carinho e atenção a mim dedicados.

Á toda a minha família, pela paciência que tiveram em todas as minhas ausências.

A todos os meus amigos e ao meu namorado Nagao, que me acompanharam durante toda a minha formação.

Aos amigos de curso, especialmente Débora, Patrícia, Sumaia, Roques, Davi, Fernanda, Ronaldo e Bárbara, pelo companheirismo, nas noites de estudo e nos momentos de descontração.

Ao amigão Diogo, pela paciência e dedicação ao nosso trabalho.

*Karen França de Oliveira*

À minha família, especialmente aos meus pais, pela força e pela paciência.

Aos amigos de curso, especialmente Davi, Rafael, Lucas, Leandro, Ronaldo e Bárbara, por toda ajuda, companheirismo, risadas e pré-relatórios.

À companheira Karen, pela paciência e força (também).

Aos meus amigos de sempre, por tudo.

*Diogo Caetano Garcia*

Os autores agradecem de forma especial: ao professor orientador, Ícaro dos Santos, pela dedicação e companheirismo; a todos os outros professores, que tanto contribuíram para a nossa formação; aos amigos do laboratório GPDS, Ortis, por toda a sua paciência e disponibilidade, Edson Mintsu, por sempre apontar quando estava “tudo errado”, Tiago Alves, Rafael e Eduardo, pelas dicas e contribuições.

# SUMÁRIO

# LISTA DE ABREVIAÇÕES

| **Abreviação** | Significado |
|---|---|
| PC | **Personal Computer** |
| WOSA | **Windows Open Services Architecture** |
| API | **Application Programming Interface** |
| SPI | **Service Provider Interface** |
| A/D | **Analógico para digital** |
| ADC12 | **Conversor analógico para digital de 12 bits** |
| USART | **Universal Synchronous and Asynchronous Receive/Transmit Peripheral Interface** |
| JTAG | **Joint Test Action Group** |

# LISTA DE FIGURAS

# LISTA DE TABELAS

# SISTEMA DE AQUISIÇÃO MULTI-CAMADA BASEADO NA ARQUITETURA WOSA/XFS

## RESUMO

O presente trabalho de graduação apresenta a elaboração de um sistema multi-camada de aquisição de dados, em tempo real, baseado na arquitetura WOSA/XFS e implementado com o microcontrolador MSP430. Para validação do sistema, também foi desenvolvido um aplicativo de teste, capaz de exibir amostras dos sinais de interesse em *displays* gráficos, de acordo com a taxa de amostragem e o número total de amostras fornecidos pelo usuário.

# MULTI-LAYERED DATA ACQUISITION SYSTEM BASED ON THE WOSA/XFS ARCHITECTURE

## ABSTRACT

The actual work presents the elaboration of real-time, multi-layered data acquisition system based on the WOSA/XFS architecture, and implemented with the MSP430 microcontroller. For system validation, a test application was developed, in order to graphically display samples of the signals of interest, according to the sampling rate and the number of samples determined by the user.

# CAPÍTULO 1

# INTRODUÇÃO

## 1.1 OBJETIVOS DO TRABALHO

Este trabalho tem como objetivo apresentar um sistema de aquisição e medição de sinais implementado em camadas, conferindo modularização e, por conseqüência, flexibilidade na definição de cada um dos serviços envolvidos. Como referência, foi utilizada a arquitetura WOSA/XFS, definida por um conselho de fornecedores de soluções financeiras, o BSVC (*Banking Solution Vendors Council*) [8]. O principal objetivo desse conselho é promover uma especificação padronizada, sem ambigüidades, para os provedores de serviço e para os desenvolvedores dos aplicativos que se utilizam desses serviços. Essa arquitetura é amplamente aceita e difundida entre os principais desenvolvedores de *softwares* para automação bancária e comercial, constituindo um padrão completo e funcional.

O sistema completo é composto de diversos módulos, com a finalidade de captar amostras de um ou mais sinais analógicos, de acordo com a taxa definida pelo usuário, fornecendo-lhes os dados que os caracterizam. Para tal, utilizou-se o microcontrolador MSP430 para amostrar os sinais, enviando suas amostras a um PC (*Personal Computer*), segundo um protocolo definido via comunicação serial. Para que a comunicação pudesse ser feita em modo *half-duplex*, foi necessário o desenvolvimento de uma placa de hardware para constituir os barramentos, ajustando também os níveis de tensão envolvidos nessa comunicação.

O resultado de todo o processo pode ser visualizado em um aplicativo de interface gráfica, onde as amostras de um sinal de entrada são plotadas, com o intuito de permitir a análise em tempo real da forma de onda de interesse, bem como criar a possibilidade de armazenamento desses dados, caso seja necessário um posterior processamento do sinal.

## 1.2 MOTIVAÇÕES DA IMPLEMENTAÇÃO DE UM SISTEMA MULTI-CAMADA

O arquiteto de sistemas é um profissional responsável pela concepção, desenho e desenvolvimento da arquitetura de sistemas computacionais. Em sistemas grandes e complexos é impraticável que a arquitetura seja concebida de forma descentralizada por um grande número de desenvolvedores ou programadores. Dividindo-se todo o sistema em camadas, permite-se que os serviços oferecidos pelas camadas inferiores destinem-se a diversas funcionalidades, ou seja, a diferentes aplicações. Funcionalidades essas definidas pelos especialistas, conhecedores dos verdadeiros requerimentos dos usuários.

Nos laboratórios de pesquisa e desenvolvimentos das universidades, verifica-se grande renovação no grupo de alunos dedicados a uma mesma linha de pesquisa. Dessa forma, é bastante importante documentar a contribuição de cada um desses grupos, para que esta já sirva como ponto de partida para os próximos desenvolvedores. Além disso, existem vários projetos que, até uma determinada camada, utilizam recursos comuns que, se devidamente documentados permitiriam esse uso compartilhado sem a necessidade do desenvolvimento destas mesmas funcionalidades por cada um dos grupos em separado.

Em especial, no laboratório GPDS (Grupo de Processamento Digital de Sinais) na UnB (Universidade de Brasília), verifica-se grande interesse pela área de Instrumentação Biomédica, e o projeto em questão destina-se a implementar em várias camadas um sistema de medição e aquisição de dados. Além disso, nesse laboratório, introduziu-se uma nova área de pesquisa em Automação Bancária, que fundamentalmente também requer a implementação de um sistema de aquisição similar àquele da Instrumentação Biomédica. Isso é verificado, pois os dispositivos de automação bancária também devem receber e analisar as informações de um conjunto de sensores, além de acessar diferentes periféricos segundo a funcionalidade pretendida pelos seus usuários. Em suma, ambas as áreas de pesquisa requerem a definição de padrões de interface para a comunicação entre dispositivos de *hardware* e aplicativos de interface com o usuário. O DICOM [] é um padrão para comunicação e armazenamento de imagens médicas e informações associadas, que contém uma arquitetura para troca de informações e também para definição de protocolos de comunicação. Atualmente, esse padrão é utilizado em diversas modalidades de exames médicos. O presente trabalho oferece uma alternativa, implementando o sistema de aquisição a partir da arquitetura WOSA/XFS, que é

bem aceita e difundida entre os principais desenvolvedores de *softwares* para automação bancária e comercial. Além disso, essa arquitetura é mais ampla, não restringindo o objeto de sua funcionalidade.

O presente projeto final de graduação apresenta a elaboração de um sistema multi-camada de aquisição de dados, em tempo real, baseado na arquitetura WOSA/XFS e implementado com o microcontrolador MSP430. Para validação do sistema, também foi desenvolvido um aplicativo de teste, capaz de exibir amostras dos sinais de interesse em *displays* gráficos, de acordo com a taxa de amostragem e o número total de amostras fornecidos pelo usuário.

## 1.3 SISTEMAS DE MEDIÇÃO

O propósito de um sistema de medição é fornecer a um observador um valor numérico correspondente à variável que está sendo medida. Em geral, esse valor numérico, ou valor medido, não é igual ao valor real da variável. Em resumo, tem-se que a entrada do sistema de medição é o valor real da variável, enquanto a saída é o valor medido. Em um sistema de medição é possível identificar quatro tipos de elementos:

- **Elemento sensitivo**: está em contato com o processo e provê uma saída que depende, de alguma forma, da variável a ser medida. Um termistor, por exemplo, é um elemento desse tipo, cuja resistência varia de acordo com a temperatura.
- **Elemento de condicionamento do sinal:** converte a saída do elemento sensitivo em uma forma mais adequada ao processamento. Como exemplo, pode-se citar uma ponte que converte as variações de resistência em variações de tensão.
- **Elemento de processamento de sinal:** converte a saída do elemento de condicionamento em uma forma de apresentação mais adequada. Um exemplo é um conversor analógico-digital que converte valores de tensão em formato digital, adequado à manipulação computacional.
- **Elemento de apresentação dos dados**: apresenta o valor medido em uma forma que pode ser facilmente reconhecida pelo observador. Como exemplo, pode-se citar um simples display alfanumérico.

A partir dessa discriminação generalizada dos elementos constituintes de um sistema de medição, pode-se então particularizá-los para as aplicações de interesse: automação bancária e instrumentação biomédica.

### 1.3.1 AUTOMAÇÃO BANCÁRIA

A atividade de automação bancária compõe-se de equipamentos e sistemas informacionais que possibilitam que as transações entre os bancos, seus clientes, suas divisões internas e com outros bancos sejam feitas eletronicamente, visando: a redução de custos e a agilidade na tomada de decisão, para o banco, e o aumento da comodidade – e da individualização – para os clientes. Podem-se destacar três grupos na cadeia produtiva desse setor:

- **fornecedores de hardware**, inclusive dispositivos de armazenamento e servidores;
- **fornecedores de software** e soluções de gestão e gerenciamento da informação (ERP, CRM, integradoras, armazenamento de dados, Business Intelligence, Internet Banking);
- **empresas prestadoras de serviços** (algumas delas, como a Cobra, a CPM, e a Itautec Philco são controladas por bancos) que operam no outsourcing.

Os bancos brasileiros, historicamente, têm participado do desenvolvimento tecnológico do setor, inclusive como inovadores, além de desenvolverem diversas soluções internamente, principalmente as relacionadas à privacidade das informações. Essa proximidade entre usuário (banco) e fornecedor de tecnologia, constitui uma das fontes do progresso tecnológico do segmento no Brasil.

O segmento de automação bancária é, tradicionalmente, entendido como um subconjunto do setor de equipamentos e periféricos de informática. De fato, os caixas eletrônicos (ATMs - *Automated Teller Machines*) ainda são o aspecto mais visível, física e financeiramente, da automação bancária. Os ATMs são também os pioneiros da automação no Brasil, tendo sido introduzidos já na segunda metade da década de setenta. Contudo, tais equipamentos e periféricos estão se tornando, crescentemente, padronizados e, por isso, a terceirização dos serviços associados aos terminais de auto-atendimento tem sido uma tendência. Por esta razão, o processo de terceirização (*outsourcing* ou *Business Process*

*Outsourcing* - BPO) das atividades intensivas em trabalho, tem se constituído uma linha de negócios em expansão, explorada por grandes empresas como IBM, Unisys, HP, TecBan e Perto. Em seguida, será feito um estudo sobre esses dispositivos tão importantes para o setor em questão.

No grupo de *hardware*, são importantes aqueles fornecedores que estão diretamente relacionados com a infra-estrutura de gestão da informação, como os de dispositivos e soluções de armazenamento de dados e os integradores de sistema. Os dispensadores de cédulas, por sua vez, são o item de custo mais elevado (40%) de um ATM.

Os diversos *softwares* de automação das atividades bancárias, por outro lado, têm se tornado cada vez mais importantes, se dividindo em softwares de gestão ou de relacionamento com o cliente. Dessa forma, ganham importância os fornecedores de soluções que possam, a partir de uma plataforma estabelecida (IBM ou Microsoft, por exemplo), prover soluções personalizadas para as necessidades de cada banco ou usuário.

O desenvolvimento do segmento de automação bancária sempre esteve, no Brasil, relacionado ao ambiente regulatório. Nos anos recentes (2001-2002), a introdução do Sistema de Pagamentos Brasileiro (SPB), pelo Banco Central, provocou um reforço da automação bancária. Em 2001, por exemplo, onze bancos investiram cerca de R$ 75 milhões apenas para adequarem seus sistemas aos requerimentos do novo SPB. Torna-se então necessário aprofundar-se o conhecimento sobre o SPB para se entender a causa de tamanho investimento.

#### 1.3.1.1 *AUTOMATED CLEARING HOUSE* (ACH) E O SISTEMA DE PAGAMENTOS BRASILEIRO (SPB)

O sistema ACH surgiu nos Estados Unidos em 1972, em resposta ao rápido crescimento do volume de cheques na década de 60, garantindo a expansão do sistema. Esse sistema é responsável por mover eletronicamente fundos entre contas de bancos diferentes, através de uma rede. Substituindo-se o antigo sistema baseado em papel, reduziu-se o custo total das transações. Por exemplo, no pagamento de salários, o empregador origina a transferência em seu banco, de modo que seu empregado receba o salário. O banco reúne todas as transferências relevantes do empregador, e envia para o operador ACH uma mensagem eletrônica contendo todas as transferências efetuadas. O operador organiza

eletronicamente todas as transferências para os bancos correspondentes e calcula o crédito total a ser transferido. São então enviadas mensagens eletrônicas com uma lista de transferências a serem feitas em cada banco. No exemplo do pagamento de salário, o operador ACH envia uma mensagem de crédito para o banco do empregado e uma mensagem de débito para o banco do empregador. Desta maneira, o operador ACH libera e arranja as transações entre os bancos, eliminando relações bilaterais entre eles. Além disso, garante-se que todas as transferências diárias, com exceção das feitas com papel-moeda, sejam traduzidas em poucas transferências de fundos de alto valor entre bancos.

Já no Brasil, o responsável por mover fundos eletronicamente entre os bancos é o Banco Central. Juntos, os agentes não-bancários, os bancos e o Banco Central compõem o Sistema de Pagamentos Brasileiro. Sistema de Pagamentos é o conjunto de procedimentos, regras, instrumentos e sistemas operacionais integrados usados para transferir fundos do pagador para o recebedor, encerrando uma obrigação. As economias de mercado dependem desses sistemas para movimentar os fundos decorrentes da atividade econômica (produtiva, comercial e financeira), tanto em moeda local quanto estrangeira.

O SPB passou por várias reformas, sendo a mais importante a criação, em 2002, do Sistema de Transferência de Reservas (STR), que permite a liquidação em tempo real de transferências interbancárias de fundos. Com isso, são reduzidos os riscos de liqüidação (isto é, o risco de perda do valor total ou parcial de uma operação devido à falta de fundos e o risco de uma operação só ser liquidada após a data combinada).

Na configuração do novo SPB, a transferência de pagamentos deve ser online com as operações (dependendo da quantia envolvida ou da disposição de o recebedor de recursos pagar para ter seu depósito instantaneamente disponível). Dessa forma, caso se verifique alguma instituição com problemas de liquidez, é possível o cancelamento dos negócios, e o Banco Central passa a intervir de imediato.

O SPB é operacionalizado através da CIP (Câmara Interbancária de Pagamentos), autorizada pelo Banco Central a atuar no Brasil como uma *clearing house*, ou camada de compensação e de liquidação de títulos no SPB.

#### 1.3.1.2 AUTOMATED TELLER MACHINES (ATMS) – CAIXAS AUTOMÁTICOS

O inventor turco Luther George Simjian foi o primeiro a surgir com a idéia de uma máquina semelhante ao caixa automático, registrando a primeira de suas 20 patentes para a idéia em 1939.

Don Wetzel, na companhia dos engenheiros Tom Barnes e George Chastain, desenvolveu a versão moderna do caixa automático em 1968 (ao custo de 5 milhões de dólares), e patenteou a idéia em 1973. Nesse ano, já existiam 200 caixas automáticos operando nos EUA, vendidos ao preço de 30 mil dólares.

Atualmente, existem mais de 1,2 milhões de caixas automáticos instalados no mundo inteiro. A aproximadamente 5 minutos, um novo caixa automático é instalado no mundo. Uma pesquisa da empresa Harris Interactive indica que os ATMs são considerados tão importantes quanto o e-mail, e perdem apenas para o computador pessoal e o microondas em popularidade.

O sistema de ATMs permite ao cliente de um banco acessar sua conta remotamente através de uma máquina, que não necessariamente pertença ao banco. Por exemplo, o empregado que decide retirar seu salário (depositado através do Sistema de Pagamentos Brasileiro) acessa a máquina, que manda um sinal para um *switch* (comutador) ligado a todas instituições financeiras que fornecem cartões de ATMs. O caixa automático é então ligado eletronicamente ao banco do cliente, que atualiza automaticamente a sua conta e libera a transação para o caixa automático. Por fim, o cliente recebe o dinheiro.

Na maioria dos ATMs modernos, o cliente é identificado pelo cartão de plástico com uma fita magnética que contém o número da conta. É então requisitado que o cliente digite o seu PIN (*Personal Identification Number* – número de identificação pessoal) para complementar a identificação. Alguns ATMs utilizam *smartcards* para armazenar informações do cliente.

Os ATMs contêm criptoprocessadores seguros, geralmente compatíveis com o computador central. A segurança da máquina, na maior parte das vezes, consiste em manter intacto o criptoprocessador seguro, enquanto o software principal roda em um sistema

operacional comum. O ATM é tipicamente conectado ao processador ATM de transação por modem *dial-up* por uma linha telefônica ou diretamente via uma linha dedicada (*leased lines* – mais rápidas, porém mais caras). A cada dia, mais ATMs se conectam a Internet banda larga, que são mais baratas que as linhas dedicadas. Os ATMs mais novos com tecnologia Microsoft se utilizam do Windows XP ou o mesmo embarcado.

Os sistemas de ATMs podem ser classificados em proprietário, regional/compartilhado e nacional/internacional. O primeiro trata do caso em que a instituição financeira compra ou aluga caixas automáticos, adquire ou desenvolve o software necessário, instala e comercializa o sistema e fornece os cartões de uso nas máquinas. Um sistema de ATMs regional/compartilhado garante que clientes tenham acesso às suas contas através de máquinas operadas ou compradas por outras instituições financeiras. Um sistema de ATMs nacional/internacional é uma rede que liga máquinas do país inteiro e de outros países através dos sistemas de ATMs anteriormente citados.

O Banco do Brasil é um exemplo de instituição financeira que conta com os três tipos de sistemas, possuindo caixas automáticos próprios, caixas automáticos em conjunto com outros bancos (Rede Bancos Integrados e a rede Banco24Horas, que oferece serviços de 45 instituições financeiras), além de possuir caixas automáticos em Miami, Nova Iorque e Lisboa.

É importante notar que a simples instalação de um ATM é capaz de modificar o funcionamento da economia local. Como exemplo, nos Estados Unidos, que possuem cerca de 371 mil caixas automáticos instalados, foi observado que usuários de caixas automáticos gastam em média 20 a 25% a mais do que aqueles que não utilizam a máquina, e que máquinas que fornecem notas de 20 dólares aumentam as vendas das lojas ao redor dela em 8%. A partir desse fato curioso, pode-se também inferir a importância de um bom planejamento da distribuição desses caixas

A Figura 1 ilustra a constituição básica de um ATM . Nele estão presentes um monitor, um teclado, uma impressora de recibos (*receipt printer*), uma leitora de cartões (*card reader*), uma dispensadora de cédulas (*money dispenser*), dentre outros módulos. Em destaque, vê-se a disposição das cédulas de dinheiro em uma das caixas (*currency boxes*) do cofre. No momento de retirada, apenas uma nota de uma pilha de, aproximadamente, duas mil

notas, deve ser movimentada por vez até a área de dispensa, onde será então acumulado o total de notas requerido. Para tal, é necessário que um sensor determine, por meio da medição da espessura, por exemplo, se duas ou mais cédulas foram pegas. A verificação da integridade do cofre também pode ser feita por meio da medição de uma variável, a pressão, indicada por um sensor. Enfim, o ATM, um dos dispositivos de automação bancária mais visíveis, pode efetuar diversas operações e, em cada uma delas, torna-se necessário o envolvimento de processos de medição e tomada de decisões, como na situação de retirada descrita.

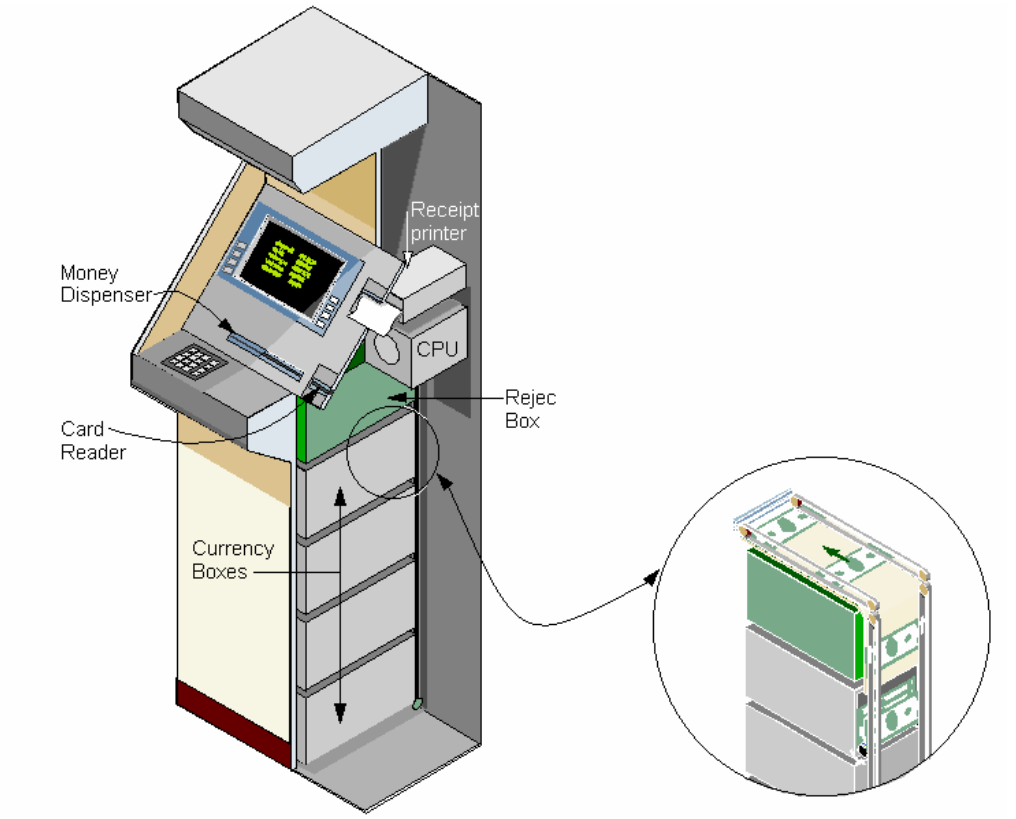


*Figura 1 – Ilustração em corte de um ATM, mostrando seus principais componentes. (modificado [17])*

### 1.3.2 INSTRUMENTAÇÃO BIOMÉDICA

Bioengenharia é a fusão de conhecimentos em engenharia e em ciências biológicas, para utilizá-los de forma mais eficaz em benefício do homem. Dentro deste conceito, podem-se distinguir pelo menos seis áreas:

- **Engenharia médica**: é a aplicação de engenharia à medicina destinada ao provimento de estruturas danificadas;
- **Engenharia Ambiental:** é a aplicação dos princípios de engenharia para controlar o ambiente, tornando-o saudável e seguro;
- **Engenharia da Agricultura:** é a aplicação dos princípios de engenharia à produção biológica e a operações externas que a influenciem;
- **Biônica:** é o estudo das funções e princípios de operação de sistemas vivos com a aplicação do conhecimento adquirido do *design* de sistemas físicos ;
- **Engenharia de Fermentação:** é a engenharia relacionada a sistemas biológicos microscópicos que são usados para sintetizar novos produtos;
- **Engenharia dos Fatores Humanos:** é a aplicação da engenharia, da fisiologia e da psicologia para a otimização do relacionamento homem-máquina.

A comunicação entre engenheiros, técnicos e doutores, instrumentação melhor e mais acurada para medir parâmetros fisiológicos vitais, e o desenvolvimento de ferramentas multidisciplinares para ajudar no combate a disfunções e doenças do organismo, tudo isso faz parte desse novo campo de estudo, chamado de forma geral de Engenharia Biomédica. Já aos métodos de medida relacionadas a esse campo, dá-se o nome de Instrumentação Biomédica.

A Instrumentação Biomédica é desenvolvida para medir e atuar sobre parâmetros médicos e fisiológicos. Devido ao progresso realizado no último século, tornou-se possível caracterizar os sistemas biológicos qualitativamente e quantitativamente, atuando, se necessário, em seus tecidos biológicos de forma seletiva. Além disso, viabilizou-se o diagnóstico de patologias desde os estágios iniciais, bem como o acompanhamento de sua evolução. Pode-se citar algumas características que diferenciam a instrumentação biomédica da instrumentação para outros fins. São elas:

- o sinal a ser medido (em geral) se origina de tecido vivo ou de energia aplicada a tecido vivo;
- faixa de freqüência dos sinais a serem medidos (faixa de áudio ou menor, inclusive com componentes DC);
- pequena amplitude dos sinais (em geral os sinais possuem amplitudes entre alguns microvolts e centenas de milivolts);
- fragilidade e complexidade das células ou tecidos;

- proteção contra choque elétricos, radiação e contaminação (macro e micro, tanto para o objeto de medida, quanto para o operador);
- caráter não crítico de exatidão (faixa de tolerância de um dado parâmetro devido à variabilidade entre indivíduos);
- dificuldade de acesso, em alguns casos, ao local de medida.

Em resumo, a Instrumentação Biomédica capta a variável, faz a transdução, amplifica, filtra, armazena, transmite e visualiza a variável medida, como ilustrado na Figura 2. Deve-se notar que, tanto para viabilizar a medição da variável de interesse, quanto para aplicar o tratamento necessário, é preciso interagir com os tecidos biológicos.

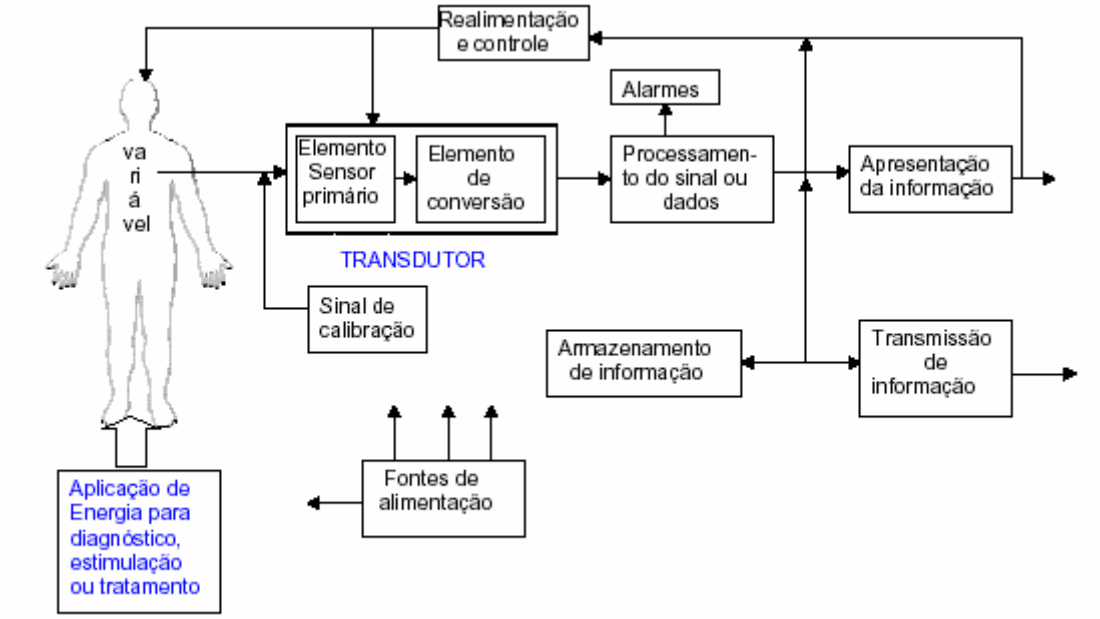


*Figura 2 – Modelo de um sistema genérico de interesse da Instrumentação Biomédica (modificado [7])*

Pode-se então, fazer uma breve descrição dos elementos constituintes desse sistema.

A **variável de medida** é a quantidade, condição ou propriedade física a ser medida pelo sistema de instrumentação. Os sinais que carregam a informação da medida podem ser analógicos, assumindo qualquer valor dentro de uma faixa dinâmica, ou digitais, assumindo um número finito de valores diferentes. As variáveis podem ser agrupadas em oito categorias:

- potencial bioelétrico (EEG, ECG, EOG, EMG, ERG)

- pressão (arterial, intraocular, intracraniana)
- deslocamento (velocidade, aceleração, força (musculatura)
- impedância elétrica (impedância transtoráxica)
- temperatura (corpórea, timpânica)
- concentrações químicas (gasometria, dosagem de hormônios)
- dimensões (circunferência craniana e toráxica de feto, para avaliar idade gestacional)
- Fluxo (de sangue com ultra-som Doppler; de urina – urodinâmica mede o fluxo de urina através da uretra para detectar possível estenose causada por hipertrofia de próstata; de ar dos pulmões).

O **transdutor** (elemento sensor primário e elemento de conversão) é um dispositivo que converte uma forma de energia em outra (geralmente elétrica), para fins de registro e processamento. Um transdutor ideal deve responder somente à energia da variável a ser medida, não alterando o seu estado, ou seja, não deve fornecer ou retirar energia do sistema.

No **processamento do sinal**, estão incluídos todos os tipos de dispositivos (equipamentos) eletrônicos, desde circuitos amplificadores até os computadores digitais. Os circuitos de condicionamento do sinal são usados para amplificar, filtrar, digitalizar, retificar, enfim, para tratar a informação tornando possível sua exibição ao usuário, sua transmissão ou, até mesmo, seu armazenamento. A meta do processamento de sinais biomédicos é extrair informações de um sinal biológico que melhorem ainda mais a compreensão dos mecanismos básicos da função biológica, auxiliando no diagnóstico ou tratamento de uma condição médica. Embora a maioria dos transdutores usados em instrumentação biomédica sejam analógicos (as grandezas biomédicas são analógicas), utilizam-se freqüentemente equipamentos digitais para o processamento dos dados. Isso exige a utilização de conversores A/D e D/A para interfaceamento dos transdutores analógicos com computadores digitais e destes com os *displays* analógicos. Uma importante questão no processamento de sinais biomédicos é a detecção e supressão de artefatos. Um artefato refere-se à parte do sinal produzida por eventos que são estranhos ao evento biológico. Esses artefatos surgem de muitas maneiras diferentes como, por exemplo:

- **Artefatos instrumentais**: gerados pelo uso de um instrumento. Um exemplo deste artefato é a interferência de 60Hz captada pelos instrumentos de gravação das tomadas da rede elétrica;

- **Artefatos biológicos:** nos quais um sinal biológico contamina ou interfere em outro. Um exemplo deste artefato é o deslocamento do potencial elétrico observado em um eletroencefalograma (EEG) devido à atividade cardíaca;
- **Artefatos de análise:** os quais podem surgir no decorrer do processamento do sinal biológico para produzir uma estimativa do evento de interesse. Por exemplo, os erros de arredondamento devidos à quantização de amostras de sinais, que surgem em função do processamento digital.

A partir do exposto, vê-se que os artefatos de análise são, de certa forma, controláveis. No exemplo citado, o erro inerente à quantização pode se tornar desprezível aumentando-se os intervalos de quantização. Já um método comum para reduzir os efeitos dos artefatos instrumentais e biológicos é a filtragem, desde que o sinal desejado e os artefatos que o contaminam ocupem faixas de freqüências que não se sobreponham.

A **apresentação dos dados** serve como interface entre o processamento e o operador. Pode ser um *display* alfa-numérico, o terminal de um computador, ou, até mesmo, uma impressora. Os dados podem ser apresentados de forma contínua, ou somente quando solicitados.

Os dados obtidos podem ser armazenados, transmitidos ou inseridos em uma malha de realimentação para permitir o controle do processo em análise. O armazenamento dos dados pode ser breve, durante o condicionamento dos sinais, ou para permitir que o operador do instrumento biomédico revise os dados. Os dados também podem ser armazenados antes de passarem pelo condicionamento, para serem submetidos a diferentes esquemas de processamento, que permitam a extração de novas informações. É importante notar que esses dados podem ser armazenados no próprio equipamento ou, até mesmo, serem transferidos para um banco de dados central. Nessa transferência de sinais e dados do paciente para centrais de processamento e/ou monitoração, pode-se utilizar comunicação por cabos, redes de computadores, fibras ópticas, ligação por rádio, dentre outros.

A malha de realimentação e controle é constituída por circuitos ou dispositivos utilizados para controlar a aquisição da informação (ajustar ganhos, faixa de freqüência ou níveis de energia), ou o seu direcionamento na cadeia de processamento. Sistemas complexos

podem incluir processamento computacional considerável para administrar a captação simultânea de sinais de vários pacientes.

A partir do disposto, pode-se então entender o ser humano como um conjunto de variáveis de interesse, que devem ser medidas, processadas e apresentadas de forma adequada, permitindo ao especialista uma análise completa da situação de seu paciente.

## 1.4 ORGANIZAÇÃO GERAL DO TRABALHO

O presente trabalho está dividido em seis capítulos, que estão dispostos da seguinte forma:

- **Capítulo 2:** Descreve o sistema total de aquisição a partir dos módulos de captura, tráfego e fornecimento de dados.

- **Capítulo 3:** Destina-se a fundamentar os requisitos de programação multi-camadas em ambiente Windows necessários à elaboração do software do Provedor de Serviços.

- **Capítulo 4:** Introduz o módulo de fornecimento de dados e a Arquitetura WOSA/XFS, padrão utilizado para a definição das camadas e da interface entre as mesmas, além de já descrever o desenvolvimento do Provedor de Serviços.

- **Capítulo 5:** Apresenta os módulos de tráfego e captura de dados, descrevendo separadamente a constituição e o funcionamento dos principais componentes de hardware do sistema, o microcontrolador MSP430 e a porta serial do PC. Elucida aspectos desse tipo de comunicação, apresentando também um circuito de compatibilização dos níveis de tensão desses dois dispositivos.

- **Capítulo 6:** Descreve o aplicativo desenvolvido para a validação do sistema em camadas implementado, fazendo análise crítica dos resultados obtidos.

- **Capítulo 7:** Apresenta conclusões e perspectivas sobre este trabalho.

# CAPÍTULO 2

# DESCRIÇÃO DO SISTEMA

## 2.1 CARACTERÍSTICAS GERAIS

O sistema de aquisição e medição de sinais desenvolvido é composto por vários módulos, que funcionam em conjunto para medir o sinal de interesse e garantir que os dados daí advindos possam ser disponibilizados para posterior análise, processamento e armazenamento. O sistema é formado por três módulos funcionais, implementados através de um PC, um microcontrolador MSP430 e um circuito de condicionamento para a comunicação entre ambos.

O módulo de *captura de dados* é composto por dois periféricos do microcontrolador: o conversor A/D de 12 *bits* (ADC12) e o contador/temporizador *Timer A*. Este módulo é responsável por fazer a medição do sinal condicionado; no caso, a tensão obtida a partir do elemento sensitivo utilizado. O ADC12 converte o sinal condicionado para o formato digital, mais adequado para a transmissão e o processamento, e o *Timer A* gera a taxa de conversão.

O módulo de *tráfego de dados* garante a troca de informações entre o PC e o microcontrolador. Através desse módulo, o PC pode configurar o módulo de captura de dados para as taxas de conversão e o número de amostras adequados, e pode receber os resultados da conversão. Fazem parte desse módulo a Interface Serial Universal USART0 do microcontrolador, a porta serial do PC e o circuito que garante a compatibilidade entre ambos, visto que cada um segue um padrão diferente.

Finalmente, o módulo de *fornecimento de dados* controla o módulo de tráfego de dados de forma adequada, garantindo a troca de informações com o módulo de captura de dados, e fornecendo o produto final: a medição do sinal condicionado em formato digital. Este módulo é composto através de um provedor de serviços e do Gerenciador WOSA/XFS, que são bibliotecas que seguem a arquitetura WOSA/XFS.

## 2.2 ESQUEMÁTICO DO SISTEMA

O sistema de aquisição caracterizado na sessão anterior é ilustrado na Figura 3. Nele, estão destacados os módulos de captura, tráfego e fornecimento de dados, e as setas indicam a direção do tráfego de informação.

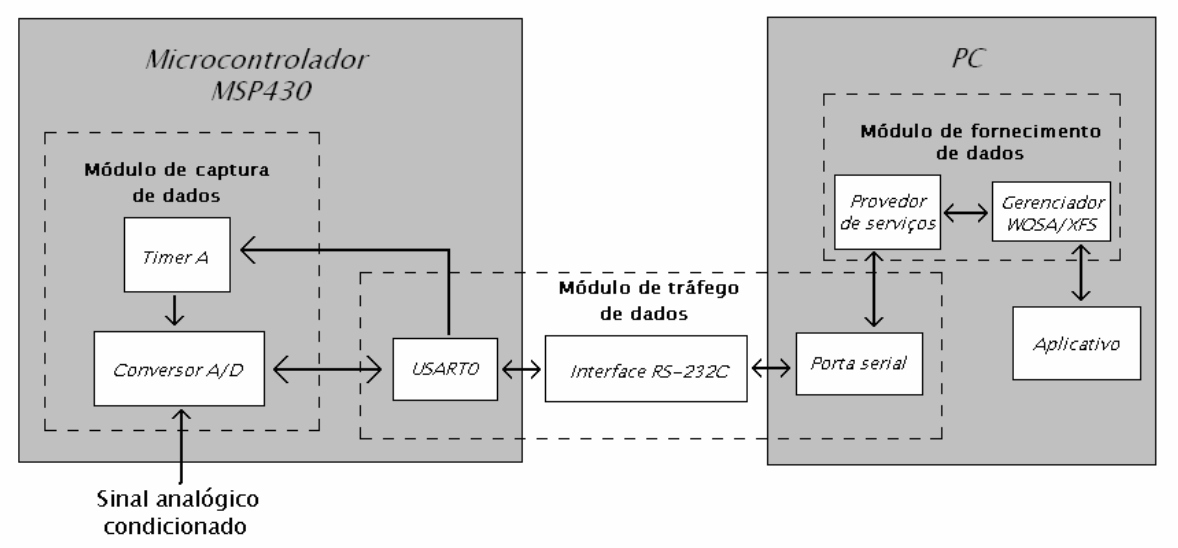


*Figura 3 – Esquemático geral do sistema*

A aquisição das amostras digitais (a partir do sinal analógico condicionado) só pode ser feita depois que o módulo de captura de dados tiver sido configurado adequadamente. Para isso, o aplicativo envia as taxas de amostragem para o Gerenciador WOSA/XFS, que então as envia para o provedor de serviços. Este realiza um processamento das taxas, e envia um pacote de configuração correspondente para o microcontrolador, através do módulo de tráfego de dados. É esse pacote que configura o módulo de captura de dados.

Em seguida, o aplicativo deve fazer o pedido de conversão, enviando o número total de amostras desejado para o Gerenciador, que as passa para o provedor de serviços, e assim por diante até chegar ao módulo de captura de dados. Quando o número total de amostras é obtido, o módulo de captura envia os resultados pelo caminho inverso, até chegarem ao aplicativo.

Neste projeto, o sistema descrito foi implementado em sua totalidade, e um aplicativo foi criado para sua validação. Dessa forma, foram desenvolvidos:

- A biblioteca do provedor de serviços, para a comunicação com o Gerenciador XFS e para o acesso à porta serial;
- A interface entre o provedor de serviços e o Gerenciador XFS;
- A interface RS-232C, para compatibilizar os padrões utilizados pelas portas seriais do PC e do microcontrolador MSP430;
- O *software* do microcontrolador MSP430, para configurar e utilizar o conversor A/D, o contador/temporizador *Timer A* e a interface serial USART0;
- O aplicativo de validação, para configurar o sistema e receber os resultados das conversões.

É importante ressaltar que o Gerenciador XFS pode ser obtido gratuitamente na Internet [9]. O código do mesmo é fechado, de forma que o desenvolvimento do provedor de serviços é feito com base nas referências dadas no *site*.

# CAPÍTULO 3

# FUNDAMENTOS DE PROGRAMAÇÃO MULTI-CAMADA EM AMBIENTE WINDOWS

Como já mencionado, o sistema de aquisição multi-camada implementado é baseado na arquitetura WOSA/XFS. Desta forma, para que possam ser atendidas as especificações das funções descritas nesse padrão, é necessário conhecer melhor o Sistema Operacional Windows, como gênero da classe Sistemas Operacionais, as funcionalidades e o controle por ele exercidos.

Sistema Operacional é um conjunto de ferramentas necessárias para que um computador possa ser utilizado de forma adequada, ou seja, como intermediário entre o aplicativo e a camada física do *hardware*. Este conjunto é constituído por um kernel, ou núcleo, e um conjunto de *softwares* básicos que executam operações simples. Se não houvesse sistemas desse tipo, todo *software* desenvolvido deveria saber se comunicar diretamente com os dispositivos do computador de que precisasse, já que esse sistema é responsável pelo funcionamento do computador, controle dos periféricos, execução de aplicativos, gerência de memória, rede, dentre outros serviços. Existindo um sistema operacional, para que o *software* se comunique com qualquer dispositivo, ele apenas chama uma função do kernel, deixando a comunicação com o dispositivo a cargo do sistema, que então repassará os resultados.

Cada sistema operacional pode ter uma maneira própria e distinta de comunicar-se com o *hardware*, razão pela qual é comum que *softwares* feitos para um sistema operacional não funcionem noutro, principalmente no caso de linguagens compiladas. Deve-se então direcionar os estudos para a plataforma Windows, que dará suporte ao sistema de aquisição.

## 3.1 SISTEMA OPERACIONAL WINDOWS

Um elemento chave da arquitetura WOSA/XFS, que será descrita posteriormente, é a definição de um conjunto de APIs, correspondentes a um outro conjunto de SPIs, e serviços

de suporte, provendo o acesso a serviços financeiros para aplicações baseadas em Windows. Para que se possa implementar todas as funcionalidades propostas por essa arquitetura, é necessário destacar algumas características desse sistema.

Todos os "objetos" com os quais o Windows trabalha possuem manipuladores (*handles*). Um *handle* é um tipo especial de ponteiro inteligente (*smart pointer*). Enquanto um ponteiro convencional contém o endereço do item ao qual faz referência, o *smart pointer* é uma referência abstrata atribuída por um outro sistema como, por exemplo, um sistema operacional. Essa opacidade permite que o item referenciado seja realocado na memória, sem invalidar o seu *handle*, o que não seria possível utilizando-se ponteiros convencionais. Estes objetos podem ser janelas, controles, menus, diálogos, processos, *threads*, áreas de memória, *displays*, impressoras, arquivos, *drives* de disco e até fontes, *brushes* e *pens* usados para desenhar e escrever. Um manipulador é um valor *dword* que pode ser requisitado pelo aplicativo. Uma vez obtido, este manipulador é utilizado pelo aplicativo para se comunicar com o Windows e solicitar seu uso ou modificações. A API do Windows usa *handles* para representar os objetos e para estabelecer um caminho de comunicação entre o próprio sistema operacional e o espaço do usuário. Por exemplo, uma janela no *desktop* é representada por um *handle* do tipo HWND.

Todos os dispositivos que mostram ou produzem uma saída possuem contextos de dispositivo. O contexto de dispositivo é uma área de memória, mantida pelo Windows, que contém informação sobre como o dispositivo deve mostrar sua saída. Portanto, uma janela em particular terá um contexto de dispositivo que conterá informações sobre a fonte e a cor que devem ser usadas para qualquer coisa que for desenhada ou escrita nesta janela. Uma impressora terá um contexto de dispositivo contendo as características da impressora, tamanho do papel, cores disponíveis e assim por diante.

O Windows assume o controle do computador praticamente desde o instante em que é iniciado até o momento em que é terminado. Uma aplicação pode rodar apenas com a permissão do Windows, com a assistência do Windows e sob o controle do Windows. Dentre os diferentes tipos de controle exercidos por esse sistema operacional, é de particular interesse aquele exercido sobre o *hardware* e sobre o processador, para que se entenda melhor o gerenciamento dos recursos entre os diferentes aplicativos.

### 3.1.1 O CONTROLE EXERCIDO SOBRE O HARDWARE

A maioria dos *microchips* que trabalham com a unidade central de processamento (CPU) são programáveis. Por exemplo, os chips de entrada/saída da impressora precisam conhecer a porta de comunicação a ser utilizada e a que velocidade os dados deverão ser transferidos. O Windows, como os demais sistemas operacionais, controla a comunicação com esses dispositivos, ou seja, atribui a cada um deles uma área de memória para que sejam informados sobre o que se espera que eles façam e quando.

Muitos programas de MS-DOS escreviam diretamente na memória de vídeo e na porta de impressora. A desvantagem desta técnica era a necessidade de prover *driver* de *software* para toda placa de vídeo e todo modelo de impressora. Windows introduziu uma camada de abstração chamada Interface de Dispositivo de Gráficos (GDI). O Windows provê o *driver* de vídeo, assim seu programa não precisa saber o tipo de placa de vídeo e impressora de seu sistema. Ao invés de acessar diretamente o *hardware*, seu programa chama funções da GDI que referencia uma estrutura de dados chamada contexto de dispositivo. O Windows mapeia a estrutura de contexto de dispositivo para um dispositivo físico que reconhece as instruções de entrada/saída apropriadas. A GDI é quase tão rápida quanto o acesso direto de vídeo, além de permite que diferentes aplicações escritas para Windows compartilhem a exibição na tela.
Em resumo, o Windows controla todo o *hardware* dos periféricos, impedindo o acesso direto a eles através de aplicativos.

### 3.1.2 O CONTROLE EXERCIDO SOBRE O PROCESSADOR

Ao iniciar um programa, o processador recebe do Windows o seu endereço inicial. O Windows pode parar o processador e solicitar que o mesmo rode um programa diferente por algum tempo, ou seja, fornece a outro programa uma fatia de tempo. Terminando o segundo programa, o Windows reconstitui os valores armazenados e continua executando o primeiro programa a partir do ponto de interrupção. È deste modo que o Windows oferece a possibilidade de rodar vários programas simultaneamente, constituindo-se um sistema multi-

tarefas. Cada um dos programas que estiver sendo executado recebe uma fatia de tempo do processador. O Windows faz a divisão do tempo de acordo com várias prioridades. Por exemplo, operações de leitura e escrita em disco possuem prioridades muito altas e podem bloquear a execução de outros programas até que sejam finalizadas.

De forma semelhante, um programa pode pedir ao Windows que inicie uma nova linha de execução (*thread* ou enlace). Neste caso, o Windows atribuirá a esta *thread* suas próprias fatias de tempo, seus próprios valores de registradores e sua própria pilha. A nova linha de execução parece estar sendo executada ao mesmo tempo que a *thread* principal do programa. Isto é chamado de programação multi-threading, sendo muito útil quando um programa precisar dar continuidade a uma determinada tarefa como, por exemplo, um cálculo muito longo, e, ao mesmo tempo, manter a interface do usuário ativa. Os enlaces de um mesmo processo ou, até mesmo, de processos diferentes podem fazer uso de determinados recursos, como arquivos e alocações dinâmicas de memória. Deve-se então gerir o acesso concorrente a recursos do sistema operacional, de maneira que um recurso não seja modificado simultaneamente, ou que os processos não fiquem esperando que um recurso seja liberado. A seguir, será feita uma breve descrição sobre processos, enlaces e sobre o grupo de mecanismos que permite a comunicação entre eles.

### 3.1.3 - CRIANDO E SINCRONIZANDO ENLACES MÚLTIPLOS

Os enlaces estão intimamente relacionados com processos. Um processo é um programa carregado na memória; é estático e não faz nada sozinho. Um enlace executa comandos de um programa, seguindo através do código. Opcionalmente, o enlace inicial (principal) pode criar outros enlaces. Todos os enlaces que pertencem a um processo compartilham todos os componentes desse processo. Todos seguem instruções da mesma imagem do código, referem-se às mesmas variáveis globais, escrevem no mesmo espaço de endereço privativo e têm acesso aos mesmos objetos.

Devem-se criar novos enlaces sempre que seu programa manipular atividade assíncrona. Um programa que interagisse com vários dispositivos assíncronos criaria enlaces para responder a cada dispositivo.

Qualquer enlace pode criar outros enlaces e também novos processos. Quando um programa precisa fazer várias coisas ao mesmo tempo, ele precisa decidir se deve criar enlaces ou processos para compartilhar o trabalho. Sempre que puder, devem-se criar enlaces porque o sistema os cria rapidamente e eles interagem facilmente entre si. Criar um processo leva mais tempo porque o sistema precisa carregar uma nova imagem do arquivo executável do disco, mas um novo processo tem a vantagem de receber seu próprio espaço de endereço privativo. Pode-se então escolher processos ao invés de enlaces como um modo de evitar que eles interfiram, mesmo acidentalmente, com os recursos uns dos outros.

No nível do sistema, um enlace é um objeto criado pelo gerenciador de objetos. Como todos os objetos do sistema, um enlace contém dados, ou atributos, e funções, ou métodos. A maioria dos métodos do enlace tem funções do Win32 correspondentes. Em outras palavras, a API Win32 exibe os métodos para os aplicativos do Win32.

Os programas que rodam no nível de usuário do sistema (em oposição ao nível kernel), não podem examinar ou modificar, diretamente, o interior de um objeto de sistema. Apenas chamando as rotinas da API Win32 você não pode fazer nada com um objeto. Este é protegido pelo Windows, que fornece uma *handle* para identificar o objeto e você a passa para funções que dela necessitam. Os enlaces também têm *handles*. A função que cria um enlace retorna uma *handle* para o novo objeto. Com a *handle*, você pode aumentar ou diminuir a prioridade de programação do enlace, dar-lhe uma pausa ou retomá-lo, finalizá-lo e descobrir que valor ele retornou ao terminar.

A interação efetiva entre os enlaces exige controle sobre a temporização. Esse controle assume duas formas: prioridade e sincronismo. A prioridade controla a freqüência com que um enlace obtém tempo do processador. O sincronismo regula os enlaces quando eles competem por recursos compartilhados e impõem uma seqüência quando vários deles devem executar tarefas em uma ordem determinada.

#### 3.1.3.1 PRIORIDADE

Quando o programador (*scheduler*) do sistema procura outro enlace para rodar, ele dá preferência aos enlaces de prioridade alta. Cada processo tem uma classificação de prioridade e os enlaces derivam sua prioridade de programação básica do processo que os possui. Os atributos do enlace incluem uma prioridade básica e uma prioridade dinâmica. Ao chamar comandos para mudar a prioridade de um enlace, a prioridade básica é alterada. Não se pode colocar a prioridade de um enlace divergente mais que dois passos da prioridade do processo que os possui. Os enlaces não podem crescer de modo a se tornarem muito mais importantes que seus pares.

Embora um processo não possa promover muito seus enlaces, o sistema pode, garantindo um tipo de promoção de campo para os enlaces que empreendem missões importantes. Por exemplo, quando o usuário dá entrada em uma janela , o sistema sempre eleva todos os enlaces do processo que possui a janela. Esses reforços temporários acrescidos à prioridade básica atual do enlace formam a prioridade dinâmica. O programador escolhe os enlaces a serem executados com base em sua prioridade dinâmica. Os reforços da prioridade dinâmica começam a degradar imediatamente. A prioridade dinâmica de um enlace volta um nível sempre que ele recebe outra fatia de tempo e finalmente estabiliza em sua prioridade básica.

#### 3.1.3.2 SINCRONISMO

Para que os enlaces possam rodar corretamente, eles precisam ser sincronizados freqüentemente. Suponha que um enlace crie um pincel e depois crie vários enlaces que o compartilham e desenham com ele. O primeiro enlace não deve destruir o pincel até que os outros terminem de desenhar. Vê-se que a situação exige um meio de coordenar a seqüência das ações dos vários enlaces. O Win32 suporta um conjunto de objetos de sincronismo, incluindo *mutexes*, semáforos, eventos e seções críticas. Todos são objetos criados pelo gerenciador de objetos e trabalham de modo análogo. Um enlace que quer executar alguma ação coordenada espera por uma resposta de um desses objetos e só prossegue após recebê-la. O programador retira da fila de expedição os objetos que estão esperando para que não consumam tempo do processador. Quando o sinal chega, o programador permite que o enlace seja retomado. Como e quando o sinal chega depende do objeto.

Tomemos os *mutexes* como exemplo. A característica essencial de um *mutex* é que somente um enlace pode possuí-lo. Se vários enlaces precisam trabalhar com um único arquivo, pode-se criar um *mutex* para protegê-lo. Quando qualquer enlace inicia uma operação de arquivo, ele primeiro pede o *mutex*. Se ninguém mais pede o *mutex*, o enlace prossegue. Por outro lado, se outro enlace acabou de pegar o *mutex*, o pedido falha e o enlace é bloqueado, tornando-se suspenso enquanto espera pela posse. Quando um enlace termina de escrever, ele libera o *mutex* e o que está esperando volta a ser ativado, recebe o *mutex* e executa suas próprias operações de arquivo. O *mutex* não protege nada ativamente. Ele apenas funciona porque os enlaces que o utlizam concordam em não escrever no arquivo sem possuí-lo. Quando vários enlaces compartilham todos os recursos do sistema, você deve considerar se deve sincronizar seu uso. Segue uma descrição sucinta e informal sobre o funcionamento dos objetos de sincronização citados.

- Um objeto de *mutex* funciona como um portão estreito para passar um enlace por vez.
- Um objeto de semáforo funciona como um pedágio no qual um número limitado de enlaces pode passar de uma vez só vez.
- Um objeto de evento difunde um sinal público para qualquer enlace ouvir.
- Um objeto de seção crítica funciona exatamente como um *mutex*, mas somente dentro de um único processo.

Fundamentalmente, um objeto de sincronismo, como outros objetos do sistema, é apenas uma estrutura de dados. Os objetos de sincronismo têm dois estados: sinalizados e não sinalizados. Os enlaces interagem com os objetos de sincronismo trocando o sinal ou esperando por ele. Um enlace que espera está bloqueado e não é executado. Quando o sinal ocorre, ele recebe o objeto, desliga o sinal, executa alguma tarefa sincronizada e volta a ligar o sinal quando abandona o objeto.

Um evento é o objeto que um programa cria quando exige um mecanismo para alertar os enlaces caso ocorra alguma ação.

Além de *mutexes*, semáforos, eventos e objetos críticos, os enlaces podem esperar por outros objetos. Esses quatro tipos de objeto só existem para sincronismo, mas às vezes faz sentido esperar por um processo, por um enlace ou por um arquivo, por exemplo. Esses

objetos também servem para outros objetivos, mas também possuem um estado de sinal como os objetos de sincronismo. Os processos e os enlaces sinalizam ao terminarem. Os arquivos sinalizam quando uma operação de escrita ou leitura acaba. Os enlaces podem esperar por qualquer um desses sinais.

### 3.1.4 BIBLIOTECAS DE VÍNCULO DINÂMICO (DLLS)

No ambiente MS-DOS, todos os módulos do programa são vinculados (*linked*) estaticamente durante o processo de construção (*build*). O Windows permite que o vínculo seja dinâmico, utilizando-se bibliotecas especificamente construídas, que podem ser lidas e vinculadas em tempo de execução do programa. Essas bibliotecas de vínculo dinâmico (DLL's) podem ser compartilhadas por múltiplos programas, permitindo economizar memória e espaço em disco. O uso de DLLs reforça a modularidade de um programa, pois essas bibliotecas podem ser testadas e compiladas separadamente.

Para que um arquivo possa ser executado pelo Windows, precisa estar no formato PE (*Portable Executable*). Como o nome sugere, este tipo de arquivo possui portabilidade de modo que pode ser executado em computadores tanto com processadores Intel, MIPS, Alpha, Power PC, Motorola 68000, assim como RISC. Um campo de grande importância num arquivo PE é a sua lista de importação, que contém uma lista das funções das quais o executável depende e que poderá chamar quando estiver sendo executado. Esta lista também possui o nome da Dll que contém a função. Ao carregar o executável, o Windows checa se todas as funções e todas as Dlls estão disponíveis. Se não estiverem disponíveis, o sistema não roda o programa.

Existem várias situações em que o uso de Dlls pode ser vantajoso. Por exemplo, dados que precisem de manutenção freqüente podem ser colocados em uma Dll, pois ao invés de atualizar todo o arquivo executável, atualiza-se um arquivo menor e em separado, facilitando a manutenção. Caso um aplicativo seja composto de mais de um programa, também se pode escrever DLLs para que os programas compartilhem o código e os dados nelas existentes. O próprio Windows usa DLLs para armazenar o código da sua API.

### 3.1.5 A INTERFACE DE PROGRAMAÇÃO DE APLICAÇÕES WINDOWS API

Uma das principais propostas de uma API é descrever como as aplicações de um computador e os desenvolvedores de *software* podem acessar um conjunto de funções de interesse, sem precisar ter acesso ao código fonte dessas funções ou da biblioteca, ou, até mesmo, sem exigir conhecimento detalhado sobre o funcionamento interno de cada função. É importante destacar que a API, em si, é tão abstrata quanto uma interface, e que cada nova versão do Microsoft Windows insere novas contribuições a ela.

As funções de serviços básicas da API dão ao aplicativo acesso aos recursos do computador e às funcionalidades do sistema operacional, como memória, sistema de arquivos, dispositivos, processos e *threads*. Um aplicativo pode então se utilizar dessas funções para gerenciar e monitorar os recursos de que necessita para completar o seu trabalho. Por exemplo, um aplicativo pode utilizar funções de gerenciamento de memória da API do Windows para alocar dados em espaços de memória durante a sua execução, liberando-os ao final, isto é, realizando alocação dinâmica de memória. Dentre essas funções, também estão incluídas aquelas de gerenciamento de processos e de sincronização, já descritas, que iniciam e coordenam a operação de múltiplos aplicativos ou de múltiplas *threads* de um único aplicativo no acesso.

# CAPÍTULO 4

## A ARQUITETURA WOSA/XFS

Para melhor caracterizar o módulo de fornecimento de dados constituinte do sistema de aquisição, deve-se fazer um estudo sobre a sua arquitetura de referência, WOSA/XFS. Assim, este capítulo destina-se a melhor descrever essa arquitetura, bem como a implementação em si do Provedor de Serviços.

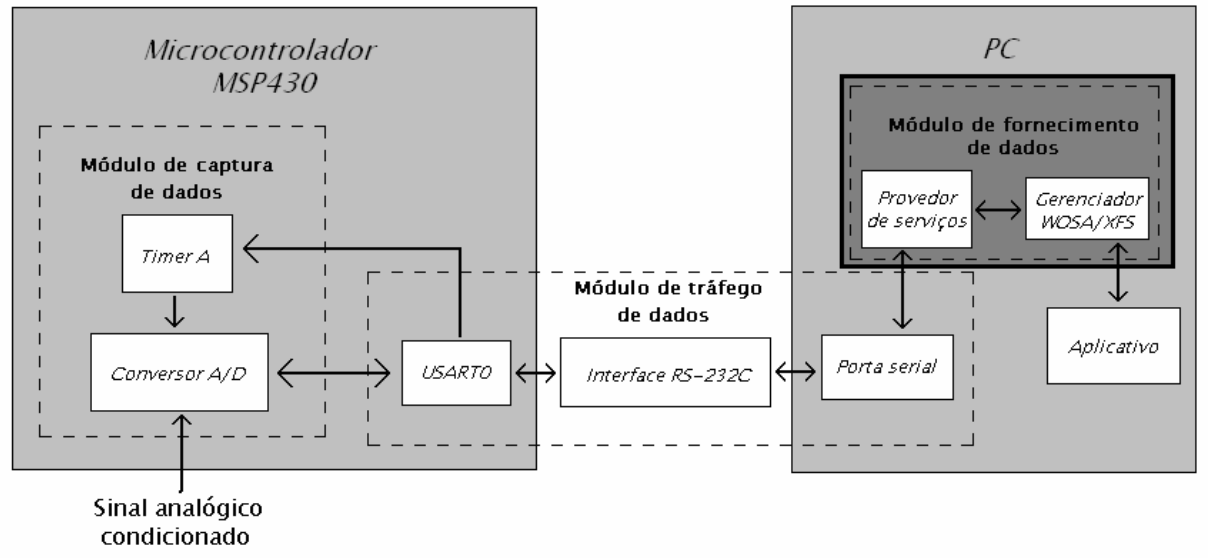


*Figura 4 – Módulo descrito no Capítulo 4*

O WOSA (*Windows Open Systems Architecture*) compreende uma família de interfaces para ambientes de computação que escondam a complexidade dos usuários e dos desenvolvedores de aplicações. O WOSA/XFS foi planejado para incluir especificações para o acesso de periféricos, transações bancárias por mensagem e sua administração, bem como serviços relacionados a redes financeiras. No entanto, é importante ressaltar que a flexibilidade que essa arquitetura provê é motivo da difusão de seu uso em outras áreas do conhecimento, como instrumentação biomédica.

Seu ponto principal é definir uma série de APIs (*Aplication Program Interfaces*), SPIs (*Service Provider Interfaces*) correspondentes e serviços de apoio, provendo acesso a serviços financeiros para aplicações baseadas em Windows. Esta arquitetura requer uma especificação que defina o padrão de interfaces para que, por exemplo, uma aplicação que use uma API para

se comunicar com um determinado provedor de serviço também possa se comunicar com o provedor de um outro fornecedor, sem qualquer alteração.

Em resumo, o WOSA/XFS define uma arquitetura geral para acessar os provedores de serviços por aplicações baseadas em Windows. É importante notar que essa padronização do acesso a dispositivos restritos a aplicações financeiras pode oferecer às instituições financeiras melhoria na produtividade e flexibilidade, já que esses dispositivos são geralmente complexos.

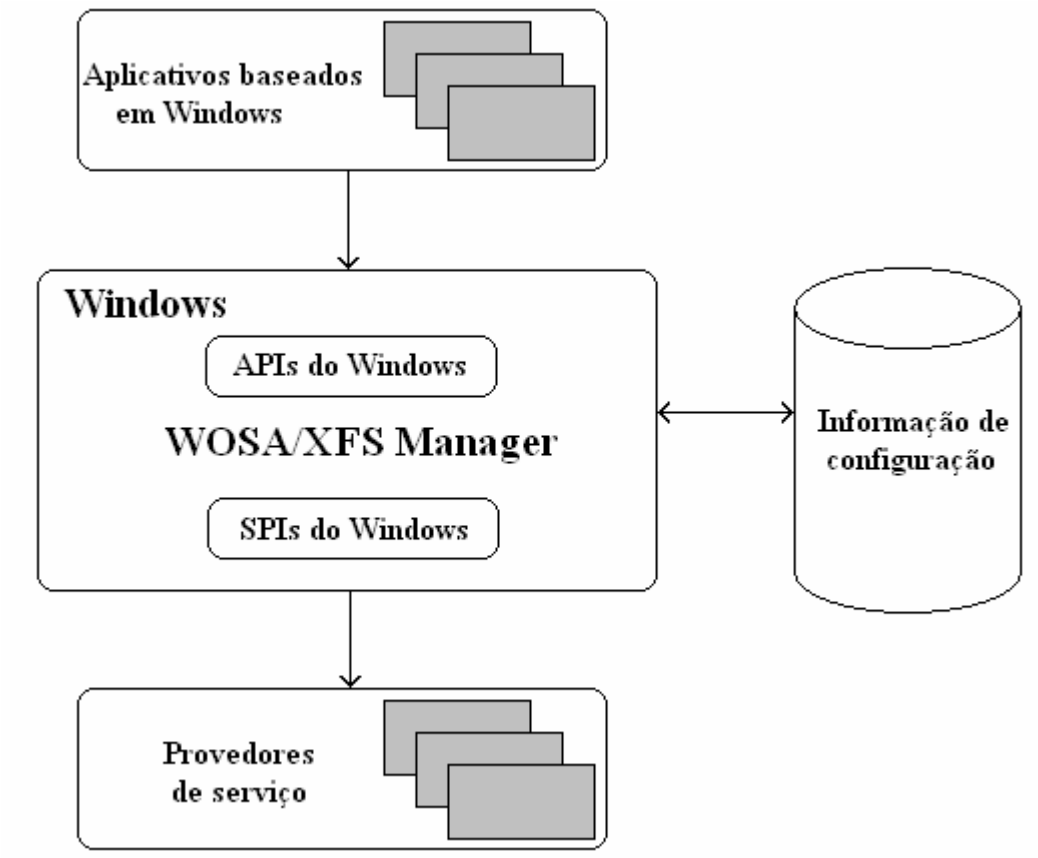


*Figura 5 – Arquitetura WOSA/XFS*

Os aplicativos se comunicam com os provedores de serviço, pelo gerenciador do WOSA/XFS, usando as funções da API. Na implementação do WOSA/XFS deve constar um gerenciador (*WOSA/XFS Manager*), que mapeia uma API específico à sua SPI correspondente, enviando o pedido ao provedor de serviço apropriado. Dessa forma, garante-se que API e SPI não se comuniquem diretamente.

Os desenvolvedores dos serviços financeiros a serem desempenhados via XFS e os fornecedores de periféricos financeiros são os responsáveis pelo desenvolvimento e

distribuição dos provedores de serviços dos seus serviços e dispositivos. É necessária uma rotina de inicialização para cada dispositivo ou serviço em que serão definidas as configurações apropriadas. Essas informações permitirão que uma aplicação obtenha informações de capacidade e *status* dos dispositivos e serviços disponíveis a qualquer tempo.

As funções primordiais dos provedores de serviços são:

- Converter pedidos genéricos de serviço em comandos específicos;
- Enviar pedidos de serviços a um dispositivo ou servidor local, ou a um sistema remoto, definindo uma interface *peer-to-peer* (P2P) entre os provedores de serviços;
- Ordenar o acesso de aplicações múltiplas a um único dispositivo ou serviço, ou a um sistema remoto, provendo acesso exclusivo quando requisitado;
- Gerenciar a interface de hardware a dispositivos ou serviços;
- Gerenciar a natureza assíncrona dos dispositivos e serviços em uma maneira apropriada, indicando essa capacidade ao Gerenciador XFS e às aplicações via mensagens de Windows.

A arquitetura do sistema é capaz de oferecer soluções eficazes a problemas complexos, pois foi desenvolvida para oferecer o máximo de flexibilidade em suas capacidades:

- Múltiplos provedores de serviços, desenvolvidos por diferentes fornecedores, podem coexistir em um único sistema e em uma rede;
- A definição das classes de serviços é baseada nas funcionalidades de serviço, sem qualquer referência às configurações físicas. Dessa forma, um dispositivo capaz de prover diversas funcionalidades não é tratado como único, mas sim como um conjunto de serviços lógicos. Deve-se notar que também um único serviço pode envolver mais de um dispositivo físico;
- Diferentes usuários podem compartilhar um dispositivo físico, cuja sincronização é de responsabilidade do provedor de serviços;
- A definição das funções da API e os serviços a elas associados oferecem a funcionalidade *time-out*, evitando a ocorrência de *deadlock* quando dois aplicativos peçam acesso exclusivo a múltiplos serviços simultaneamente;
- A arquitetura é definida de forma a permitir o desenvolvimento futuro do gerenciamento da rede e do sistema.

Para o sistema WOSA/XFS como um todo, o foco é dado no provedor de serviço e não no serviço em si, visto que a comunicação de cada provedor com seus serviços deve ser definida pelo seu fornecedor. Sendo assim, os aplicativos e o gerenciador do XFS não enxergam essas especificações internas dos fornecedores.

## 4.1 RESUMO DAS API'S E SPI'S

Uma API é estruturada por:

- Funções básicas: comuns aos dispositivos e classes de serviços do WOSA/XFS. Ex: Open/Close
- Funções administrativas: utilizadas para gerenciar os serviços e os dispositivos. Ex: inicialização.
- Comandos específicos: usados para requerer informações sobre um dispositivo/serviço e para inicializar funções específicas de dispositivos e serviços. O conjunto de parâmetros deve ser especificado para cada classe de serviços, porém deve-se padronizar os códigos e estruturas de funções para uma grande variedade de dispositivos.

Um cenário típico do uso de APIs é ilustrado a seguir.

StartUp: conecta o aplicativo ao XFS *Manager*

Open: abre uma sessão entre o aplicativo e o provedor de serviços

Register: especifica o conjunto de mensagens a serem enviadas pelo provedor de serviços para o aplicativo

Lock: obtém acesso exclusivo ao serviço para o aplicativo

Funções de Execução: comandos específicos

Unlock: libera o acesso exclusivo

Deregister: interrompe o envio de mensagens do provedor de serviços para o aplicativo

Close: termina a sessão entre o aplicativo e o provedor de serviços

CleanUp: desconecta o aplicativo do XFS *Manager*

Deve-se notar que o conjunto de instruções acima não é fixo, podendo-se, a qualquer momento, alterar o conjunto de mensagens a serem trocadas, bem como requerer o acesso exclusivo por apenas parte da sessão.

## 4.2 ARQUITETURA E IMPLEMENTAÇÃO

A especificação define um conjunto padrão de interfaces que oferecem interoperabilidade para fornecedores múltiplos, isto é, a API utilizada para se comunicar com um provedor de serviço de um fornecedor, também o fará com provedores de serviços de outros fornecedores. Da mesma maneira, qualquer provedor de serviços que se enquadre nas definições da SPI poderá se comunicar com os aplicativos que também se enquadrem.

Os prefixos utilizados nas funções das interfaces do WOSA/XFS são:

Tabela 4.1 Prefixos utilizados nas funções das interfaces do WOSA/XFS

| **Tipo de função: Prefixo** | **Funções chamadas por:** | **Funções providenciadas por:** |
|---|---|---|
| Funções de API: WFS... | Aplicativos | XFS *Manager* |
| Funções de SPI: WFP... | XFS *Manager* | Provedor de Serviços |
| Funções de configuração ou suporte: WFM... | Aplicativos<br>Provedor de Serviços | XFS *Manager* |

## 4.3 WOSA/XFS *MANAGER*

Sua função é gerenciar o subsistema WOSA/XFS. É ele quem mapeia a função da API para a respectiva função da SPI, chamando o provedor de serviço apropriado. Deve-se notar que as chamadas são sempre para um provedor local.

O *Manager* determina qual provedor de serviço chamar a partir do parâmetro de nome lógico da função **WFSOpen** ou da função **WFSAsyncOpen**. O nome lógico permite o acesso às informações de configuração que definem a classe do serviço (impressora, dispensador de notas, entre outros), o tipo do serviço (impressora de recibo, impressora de chuques) e o provedor de serviço (nome do arquivo de DLL), além de informações adicionais. O nome lógico deve ser único, pelo menos, para cada estação de trabalho. Além disso, o XFS *Manager* oferece funções de suporte e configuração (funções do tipo **WFM…**).

Deve-se lembrar que, antes de um aplicativo utilizar os serviços gerenciados pelo WOSA/XFS, é necessário se identificar ao subsistema. Isso é feito utilizando-se a função **WFSStartUp**, geralmente na inicialização do aplicativo. De forma similar, a função **WFSCleanUp** é chamada durante a finalização do aplicativo. Se o aplicativo for finalizado sem chamar esta função, o XFS *Manager* se desconecta do aplicativo automaticamente.

## 4.4 PROVEDORES DE SERVIÇO (*SERVICE PROVIDERS*)

Cada serviço do WOSA/XFS de cada fornecedor é acessado por um módulo específico chamado provedor de serviço. Suas principais funções, trabalhando em conjunto com seus respectivos serviços ou *drivers* são descritas a seguir. Note que a forma de implementação dessas funções é atribuição do desenvolvedor do provedor de serviço. São essas as sua principais funções:

- Enviar os pedidos ao dispositivo ou serviço, que devem estar em uma estação de trabalho remota;
- Traduzir pedidos genéricos em comandos específicos;
- Ordenar o acesso por múltiplos aplicativos;
- Gerenciar a interface com os dispositivos de hardware para controlar os dispositivos. Por exemplo, os provedores de serviços podem utilizar *drivers* padrão do sistema operacional ou criados pelo próprio fornecedor;
- Gerenciar a natureza assíncrona dos serviços de maneira consistente com aplicativos;
- Realizar procedimentos para a cobertura de erros: em alguns tipos de falhas de software, como as capazes de desconectar o provedor de serviços da aplicação, o provedor de serviços deve prover um desligamento ordenado da sessão com o aplicativo.

Os provedores de serviços do WOSA/XFS podem ser empacotados em DLLs de várias maneiras: uma DLL para cada provedor de serviços, múltiplos provedores de serviços por DLL ou então todos em uma única DLL.

## 4.5 - FUNÇÕES SÍNCRONAS, ASSÍNCRONAS E IMEDIATAS

O Windows e o WOSA/XFS são baseados num modelo assíncrono. A arquitetura do WOSA/XFS, contudo, permite aos aplicativos que usam as suas interfaces agirem de forma síncrona ou assíncrona. Assim, as funções pertinentes podem requerer acesso exclusivo a um serviço.

As funções da API podem ter 3 modos de sincronização: assíncrono, síncrono ou imediato. Já as funções da SPI trabalham somente nas formas assíncrona ou imediata.

O modo assíncrono é utilizado para operações em que não se conhece o tempo necessário para seu término. Dessa maneira, o aplicativo pode trabalhar no ambiente comum do Windows, baseando-se em eventos e mensagens. Um pedido assíncrono é processado da seguinte forma:

- O aplicativo chama o *Manager*;
- O *Manager* gera uma seqüência de números (*RequestID*), designa-o para o serviço e chama o provedor de serviços;
- O provedor de serviços agenda o pedido para posterior processamento e imediatamente retorna ao XFS *Manager*;
- O *Manager* retorna o *RequestID* para o aplicativo com um *status* indicando que o pedido foi iniciado e está sendo processado;
- Em algum momento, o provedor de serviços processa o pedido;
- Ao terminar, o provedor de serviços manda uma mensagem de finalização para o *Window handle* de janela especificado pelo aplicativo na chamada original, que contém um ponteiro para uma estrutura WFSRESULT contendo os resultados do pedido.

O modo síncrono é utilizado para operações em que não se conhece o tempo necessário para seu término, mas o aplicativo deseja lidar com a função de maneira sequencial. O XFS *Manager* não retorna o controle para o aplicativo enquanto a operação não tiver sido completada. Cada chamada síncrona feita pelo aplicativo é traduzida pelo XFS *Manager* para a sua contraparte assíncrona na SPI antes de ser passada ao provedor de serviços.

Se uma operação síncrona não é completada imediatamente em um sistema Windows 3.X, o XFS *Manager* executa um *loop* de mensagens do Windows referentes à *thread* que está sendo chamada, de forma a manter o sistema do Windows rodando. No Windows NT, a *thread* chamada pelo aplicativo é bloqueada quando o pedido é completado.

Um pedido síncrono é processado da seguinte forma:

- O aplicativo chama o *Manager*;
- O *Manager* traduz o pedido para uma SPI assíncrona, gera um *RequestID* para rastrear o pedido, providencia o seu próprio *Window handle* para receber a mensagem de finalização e chama a DLL do provedor de serviços. O *Manager* retorna o *RequstID* para o aplicativo com um *status* indicando que o pedido foi iniciado e está sendo processado;
- O provedor de serviços agenda o pedido para posterior processamento e imediatamente retorna ao XFS *Manager*;
- O XFS *Manager* simula o processamento síncrono;
- Em algum momento, o provedor de serviços processa o pedido;
- Ao terminar, o provedor de serviços manda uma mensagem de finalização para o *Window Handle* especificado pelo XFS *Manager*, que contém um ponteiro para uma estrutura WFSRESULT contendo os resultados do pedido;
- O XFS *Manager* converte a informação da mensagem de finalização para parâmetros apropriados, que são retornados para o aplicativo, desbloqueando-o.

As funções imediatas são funções que tipicamente não se comunicam com um serviço ou dispositivo fixo (ou usem a rede), sendo completadas imediatamente, com ou sem sucesso. São executadas de duas formas diferentes: processadas inteiramente pelo XFS *Manager*, que retorna imediatamente para o aplicativo (a função **WFSStartUp**, por exemplo); ou passadas do *Manager* para o provedor de serviços como uma SPI imediata (a função **WFSCancelAsyncRequest**, por exemplo). No último caso, o provedor de serviços processa o pedido e retorna imediatamente para o XFS *Manager*, que retorna imediatamente para o aplicativo.

## 4.6 - PROCESSAMENTO DE FUNÇÕES API

Quando um aplicativo chama uma função API do WOSA/XFS, um dos seguintes procedimentos acontece:

- A função é convertida diretamente para a função SPI correspondente pelo XFS *Manager*;
- O *Manager* faz algum pré-processamento e depois converte para a função SPI correspondente;
- O *Manager* faz algum pré-processamento e depois converte para uma função SPI diferente, que a transmite ao provedor de serviço. A maioria das funções APIs síncronas são desse tipo, já que são traduzidas para as suas funções SPIs (assíncronas) equivalentes;
- O *Manager* faz algum pré-processamento e depois traduz a função API para múltiplas funções SPI transmitidas ao provedor de serviço;
- A função é completamente processada dentro do *Manager*.

## 4.7 - ABRINDO UMA SESSÃO

Assim que uma conexão entre um aplicativo e o *Manager* foi estabelecida (via **WFSStartUp**), o aplicativo estabelece uma sessão virtual com o provedor de serviços mediante uma requisição **WFSOpen** (ou **WFSAsyncOpen**). Essas funções são direcionadas para "serviços lógicos" na forma definida na configuração do WOSA/XFS. Um *handle* de serviço (*hService*) é designado para a sessão e é usado em todas as chamadas para o serviço no período de duração da sessão.

Quando os aplicativos precisam utilizar dispositivos compostos interdependentes, eles podem-se utilizar do conceito de "identidade do aplicativo", por meio da função **WFSCreateAppHandle**. Essa função fornece um *handle* para o aplicativo (*hApp*), que é único dentro do sistema, e pode ser chamadas diversas vezes para gerar diversos *handles* únicos. O parâmetro *handle* do aplicativo é então usado pela função **WFSOpen**, que ordena ao provedor de serviços ligar o *handle* do aplicativo especificado à sessão iniciada. Isso permite que um processo do aplicativo (potencialmente *multi-threaded*) aja como aplicativos

múltiplos para o subsistema XFS, permitindo o uso efetivo de dispositivos compostos interdependentes.

Na abertura de uma sessão, o *Manager* realiza as seguintes ações:

- Recupera as informações de configuração do serviço lógico especificado, a fim de determinar o nome da DLL do provedor de serviços;
- Carrega a DLL contendo o provedor de serviços requisitado, caso ele ainda não tenha sido carregado;
- Realiza o pré-processamento e a tradução necessários, dependendo do tipo de abertura de sessão requisitada (síncrona ou assíncrona);
- Gera um *handle* único de serviço, que identifica a sessão com o provedor de serviço, para passá-lo como parâmetro ao aplicativo;
- Chama a função **WFPOpen** do provedor de serviços, passando os parâmetros necessários.

O provedor de serviços então realiza as seguintes ações:

- Negocia a versão utilizada, usando os parâmetros que especificam a versão de SPI requerida pelo *Manager* e a versão de interface do serviço específico requisitado pelo aplicativo;
- Recupera a informação de configuração;
- Estabelece assincronamente a sessão com o serviço especificado na configuração para a estação de trabalho especificada, se utilizando, caso precise, das facilidades de transporte oferecidas;
- Após o recebimento do pedido, encaminha a mensagem de recebimento (WFS_OPEN_COMPLETE), que retorna ao aplicativo para a chamada **WFSAsyncOpen** , e ao *Manager* para a chamada **WFSOpen**.

Note que o programador tem a opção de utilizar a função **WFSOpen** uma única vez na inicialização do aplicativo ou a cada vez em que o serviço for solicitado. Cada uma dessas técnicas tem suas vantagens. O primeiro caso fornece melhor performance, e o segundo pode ser mais fácil de programar. Em qualquer caso, o subsistema WOSA/XFS retorna um *handle* de serviço que deve ser utilizado em toda a comunicação posterior com o serviço.

O aplicativo deve realizar uma abertura de sessão para cada serviço lógico a ser utilizado, mesmo que sejam do mesmo tipo. Por exemplo, um aplicativo que utilize duas impressoras de recibo distintas deve abrir duas sessões, uma para cada serviço lógico. De forma semelhante, para um dispositivo composto, deverão ser abertas sessões para cada serviço lógico, ou seja, para cada um de seus componentes mobilizados.

## 4.8 - FECHANDO UMA SESSÃO

Quando um aplicativo não precisa mais de um serviço particular, ele faz um pedido **WFSClose** ou **WFSAsyncClose**. O subsistema WOSA/XFS fecha então a sessão da seguinte maneira:

- O *Manager* chama a função **WFPClose** do provedor de serviços;
- O provedor de serviços agenda o pedido para processamento posterior e retorna imediatamente ao *Manager*. Nesse ponto, o *handle* do serviço se torna inválido;
- Em algum momento, o provedor de serviços processa o pedido de fechamento, comunicando com o serviço, se necessário, para completar o pedido;
- Os pedidos que foram feitos pelo aplicativo antes do fechamento são executados;
- Se o aplicativo que pede o fechamento tem a exclusividade sobre o serviço, esta é desfeita automaticamente;
- O serviço apaga as suas informações administrativas.

Se o WOSA/XFS perde a sua conexão com o aplicativo, ele fecha a sessão como o descrito acima, e:

- Gera um evento de desconexão da classe SYSTEM_EVENT com o aplicativo;
- Converte as mensagens de término de comandos e as mensagens de notificação de eventos deste serviço para o aplicativo em eventos de “a mensagem não pode ser entregue ao destinatário” da classe SYSTEM_EVENT.

## 4.9 - INFORMAÇÃO DE CONFIGURAÇÃO

O *Manager* usa suas informações de configuração para definir as relações entre os aplicativos e os provedores de serviço. Em particular, esta informação mapeia a interface de

serviço lógico fornecida na API (via nome lógico do serviço) aos pontos de entrada apropriados ao provedor de serviço.

Essa informação de configuração também inclui informações especificas sobre serviços lógicos e provedores de serviços, das quais algumas são comuns a todos provedores de soluções. Pode também incluir informações sobre serviços físicos, caso existirem, e informações especificas do fornecedor. O local de armazenamento da informação é transparente tanto para o aplicativo quanto para os provedores de serviço. Estes sempre a armazenam e a recuperam usando as funções de configuração oferecidas pelo *Manager*, para que se tenha portabilidade nas plataformas Windows.

É de responsabilidade dos provedores de solução e dos desenvolvedores de provedores de serviço: implementar as utilidades apropriadas de *setup* e gerenciamento; criar e gerenciar a informação de configuração do subsistema XFS e de seus provedores de serviço, usando suas funções de configuração.

Essas funções são usadas pelos provedores de serviço e aplicativos para escrever e recuperar a informação de configuração de um subsistema WOSA/XFS, que é armazenada em uma estrutura hierárquica chamada *registro de configuração XFS*. A estrutura e as funções são baseadas na arquitetura de registro do Win32 e nas funções API, e são implementadas no Windows NT e versões futuras do Windows usando o Registro e suas funções associadas. Para implementações no Windows 3.1 e Windows for Workgroups baseadas em Win32, apenas parte da funcionalidade descrita está disponível e é definida pelo SDK (*System Development Kit)*.

No registro de configuração, cada nó é chamado de chave, tendo um nome e, opcionalmente, valores. Todos valores consistem em um par nome-data, ambos sendo *strings* de caracteres terminadas em zero. A estrutura é a seguinte:

- O nível mais alto é o nó-raiz para o subsistema WOSA/XFS. Seu nome é WOSA/XFS_ROOT, e é uma sub-chave de HKEY_CLASSES_ROOT no Registro Win32.
- O segundo nível contem pelo menos três chaves: XFS_MANAGER, LOGICAL_SERVICES e SERVICE_PROVIDERS. Outras chaves (PHYSICAL_SERVICES, por exemplo) podem ser definidas e usadas.

- Abaixo de XFS_MANAGER existem valores e/ou chaves para informações que o *Manager* cria e usa.
- Abaixo de LOGICAL_SEVICES existe uma chave para cada serviço lógico definido para o sistema aonde o registro reside. Os nomes das chaves e dos serviços lógicos (parâmetro *lpszLogicalName* das funções **WFSOpen**, **WFSAsyncOpen** e **WFPOpen**). Como existe apenas um registro por estação de trabalho, isso sugere que os nomes dos serviços lógicos sejam únicos pelo menos na estação de trabalho.
- Abaixo de SERVICE_PROVIDERS existe uma chave para cada provedor de serviços definido pelo sistema.

As funções de configuração fornecem capacidade de criar, enumerar, abrir e apagar chaves, e de definir, buscar e apagar valores dentro de cada chave. Programas de configuração dos fornecedores definem a estrutura de registro e seu conteúdo, usando essas funções. O terceiro nível contém os valores e as chaves que definem como o subsistema XFS, os serviços e os provedores são configurados. Essas informações são usadas pelo *Manager*, pelos aplicativos e pelos provedores de serviços. Informação específica do fornecedor pode ser adicionada para qualquer chave na estrutura, usando valores opcionais.

A figura abaixo ilustra a estrutura do registro de configuração:

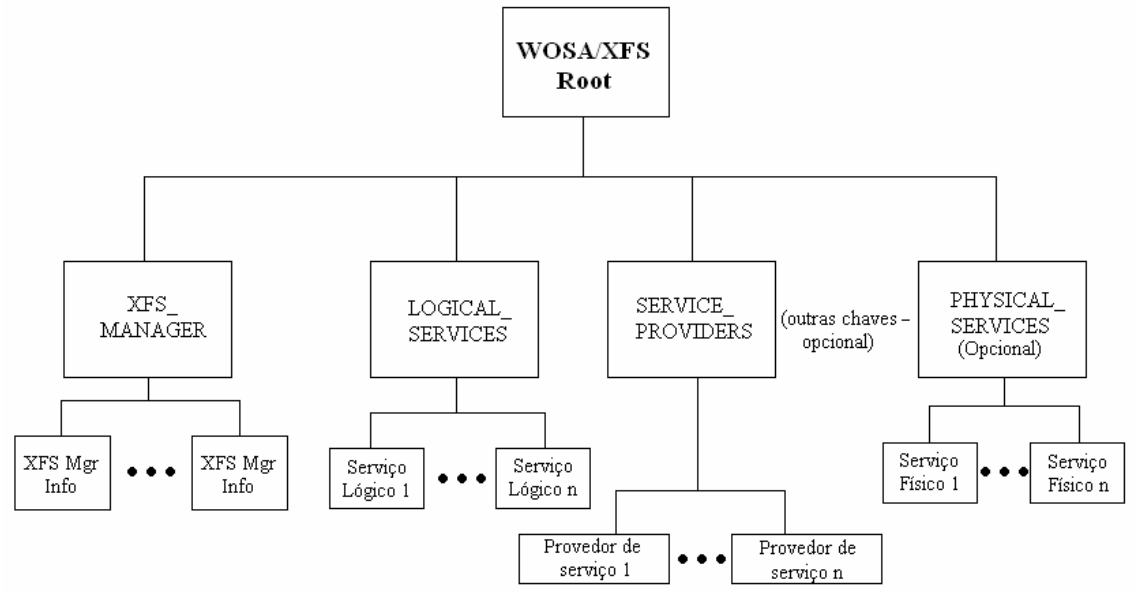


*Figura 6 – Estrutura do registro de configuração*

A chave do *Manager* tem os seguintes valores opcionais:

- Tracefile: o nome do arquivo que contém dados de caminho (*trace data*). Se esse valor não é definido na configuração, é atribuído o padrão arquivo diretório\nome C:\XFSTRACE.LOG.
- ShareFilename: o nome do arquivo de memória mapeada usado pelas funções de gerenciamento do *Manager*.
- ShareFilesize: o tamanho do arquivo de memória mapeada usado pelas funções de gerenciamento do *Manager*.

Toda chave de serviço lógico tem três valores obrigatórios:

- class: classe do serviço lógico.
- type: tipo do serviço lógico; os valores padrão são definidos no SDK do WOSA/XFS.
- provider: nome do provedor de serviços que provê o serviço lógico (nome da chave correspondente à chave do provedor de serviço).

A chave do provedor de serviço também tem três valores obrigatórios:

- dllname: nome do arquivo que contem a DLL do provedor de serviços.
- vendor_name: nome do fornecedor desse provedor de serviços.
- version: número da versão desse provedor de serviços.

## 4.10 - SERVIÇO EXCLUSIVO E ACESSO A DISPOSITIVOS

Essa seção descreve como se dá o acesso dos aplicativos a serviços e dispositivos por meio dos subsistemas WOSA/XFS, usando a facilidade *lock*.

Um aplicativo requisita acesso exclusivo a um serviço particular, especialmente em combinação com outros serviços associados. Por exemplo, quando um aplicativo precisa usar o leitor de cartões magnéticos e a impressora de recibos para completar a transação. Esse aplicativo deve então garantir seu acesso a ambos os dispositivos antes de iniciar a transação, assim como inibir o acesso de outros aplicativos a eles até que a transação se complete e seja retirada a exclusividade. Esse procedimento é alcançado utilizando-se as funções **WFSLock** (ou **WFSAsyncLock**) e a função complementar **WFSUnlock**.

Um aplicativo deve agir de maneira cooperativa quando obtiver a exclusividade sobre um serviço, mantendo-a pelo mínimo período de tempo. Os aplicativos devem usar técnicas

apropriadas para evitar o *deadlock* quando recebem a exclusividade sobre múltiplos serviços, usando os parâmetros de *timeout* nas funções *lock*.

Existem situações em que o acesso exclusivo não é obrigatório, de modo que não será necessário utilizar a função *lock* antes que o aplicativo execute operações sobre o serviço.

A política de exclusividade estabelece regras que os serviços devem utilizar no gerenciamento dos pedidos. Nessa política, os pedidos são divididos em três tipos:

- *Não-deferidos*: podem ser processados completamente por um serviço assim que chegam. Exemplos: **WFPOpen**, **WFPRegister** e a maioria das chamadas **WFPGetInfo**.
- *Deferidos*: pode não ser possível processá-los completamente assim que chegam, geralmente porque requerem *hardware* e/ou interação do operador. Exemplos: **WFPExecute** e algumas chamadas **WFPGetInfo**.
- *Lock*: são as chamadas **WFPLock**.

Primeiramente, a política de exclusividade será descrita para os dispositivos independentes, isto é, a operação de seus serviços lógicos não é dependente de outros serviços.

### 4.10.1 - POLÍTICA DE EXCLUSIVIDADE PARA DISPOSITIVOS INDEPENDENTES

Em seguida, será descrito como os pedidos são atendidos em cada um dos estados de um serviço exclusivo.

Estado do serviço: UNLOCKED

- Pedidos não deferidos são processados na chegada.
- Pedidos deferidos são dispostos em uma fila e processados em ordem de chegada (FIFO – *first in, first out*).
- Ao chegar um pedido **WFPLock,** o pedido de exclusividade é colocado na fila de espera e o estado do serviço muda para LOCK_PENDING.

Estado do serviço: LOCK_PENDING

- Todos os pedidos deferidos que estiverem na fila antes de o pedido de exclusividade chegar são processados FIFO, e então o pedido de exclusividade é processado. Dependendo do tipo de serviço ou dispositivo, o pedido de exclusividade pode ser processado FIFO ou de alguma outra forma.
- Ao ser processado um pedido de exclusividade, o estado do serviço muda para LOCKED, e quaisquer outros pedidos de exclusividade do mesmo proprietário serão também concedidos (o proprietário é o mesmo se vem da mesma estação de trabalho e tem o mesmo aplicativo e o mesmo *handle* de serviço).

  Estado do serviço: LOCKED

- Os pedidos que chegam (exceto os de exclusividade) são processados da seguinte forma:
  - Os não-deferidos são processados na chegada;
  - Os deferidos que não são pedidos **WFPExecute** são colocados na lista de espera;
  - Os pedidos **WFPExecute** do proprietário da exclusividade são colocados em uma lista de espera;
  - Os pedidos **WFPExecute** que não são do proprietário são rejeitados (com o código de erro WFS_ERR_LOCKED);
  - Os pedidos **WFPUnlock** e **WFPClose** encaminhados pelo proprietário são colocados em uma lista de espera. Note que um pedido de *close* a um serviço com estado LOCKED é tratado como um *unlock* seguido de *close*;
  - Os pedidos **WFPUnlock** e **WFPClose** que não são encaminhados pelo proprietário são tratados como pedidos não-deferidos, isto é, processados na chegada.
- A lista de espera é processada de forma FIFO.
- Ao chegar um pedido **WFPLock**, ele é processado se for do proprietário; senão, é posto na fila de pedidos de exclusividade.
- Quando um pedido de **WFPUnlock** ou **WFPClose** na lista de espera for processado, ou a conexão entre o serviço e o proprietário é perdida:
  - Se a fila dos pedidos de exclusividade não estiver vazia, o estado do serviço muda para LOCK_PENDING;
  - Se a fila dos pedidos de exclusividade estiver vazia, o estado do serviço muda para UNLOCK.

É importante incluir na maioria dos pedidos um parâmetro de *timeout*, que será gerenciado de maneira apropriada, isto é, quando expirado, o pedido será rejeitado com o código de erro WFS_ERR_TIMEOUT.

### 4.10.2 - DISPOSITIVOS COMPOSTOS

Dispositivos compostos são bastante comuns na indústria de serviços financeiros, e podem ser agrupados em três tipos:

- Dois ou mais dispositivos lógicos que compartilham algum atributo (físico, por exemplo), mas funcionam de forma completamente independente;
- Dois ou mais dispositivos lógicos que são funcionalmente interdependentes, como impressoras de recibo e de cheques, que utilizam o mesmo mecanismo;
- Dois ou mais dispositivos lógicos que simplesmente são visões lógicas diferentes de um único dispositivo físico, como uma única impressora que é gerenciada para funcionar como impressora de documentos e recibos.

O primeiro tipo não tem importância do ponto de vista XFS. Cada um dos dispositivos são separados de forma lógica e física, e as questões de configuração do sistema são deixadas para utilitários de aplicativos fora do escopo dessa especificação.

Os dois últimos tipos são tratados de forma idêntica no sistema XFS. Quando qualquer um dos dispositivos lógicos que formam um dispositivo composto oferece exclusividade, todos outros dispositivos componentes também a oferecem. Se o mesmo aplicativo (identificado pelo *handle* de aplicativo) explicitamente pede exclusividade sobre outro desses dispositivos lógicos, ela é garantida. Para que o aplicativo saiba que esses dispositivos fazem parte de um dispositivo composto, e, portanto, são interdependentes, a função **WFSLock** retorna um vetor de *handles* de serviço, definindo o conjunto de dispositivos dentro do dispositivo composto que lhe garantiram exclusividade. Isso permite que o aplicativo gerencie seu uso sobre esses dispositivos apropriadamente. Geralmente, o aplicativo usa o dispositivo de maneira seqüencial para evitar conflitos, mas se existir algum conhecimento sobre como os dispositivos se relacionam, pode-se multiplexar os pedidos de forma diferente.

Deve-se notar que um aplicativo também pode determinar como um dispositivo é composto, utilizando-se das capacidades do dispositivo retornadas pela função **WFSGetInfo**.

Existem várias maneiras diferentes que os programadores podem utilizar em aplicações financeiras como múltiplas *threads* e/ou processos. Cada serviço WOSA/XFS pode ser controlado por sua própria *thread*; todos os serviços podem ser controlados por uma única *thread*, com outras *threads*/processos usados por outras funções do aplicativo; diversas *threads* iguais podem lidar com todos os serviços abertos da forma necessária; etc. Em alguns desses modelos, o usuário de um serviço pode ser considerado um processo como um todo; em outros modelos, o usuário é uma única *thread*. A arquitetura WOSA/XFS permite ambos os modelos, oferecendo ao programador a capacidade de controlar a identidade do aplicativo. O programador pode fazer com que todas as *threads* de um processo pareçam um aplicativo para o provedor de serviços, identificar cada *thread* como um aplicativo diferente, ou criar algum modelo híbrido dos anteriores.

Para permitir essa flexibilidade, a identidade de cada aplicativo pode ser opcionalmente gerenciada utilizando-se o conceito de *handles* de aplicativos. Um *handle* de aplicativo (*hApp*) é criado pela função **WFSCreateAppHandle**, que é único para o sistema.A função **WFSOpen** toma opcionalmente como parâmetro o *handle* de aplicativo, que é sujeito ao *handle* de serviço (*hService*) retornado por essa função. Isso permite aos aplicativos que usam dispositivos compostos interdependentes ser implementados por qualquer combinação de processos e/ou *threads*, definindo de forma apropriada um conjunto de *handles*. Se o aplicativo utilizar o valor padrão WFS_DEFAULT_HAPP para o parâmetro *hApp* da função **WFSOpen**, essa facilidade não será utilizada, e o subsistema XFS automaticamente atribuirá um único *handle* de aplicativo para cada processo.

A política de exclusividade para dispositivos compostos interdependentes utiliza as mesmas regras que as utilizadas pelos dispositivos independentes, com algumas restrições adicionais. Para sincronizar o acesso de múltiplos dispositivos lógicos a um único dispositivo físico ou a dispositivos interdependentes, o serviço gerencia uma fila de pedidos de exclusividade e uma fila de pedidos deferidos para um conjunto de serviços lógicos relacionados. As restrições adicionais são:

Estado do serviço: LOCK_PENDING

- Quando um pedido de exclusividade foi atendido para um ou um conjunto de serviços lógicos:

- Todos os outros serviços relacionados no conjunto mudam para um estado reservado no qual são tratados como no estado LOCKED para pedidos de terceiros.
- Qualquer pedido de exclusividade do proprietário para um dos serviços reservados é garantido na chegada.
- Pedidos de exclusividade que não são do proprietário a dispositivos reservados são colocados na fila de pedidos de exclusividade.

Estado do serviço: LOCKED

- Qualquer um pedido de exclusividade do proprietário para um dos serviços reservados é deferido na chegada.
- Pedidos de exclusividade que não são do proprietário a um dos serviços reservados é colocado na fila de pedidos de exclusividade.
- Note que se os pedidos **WFPUnlock** ou **WFPClose** forem processados para o serviço e qualquer outro serviço lógico relacionado a esse serviço estiver no estado LOCKED, então o estado de serviço é reservado, e não UNLOCKED.
- Note também que se os pedidos **WFPUnlock** ou **WFPClose** forem processados para o serviço e qualquer outro serviço lógico relacionado a esse serviço estiver no estado reservado, então todos esses serviços mudam para o estado UNLOCKED.

## 4.11 - TIMEOUT

Existem dois domínios de tempo diferentes no sistema, cada um implicando em um conceito diferente de *timeout*:

- "user time" = tempo real; aqui, *timeout* simplesmente significa que o trabalho está demorando demais, e é definido pela aplicação e/ou pelo usuário (indicado pelo código de erro WFS_ERR_TIMEOUT.
- "SERVICE TIME" = tempo que o pedido ao serviço leva para ser atendido; tipicamente, a operação do dispositivo físico (indicado pelo código de erro WFS_ERR_DEV_NOT_READY ou WFS_ERR_HARDWARE_ERROR).

No sistema WOSA/XFS, o serviço gerencia o atraso, sem a necessidade de qualquer entrada do aplicativo, já que ele conhece as características do dispositivo e é capaz de gerar um evento de *timeout* se esse dispositivo estiver demorando, até mesmo se o valor *timeout* do

aplicativo não tiver sido excedido. Logo, o valor de *timeout* fornecido ao API pelo provedor de serviço é em tempo real (“user time”). Se o tempo for excedido, o provedos de serviços cancela o pedido e retorna um evento de *timeout* para o aplicativo. O aplicativo pode também especificar que o pedido deve aguardar o seu termino, independentemente de quanto tempo o pedido leve, especificando o valor especial WFS_INDEFINITE_WAIT.

## 4.12 - RETORNO DE STATUS DA FUNÇÃO

Quando a chamada de função API ou SPI é completada, é retornado um valor que define o *status* de término ou no caso de funções assíncronas, o *status* de inicio do processamento do pedido. Quando uma função assíncrona é completada, a mensagem de término inclui o *status* final do pedido. O valor retornado pela maioria das funções é um *handle* de resultado *hResult* (tipo HRESULT). Valores *hResult* são definidos para serem WFS_SUCCESS (zero) para sucesso; outros valores indicam o erro especifico que ocorreu, definido na especificação da função.

O *Manager* e os provedores de serviço retornam o *status* de um pedido na forma de um *hResult* de duas maneiras:

- Retornando um valor *hResult* como o retorno da função.
- Enviando uma mensagem de término à janela especificada no pedido. A mensagem contém um ponteiro para a estrutura que contém o *hResult*.

O mecanismo depende da categoria da função que estiver sendo processada, como segue:

- API Imediato
- O *Manager* processa o pedido e imediatamente retorna um *handle* de resultado. Em alguns casos, o *Manager* chama o provedor de serviço para processar o pedido, e então retorna o *handle* de resultado do provedor para o aplicativo.
- API Assíncrono
- Como o processamento é feito em vários passos, como descrito anteriormente, o *status* de retorno é também gerado em vários níveis:
  - O provedor de serviços realiza as validações que puderem ser processadas imediatamente.

- Se algum erro for detectado, o provedor de serviços retorna o *hResult* ao *Manager*, que imediatamente o retorna ao aplicativo.
- De outra forma, o pedido é agendado e o *hResult* WFS_SUCCESS é imediatamente retornado ao *Manager*, que o retorna ao aplicativo. Isso informa ao aplicativo que o pedido foi aceito e está sendo processado.
- Após o pedido ser completado, uma mensagem de término é enviada à janela do aplicativo. Essa mensagem aponta para a estrutura que inclui o *hResult* indicando o *status* de término do pedido.

- API Síncrono
  - Como um pedido de API síncrono é traduzido pelo *Manager* a um SPI assíncrono, o provedor de serviços age da mesma forma que no processamento de um API assíncrono. Especialmente, o provedor de serviços realiza quaisquer validações que possam ser processadas imediatamente.
  - Se algum erro for detectado, o provedor de serviços retorna o *hResult* ao *Manager*, que imediatamente o retorna ao aplicativo.
  - De outra forma, o pedido é agendado e o *hResult* WFS_SUCCESS é imediatamente retornado ao *Manager*, indicando que o pedido foi aceito e está sendo processado.
  - Após o pedido ser completado, uma mensagem de término é enviada à janela do *Manager*. Este recupera o *hResult* da estrutura apontada pela mensagem e o retorna para o aplicativo.

## 4.13 - MECANISMOS DE NOTIFICAÇÃO – REGISTRO PARA EVENTOS

As funções **WFSRegister** e **WFSDeregister** (e suas contrapartes assíncronas) são usadas para registrar e retirar o registro de procedimentos de janela destinados ao recebimento de mensagens do Windows quando eventos particulares não-solicitados e assíncronos ocorrem, mesmo durante o processamento do pedido ou a qualquer tempo. Em outras palavras, são usados para habilitar ou desabilitar o recebimento de eventos de notificação. Fornecendo notificações desse tipo aos aplicativos, a exigência de procura por *status* é removida, e um simples método de implementação de aplicativos de "monitoramento" é providenciado. Cada pedido **WFSRegister** especifica um *handle* de serviço (*hService*), uma ou mais classes de eventos e um *handle* para a janela do aplicativo (*hWnd*) destinado a

receber todas as mensagens de classes especificadas. As funções SPI correspondentes, **WFPRegister** e **WFPDeregister**, implementam as funções API.

Existem quatro classes de eventos:

- SERVICE_EVENTS
- USER_EVENTS
- SYSTEM_EVENTS
- EXECUTE_EVENTS

Para as três primeiras classes de evento, se uma classe estiver sendo monitorada e um evento ocorrer nessa classe, a mensagem é enviada para todo *hWnd* registrado para essa classe, especificando o serviço identificado pelo *hService handle*. Os eventos são gerados quando:

- O *status* do serviço muda (SERVICE_EVENTS); por exemplo, uma impressora é suspensa ou não está disponível.
- O serviço precisa que uma operação do usuário aconteça (USER_EVENTS); por exemplo, um dispositivo necessita de atenção "anormal", como acrescentar papel ou *toner* a uma impressora.
- Um evento de sistema ocorre (SYSTEM_EVENTS); por exemplo, um erro de *hardware* ocorre, uma negociação de versão falha, a rede não esta disponível ou não há mais espaço em disco disponível.

A classe EXECUTE_EVENTS é diferente das outras três. Esse tipo de eventos ocorre como uma parte normal do processamento do comando **WFSExecute**. Exemplos incluem a necessidade de interagir com ou usuário ou operador para pedir uma ação, passar um cartão magnético etc. A mensagem gerada por cada um desses eventos é enviada somente ao aplicativo que utilizou o comando **WFSExecute** que causou o evento, mesmo se outros aplicativos estiverem registrados para EXECUTE_EVENTS. Note que um aplicativo tem que explicitamente registrar para estes eventos; se ele não o fizer e tal evento ocorrer, o evento não é noticiado e o comando **WFSExecute** é completado normalmente.

A lógica do **WFPRegister** é cumulativa: para um dado serviço, o número de mensagens de notificação enviadas pode ser aumentado especificando-se classes de eventos adicionais. Como o *Manager* não guarda para quais eventos o aplicativo está registrado e a

lógica do mecanismo de registro e retirada de registro é cumulativa, o provedor de serviços é responsável por implementar a lógica desse processo.

Um aplicativo pede registro para mais de uma classe de evento em uma única chamada, utilizando um "OU" lógico:

hr = **WFSRegister** (hService, USER_EVENTS|SERVICE_EVENTS, hWnd)

Note que os serviços sempre monitoram recursos, independentemente de o aplicativo ter pedido. Usar o comando **WFSRegister** simplesmente faz com que o serviço mande notificações para o provedor de serviços, que por sua vez envia as notificações para um ou mais aplicativos.

Ao se comunicar com o *Manager*, normalmente se deseja receber mensagens em uma ou mais classes de eventos, um aplicativo pode cancelar qualquer registro prévio usando a função **WFSDeregister**. As funções **WFSRegister** e **WFSDeregister** são simétricas: o aplicativo pode cancelar o registro de uma ou mais classes de eventos monitoradas por cada janela, especificando-as de forma apropriada na lista de parâmetros. Para cancelar o registro completamente, isto é, cancelar toda classe de eventos de toda janela, um aplicativo usa a classe de evento NULL e os valores de *handle* das janelas na lista de parâmetros.

Apesar do comando **WFSDeregister** surtir efeitos imediatamente, é possível que as mensagens fiquem esperando na lista de mensagens do aplicativo. No entanto, um aplicativo robusto deve estar preparado para receber mensagens de eventos mesmo após o cancelamento do registro.

Note que uma mensagem de notificação de evento sempre passa a informação descrevendo o evento para o aplicativo apontando para uma estrutura de dados **WFSRESULT**. Após o aplicativo ter utilizado o dado na estrutura, ele deve liberar a memória que o provedor de serviços alocou para a estrutura de dados **WFSRESULT**, usando a função **WFSFreeResult**.

## 4.14 - PROCESSOS, *THREADS* E FUNÇÕES DE BLOQUEIO DE APLICATIVOS

O processo de um aplicativo contém uma ou mais *threads* de execução. A interface WOSA/XFS é projetada para trabalhar nas versões *single-threaded* do sistema operacional Windows (Windows 3.1 e Windows for Workgroups) e nas versões *multi-threaded* desse sistema (Windows NT e versões atuais). Em ambientes *single-threaded*, *thread* é sinônimo de processo. Todas referências feitas a *threads* se relacionam a sistemas atuais (ambientes Windows *multi-threaded*).

Junto ao *Manager*, uma função de bloqueio (síncrona) é tratada da seguinte maneira: o *Manager* inicia a operação, e então entra em um *loop* no qual ele despacha qualquer mensagem de Windows (deixando o processador livre outros aplicativos necessários) e checa o término da operação. Quando esta for completada, ou quando uma chamada **WFSCancelBlockingCall** é invocada, a operação de bloqueio é completada com o resultado apropriado.

Quando uma mensagem de Windows é recebida por uma *thread* para a qual uma operação de bloqueio está sendo processada, a *thread* é proibida de fazer outras chamadas WOSA/XFS durante o processamento da mensagem, salvo duas funções especificas que assistem o programador nessa situação:

- **WFSIsBlocking** determina se uma chamada de bloqueio está sendo processada.
- **WFSCancelBlockingCall** cancela uma chamada de bloqueio que está sendo processada.

Qualquer outra função chamada quando uma chamada de bloqueio já estiver sendo processada falha com o erro WFS_ERR_OP_IN_PROGRESS. Esta restrição se aplica a operações de bloqueio e desbloqueio.

Embora esse mecanismo seja suficiente para aplicativos simples, ele não pode suportar aplicativos que exijam processamento complexo de mensagem enquanto bloqueados por uma chamada síncrona, como o processamento de mensagens relacionadas a eventos MDI (*Multiple Document Interface*). Para esses aplicativos, o API WOSA/XFS inclui a função **WFSSetBlockingHook**, que permite ao programador definir uma rotina especial que será chamada ao invés da rotina ordinária de despacho de mensagem descrita. Essa função dá ao

aplicativo a habilidade de executar sua própria rotina no tempo de bloqueio, no lugar da rotina padrão. Esse não deve ser um mecanismo para executar funções gerais dos aplicativos enquanto estiverem bloqueadas; porém, as funções **WFSIsBlocking** e **WFSCancelBlockingCall** ainda são as únicas funções que podem ser chamadas de uma rotina de bloqueio. Versões assíncronas das funções WOSA/XFS têm que ser utilizadas para permitir que um aplicativo continue processando enquanto uma operação estiver em progresso.

Esse mecanismo é usado para permitir que aplicativos do Windows 3.x e do Windows for Workgroups façam chamadas de bloqueio sem bloquear o resto do sistema. Em sistemas Windows *multi-threaded*, a ação padrão de bloqueio é suspender a chamada do *thread* do aplicativo até que o pedido seja completado. Isso acontece porque o sistema não é bloqueado por um único aplicativo aguardando que uma operação seja completada, não chamando, portanto, as funções **PeekMessage** e **GetMessage**, que são necessárias em sistemas não-preemptivos para dar o controle ao aplicativo.

Logo, se um aplicativo *single-threaded* é feito para ambientes *single-* e *multi-threaded* e depende dessa funcionalidade, ele deve instalar um "gancho de bloqueio" (*blocking hook*) específico chamando a função **WFSSetBlockingHook**, mesmo que o gancho padrão seja suficiente. Isso maximiza a portabilidade dos aplicativos que dependem do comportamento do "gancho de bloqueio".

Na implementação do WOSA/XFS em ambiente *single-threaded*, a função de bloqueio funciona da seguinte maneira. Quando um aplicativo faz um pedido de uma função API de bloqueio, o *Manager* inicia a função requisitada e entra em um *loop* equivalente ao seguinte pseudo-código:

```
for (;;)  {
        /* flush messages for good user response */
        DefaultBlockingHook ( );
        /* check for WFSCancelBlockingCall */
        if ( operation_cancelled ( ) );
                break;
        /* check to see if operation completed */
        if ( operation_complete ( ) );
                break;
}
```

A rotina **DefaultBlockingHook** é equivalente a:

```
BOOL DefaultBlockingHook (void)  {
        MSG msg;
        BOOL ret;
        /* Wait for the next message */
        ret = GetMessage ( &msg, NULL, 0, 0);
        if ( (int) ret != –1)  {
                TranslateMessage (&msg);
                DispatchMessage (&msg);
        }
        /* FALSE if we got a WM_QUIT message */
        return (ret);
}
```

Em um ambiente *multi-threaded*, o desenvolvedor de um aplicativo *multi-threaded* deve estar ciente de que é da responsabilidade do aplicativo, e não do *Manager*, sincronizar o acesso ao serviço por múltiplas *threads*. Falhas na sincronização das chamadas a um serviço levam a resultados imprevisíveis. Por exemplo, se duas *threads* simultaneamente fazem chamadas **WFSExecute** para mandar dados para o mesmo serviço, não é garantida a ordem de chegada dos dados.

De forma a permitir flexibilidade máxima no *design* e implementação de aplicativos, especialmente em ambientes *multi-threaded*, o conceito identidade do aplicativo pode ser explicitamente utilizado pelo desenvolvedor usando *handles* de aplicativos.

## 4.15 - GERENCIAMENTO DE MEMÓRIA

O WOSA/XFS especifica um protocolo para a alocação e liberação dinâmica de memória. A estratégia geral é que o provedor de serviço aloca memória sempre que precisa, e o aplicativo especifica quando pode ser liberado. Isso é implementado utilizando-se uma estrutura padrão (WFSRESULT) que é sempre usada para passar informações dos serviços aos aplicativos.

A maioria das chamadas de funções do provedor de serviços é assíncrona, e retorna seus resultados via mensagem de término, que contem um ponteiro para a estrutura WFSRESULT, contendo o *status* do retorno da função (*hResult*) e dados opcionais. A desalocação da estrutura é feita da seguinte maneira:

- Funções API assíncronas

  O aplicativo recebe a estrutura do provedor de serviços via mensagem de término, e é responsável pela desalocação.

- Funções API síncronas **WFSExecute**, **WFSGetInfo** e **WFSLock**

  O *Manager* passa a estrutura WFSRESULT para o.aplicativo como um parâmetro de retorno, e o aplicativo é responsável pela desalocação, como nas chamadas assíncronas.

- Demais Funções API síncronas

  O *Manager* desempacota a informação requerida pela estrutura WFSRESULT em parâmetros de retorno para o aplicativo, desaloca a estrutura e retorna ao aplicativo.

Quatro funções são oferecidas pelo *Manager* para implementar esse protocolo: **WFMAllocateBuffer**, **WFMAllocateMore**, **WFMFreeBuffer** e **WFSFreeResult**. Com essas funções, duas políticas de alocação são suportadas: a linear e a vinculada (*linked*).

A alocação linear pode ser utilizada para qualquer estrutura de dados plana ou alocada contiguamente. Tais estruturas são retornadas em único bloco de memória alocada pela função **WFMAllocateBuffer**.

A alocação conectada pode ser usada como uma maneira eficiente de gerenciar estruturas complexas de dados, permitindo ao provedor de serviços alguma flexibilidade enquanto permite ao aplicativo liberar a estrutura inteira com uma única chamada. Nos casos em que o provedor de serviços não sabe *a priori* o tamanho do conjunto de resultados, ele faz uma estimativa inicial e usa **WFMAllocateBuffer**. Se o provedor de serviços determinar mais tarde que mais espaço é requerido pelos dados, mais quantidade de memória será requerida utilizando a função **WFMAllocateMore**, e essa nova área será automaticamente ligada ao bloco originalmente alocado. O novo bloco retornado pela função **WFMAllocateMore** não é, em geral, contíguo ao bloco raiz, e o usuário dessa função deve saber disso.

O provedor de serviços é livre para escolher qual forma de alocação é mais conveniente. Isso é completamente transparente ao aplicativo e ao *Manager*, que libera a estrutura WFSRESULT inteira com uma única chamada **WFSFreeResult**. Os aplicativos sempre devem ter certeza de liberar uma estrutura WFSRESULT retornada. Note que a estrutura WFSRESULT pode ser retornada mesmo quando o provedor de serviço tiver retornado um erro. Se nenhum WFSRESULT for retornado, o ponteiro para a estrutura é nulo. O provedor de serviço deve, inclusive, utilizar essa facilidade para suas exigências particulares de gerenciamento de memória, usar a função **WFMFreeBuffer** para liberar memória alocada.

É importante notar que os aplicativos e os provedores de serviços têm que utilizar as facilidades oferecidas pelo *Manager* para alocação e desalocação de memória para evitar conflitos de gerenciamento de memória entre os aplicativos, o *Manager* e os provedores de serviço.

O exemplo a seguir ilustra a alocação dinâmica de dois *buffers* (estrutura WFSRESULT e dados adicionais). A funcao **WFMAllocateMore** os liga automaticamente, permitindo ao aplicativo liberar ambas as estruturas com uma única chamada.

```
WFSRESULT *lpResultBuffer;

//provedor de serviços aloca o buffer da estrutura WFSRESULT

result = WFMAllocateBuffer (sizeof(WFSRESULT), ulMemFlags,
        lpResultBuffer);
...
...
...
//provedor de serviços aloca memória adicional

hr = WFMAllocateMore (evenMoreMemory, lpResultBuffer,
        &lpResultBuffer–>lpBuffer);
...
...
...
```

Depois que o aplicativo tiver obtido toda a informação que precisar do *buffer* WFSRESULT e quaisquer estruturas associadas, ele deve liberar a memória, o que é feito em uma única chamada:

```
...
...
...
//aplicativo desaloca a estrutura quando termina de utiliza-la

hr WFSFreeResult (lpResultBuffer);  //libera o buffer de resultado
                                    //e buffers adicionais
```

## 4.16 – IMPLEMENTAÇÃO DO *SERVICE PROVIDER*

Na Seção 4.2, foi visto que arquitetura WOSA/XFS prevê um provedor de serviços para fazer a comunicação com o dispositivo de hardware conectado ao PC. Já o gerenciador XFS (XFS *Manager*) e as interfaces API e SPI são obtidos gratuitamente na Internet [9].

Assim, foram desenvolvidas as funções do provedor de serviço para permitir acesso básico à placa de aquisição: conexão e desconexão do gerenciador com o provedor de serviços (**WFPOpen**, **WFPClose** e **WFPUnloadService**), acesso exclusivo à placa de aquisição (**WFPLock** e **WFPUnlock**), configuração da mesma e pedidos de conversão (**WFPExecute**).

Cada uma das funções da SPI foi criada em um módulo separado, para garantir a clareza do código. Para seguir o modelo assíncrono do provedor de serviços (Seção 4.2.3), cada um dos módulos contém uma segunda função, que é uma **thread** criada pela função principal, e que é responsável pelo processamento posterior. A seguir, uma explicação de cada um dos módulos.

### 4.16.1 – WFPOPEN.C

A função **WFPOpen** é a primeira chamada pelo XFS *Manager*, e é responsável por realizar a conexão deste com o provedor de serviços. **WFPOpen** inicia chamando as funções **GetWindowInfo** e **GetLastError** da API do Windows, para saber se o parâmetro "hWnd" de entrada é um *handle* válido ("hWnd" é o *handle* da janela que receberá a mensagem de finalização da função). Em seguida, são feitas negociações de versão do provedor de serviços e da SPI, para garantir que o aplicativo não está utilizando uma versão muito baixa ou muito alta destes. Caso nenhum desses erros tiver sido encontrado, o *handle* de serviço "hService" (isto é, o identificador da sessão aberta com o XFS *Manager*) é armazenado, e uma estrutura do tipo WFSRESULT é alocada dinamicamente para ser enviada junto com a mensagem de finalização (seguindo o gerenciamento de memória proposto no Seção 4.2.13). Finalmente, a *thread* de finalização (**OpenMessage**) é criada, e a função retorna o valor WFS_SUCCESS.

A função **OpenMessage** só é atingida se não houve erros de negociação, portanto ela não realiza processamento posterior, simplesmente enviando a mensagem de finalização da função (WFS_OPEN_COMPLETE) e a estrutura de resultado (WFSRESULT) através da função **SendMessage** da API do Windows.

### 4.16.2 – WFPCLOSE.C

Esta é a função complementar à função **WFPOpen**. Ela é responsável por fechar a sessão com o provedor de serviços, apagando o *handle* de serviço da sessão. Caso este *handle* seja válido, o *handle* "hWnd" é analisado da mesma forma que na função **WFPOpen**. Caso não seja encontrado nenhum erro, uma estrutura de resultado é alocada dinamicamente, a *thread* de finalização é criada (**CloseMessage**) e o a função retorna WFS_SUCCESS.

A função **CloseMessage** funciona analogamente à função **OpenMessage**, sendo que no final, a função **WFMReleaseDLL** é chamada, indicando que o gerenciador XFS pode desalocar o provedor de serviços.

### 4.16.3 – WFPLOCK.C

A função **WFPLock** simplesmente testa se os *handles* de serviço e de janela ("hService" e "hWnd", respectivamente) são válidos, e em caso afirmativo, aloca dinamicamente uma estrutura de resultado e cria a *thread* de finalização da função, **LockMessage**. Nesta, a função **WaitForSingleObject** da API do Windows é usada para tentar conseguir acesso ao semáforo criado quando o provedor de serviços é carregado (módulo DLLMain.c). Se o acesso for garantido, o *handle* da porta serial (tópico que é explicado mais minuciosamente na sessão 5.1.2) é garantido, a porta é configurada, a variável global "iLock" é feita igual a 1 (sinalizando o acesso exclusivo), e o valor WFS_SUCCESS é enviado na mensagem de finalização; se o acesso não for obtido dentro do tempo determinado (variável de entrada "dwTimeOut" da função **WFPLock**), é retornado o erro WFS_ERR_TIMEOUT dentro da mensagem de finalização.

### 4.16.4 – WFPUNLOCK.C

Como função complementar à **WFPLock**, esta função executa o mesmo processamento relativo a *handles*, e cria uma thread (**UnlockMessage**) se não houver erros. Dentro desta, o semáforo é liberado (função **ReleaseSemaphore** da API do Windows) se a variável "iLock" tiver sido sinalizada anteriormente pela chamada da função **WFPLock**, e o acesso à porta serial é abandonado pela liberação do *handle* da mesma. Caso "iLock" não

tiver sido sinalizada, o erro WFS_ERR_NOT_LOCKED é mandado como resultado na mensagem de finalização.

### 4.16.5 – WFPEXECUTE.C

Está a função principal do provedor de serviços, que realiza o acesso ao hardware. Inicialmente, ela também procura erros nos *handles* enviados nas variáveis de entrada ("hService" e "hWnd"); se não houver nenhum erro, a estrutura de resultado é alocada, e é criada a *thread* de finalização, a função **ExecuteMessage**.

Nesta função, a variável "iLock" é testada para saber se o acesso à placa de aquisição foi garantido. Ou seja, só é possível utilizar a placa se o aplicativo chamou a função **WFSLock** ou **WFSAsyncLock** com sucesso anteriormente.

Se o acesso tiver sido garantido, a variável de entrada "dwCommand" é testada para saber para que finalidade a função **WFPExecute** foi chamada. No caso WFS_CMD_MSP_CONFIG (definido na interface XFSMSP.h), deseja-se fazer a configuração da placa de aquisição, e no caso WFS_CMD_MSP_CONVERT, deseja-se iniciar o processo de conversão. Na chamada da função **WFSExecute**, uma das variáveis de entrada é um ponteiro para uma estrutura WFSMSPCONFIG, que contém um vetor com as taxas de amostragem desejadas e uma variável com o número de bytes que se deseja obter na conversão A/D (número este que é o dobro do número total de amostras desejadas). Por exemplo, para utilizar o canal 0 a 4KHz e o canal 1 a 2KHz, temos o vetor {4000, 2000, 0, 0, 0, 0, 0, 0}, e para obtermos 15 amostras, a variável "amostras" vale 15. Essa informação deve ser processada para ser enviada para o microcontrolador. Para compreendermos essa parte do código, façamos uma breve explicação do protocolo de comunicação.

#### 4.16.5.1 – PROTOCOLO DE COMUNICAÇÃO COM O MICROCONTROLADOR MSP430

Para configurar o microcontrolador, foi escolhida uma palavra de 13 bytes a ser enviada pelo PC. O primeiro byte deve ser sempre igual ao START_BYTE (21h), para evitar a leitura de informação espúria por parte do microcontrolador. O segundo byte deve ser igual ao CONFIG_BYTE (AAh), para diferenciar o pacote de configuração daquele enviado no pedido de conversão.

Para gerar as taxas de conversão, escolhemos a taxa mais alta para ser contada pelo contador/temporizador do microcontrolador, de modo que todos os sinais são amostrados a esta velocidade. As outras taxas são geradas ignorando-se os dados obtidos e obtendo-se taxas correspondentes. Por exemplo, para obtermos uma taxa de 4KHz no canal 0 e 2 Khz no canal 1, enviamos o valor 8MHz/4KHz = 2000 (relógio do microcontrolador dividido pela maior taxa) para a contagem no microcontrolador, e a cada duas amostras armazenadas do canal 0, guardamos uma amostra do canal 1. O valor de contagem correspondente à taxa mais alta de conversão vem no terceiro e quarto bytes, sendo esse último o mais significativo. Do quinto até o décimo-segundo bytes, temos o número de amostras mais um que devemos ignorar para cada canal (canal 0 – quinto byte; canal 7 – décimo-segundo byte).

O último byte é um cálculo simples de CRC (Cyclic Redudancy Check) para garantir integridade nos dados. Para isso, são somados os primeiros doze bytes, e considerados apenas os 8 bits menos significativos dessa soma.

Para fazer um pedido de conversão, enviamos um pacote de 13 bytes, com o primeiro byte igual ao START_BYTE, o segundo igual ao CONVERT_BYTE (33h), o terceiro e o quarto bytes (LSB e MSB, respectivamente) representando o número de bytes gerados na conversão (por exemplo, 4 amostras geram 8 bytes), e o último byte igual ao cálculo de CRC do pacote.

Voltando ao código da função **ExecuteMessage**, comecemos pelo caso em que “dwCommand” é igual a WFS_CMD_MSP_CONFIG. O vetor de taxas de amostragens é analisado para saber qual é a taxa mais alta, e a partir dela, os valores do terceiro e quarto bytes do pacote de configuração são definidos. Comparando a taxa mais alta com as outras, os valores dos bytes 5 a 12 do pacote são definidos. O último byte é definido pelo cálculo do CRC, e o pacote é enviado para o microcontrolador, com uso da função **WriteFile** (sessão 5.1.3). Finalmente, a mensagem de finalização é enviada.

Quando “dwCommand” vale WFS_CMD_MSP_CONVERT, o número de amostras é enviado no pacote de 13 bytes (incluindo bytes de início, de pedido de conversão e de

CRC). Em seguida, um buffer com o número de amostras é alocado dinamicamente, e a leitura dos dados é feita (**ReadFile** – sessão 5.1.3). Quando terminado o processo, é enviada uma mensagem de finalização.

### 4.16.6 – WFPUNLOADSERVICE.C

Esta função é imediata, e só indica para o gerenciador XFS que o provedor de serviços continua pronto para ser desalocado.

### 4.16.7 – DLLMAIN.C

Este é o módulo mais simples. Nele são declaradas as variáveis globais do provedor de serviços. Além disso, temos a função **DllMain**, que é chamada no momento em que o provedor de serviços é carregado pelo XFS *Manager*, e dentro dessa função, o semáforo utilizado nas funções **WFPLock** e **WFPUnlock** é criado (através da função **CreateSemaphore** da API do Windows).

# CAPÍTULO 5

# HARDWARE: PORTA SERIAL DO PC E O MICROCONTROLADOR MSP430

No capítulo anterior, foi detalhado o *software* que implementa o módulo de fornecimento de dados ao aplicativo, bem como a arquitetura utilizada para esse *software*. Neste capítulo, veremos os detalhes dos outros dois módulos do sistema. Para isso, descreveremos o funcionamento da comunicação serial, utilizada no módulo de tráfego de dados, e do microcontrolador MSP430, utilizado nos módulos de tráfego e de captura de dados.

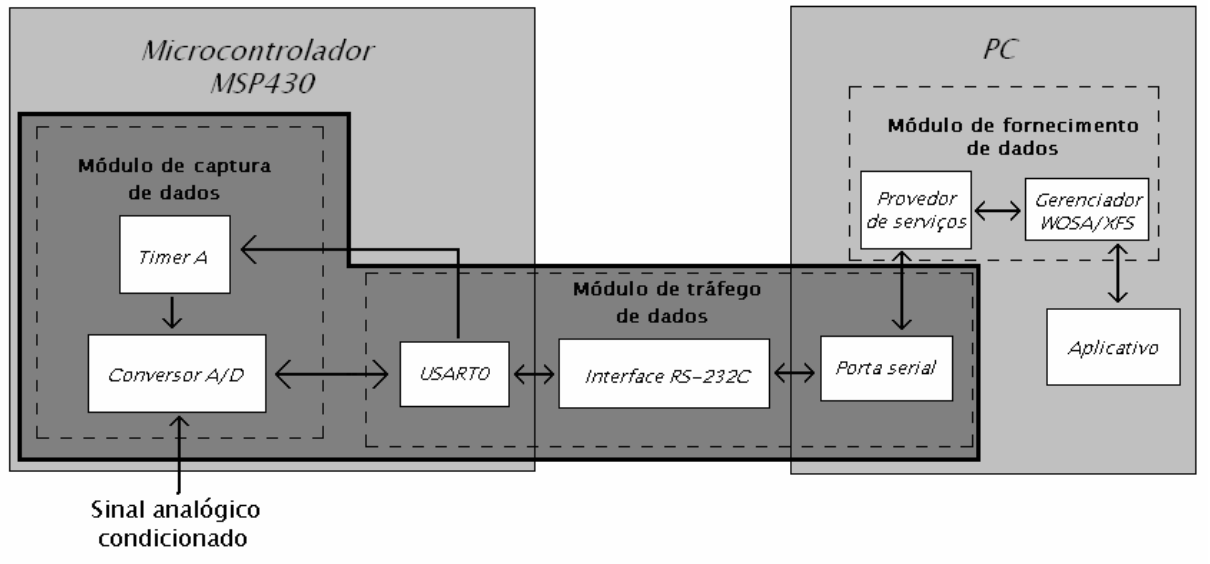


*Figura 7 – Módulos descritos no Capítulo 5*

## 5.1 COMUNICAÇÃO SERIAL

A porta serial é uma interface física comunicada ao PC que transfere dados um bit por vez, ao contrário da porta paralela, que transmite um grupo de bits simultaneamente. Com isso, a porta serial oferece um circuito mais simples para comunicação, pois necessita de menos conexões, e cabos mais longos, visto que a comunicação serial usa níveis de tensão que garantem uma variação maior da mesma. Em compensação, o envio serial de dados é mais

lento, pois mais bits são enviados paralelamente, e ele necessita de mais registradores para configuração.

### 5.1.1 INTERFACE RS-232C

Este padrão foi criado pelo Eletronic Industries Alliance (EIA), e data de 1969. Ele cobre características mecânicas, elétricas e funcionais da interface. Quaisquer dispositivos que trabalhem com níveis TTL de tensão devem usar circuitos para converter para os níveis RS-232C, e vice-versa.

As linhas RS-232C realizam enlaces de comunicação para uma distância de até 15,24m, com uma taxa máxima de 20Kb/s. Os níveis de tensão correspondentes aos níveis TTL são os seguintes:

Tabela 5.1 – Conversão de níveis TTL – RS-232C

| | Nível alto | Nível baixo |
|---|---|---|
| TTL | +5 V | 0 V |
| RS-232C | – 12 V | +12 V |

Na verdade, qualquer sinal entre –3 V e –15 V é interpretado como nível alto no padrão RS-232C, e qualquer sinal entre +3 V e 15 V é interpretado como nível baixo. Níveis fora dessa duas faixas são considerados inválidos.

Um circuito bastante comum na conversão de níveis é o *transceiver* HIN232 [23]. Ele atende ao padrão RS-232C com uma alimentação simples de +5 V, e tem entradas compatíveis com níveis TTL e CMOS. A Figura 6 mostra o esquemático do circuito.

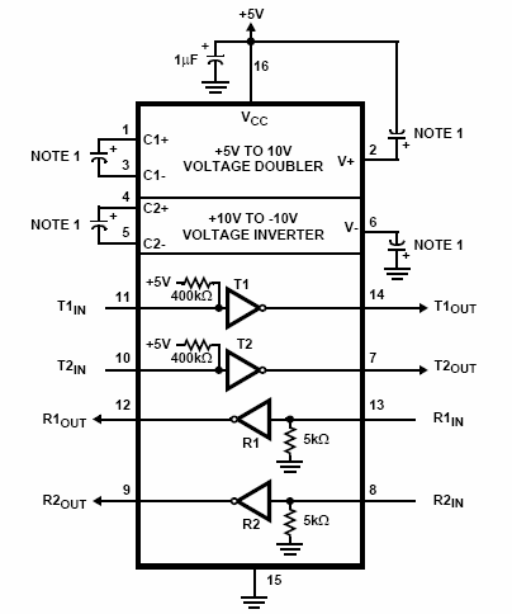


*Figura 8 – Circuito de conexão do transceiver HIN232*

### 5.1.2 COMUNICAÇÃO SERIAL NO PC

O PC oferece comunicação serial assíncrona, onde não existe um relógio determinado governando a transferência de dados. O circuito 8250 (ou equivalente) é responsável pela comunicação serial e é programável, determinando a quantidade de *bits* de partida, de paridade, de dados e de parada, além da taxa de comunicação. É possível enviar caracteres de 5, 6, 7 ou 8 *bits*, com 1, 1½ ou 2 *bits* de parada.

A porta serial COM1 (utilizada no projeto) tem seu circuito 8250 mapeado no endereço 3F8h. Ele insere os *bits* de partida, parada e paridade na transmissão, bastando ao aplicativo escrever ou ler dados em registradores de transmissão e recepção, respectivamente. O controlador 8250 possui 11 registradores para programação e utilização do protocolo RS-232C.

### 5.1.3 ACESSO Á PORTA SERIAL DO PC

Anteriormente ao sistema operacional Windows, era relativamente fácil acessar portas I/O em um PC típico, pois qualquer linguagem possuía comandos específicos para fazê-lo. Com a implantação e posterior evolução do Windows, isso não mais poderia ser feito devido à sua habilidade em virtualizar o hardware. Isso significa que quando um aplicativo acredita estar se comunicando diretamente com um dispositivo físico, na verdade ele está se

comunicando com um *driver* que emula esse hardware, transferindo dados da forma apropriada. O Windows 95/98 permite que sejam executadas operações de I/O no nível da aplicação; tipicamente esse tipo de controle em baixo nível é feito em linguagem *assembly*.

Já o Windows NT/2000/XP possui uma política de segurança que não permite que se realizem operações de I/O no nível de aplicação. Entretanto, é permitido o uso de instruções de I/O em *drivers* no modo *kernel*. Um *driver* em modo *kernel* roda no mais privilegiado nível do processador, e pode fazê-lo sempre que quiser. O problema é que um outro *driver* nesse mesmo modo pode retirar o acesso às portas de I/O do aplicativo. Deve então ser possível para um outro *driver* em modo *kernel* retomar esse acesso.

A API do Windows define que interfaces físicas como as portas serial e paralela são recursos de comunicação. Mais especificamente, recursos de comunicação são dispositivos lógicos ou físicos que oferecem comunicação assíncrona bidirecional. Para cada um deles existe um provedor de serviços que acessa o recurso (uma biblioteca ou *driver*), análogo à arquitetura WOSA/XFS. Assim, o acesso à porta serial pode ser feito por meio da API do Windows.

Para que tal acesso possa ser feito, inicialmente, um processo deve criar um *handle* para a porta serial (ou qualquer outro recurso de comunicação) através da função **CreateFile**. Este *handle* identifica a porta. Se outro processo ou *thread* estiver acessando a porta serial, esta função falha. Aberto o *handle*, pode-se utilizar a função **GetCommState** para determinar a configuração inicial do circuito 8250, controlador da porta serial. Através da função **SetCommState**, pode-se modificar a configuração do mesmo. Finalmente, as funções **ReadFile** e **WriteFile** são utilizadas para a leitura e escrita de dados pela porta serial.

## 5.2 O MICROCONTROLADOR MSP430

Um microcontrolador é uma espécie de microprocessador com periféricos de interesse (conversor A/D, porta serial, entre outros) e possivelmente memória, todos integrados no mesmo *chip*. Uma arquitetura desse tipo é voltada para o controle de diversos dispositivos, atendendo a diferentes aplicações.

O microcontrolador da linha MSP430, produzido pela *Texas Instruments*, possui uma CPU RISC de 16 bits, periféricos analógicos e digitais e um sistema de relógio flexível, todos conectados por barramentos von-Neumann de memória de endereços e memória de dados. A família MSP430x1xx utilizada inclui: arquitetura de baixíssimo consumo de energia; conversor analógico-digital de 10 ou 12 bits, com gerador interno de tensão de referência e supervisor de tensão de alimentação; grande número de registradores e pequeno número de instruções, permitindo otimização para programação em alto nível; e memória *flash* programável, que garante mudanças no código em campo e acesso aos dados.

O sistema de memória da linha MPS430 é formado por um espaço único de memória de 64 KB, onde se encontram registradores de funções especiais (SFRs), periféricos, memória RAM e memória *flash*/ROM. O acesso é feito em endereços pares, podendo ser de 8 ou 16 bits (*bytes* e *words*, respectivamente).

O MSP430 é adquirido através de amostra grátis do fabricante, e possui emulação embarcada no próprio dispositivo, sem necessidade de recursos adicionais do sistema. A compilação e o *debug* de projetos são feitos através de uma interface JTAG (*Joint Test Action Group*) e de um *software* dedicado (*IAR Embedded Workbench*), possibilitando o monitoramento das variáveis do programa e dos registradores do microcontrolador em tempo real. Dessa forma, a detecção de erros é facilitada, constituindo uma poderosa ferramenta no desenvolvimento de projetos.

### 5.2.1 DISPOSITIVOS DE ENTRADA/SAÍDA (I/O)

A linha MSP430x14x conta com seis portas de entrada e saída de dados (portas de I/O P1 a P6), todas com 8 pinos configuráveis individualmente. As portas P1 e P2 podem ser configuradas para causar interrupções em borda de subida ou de descida. A configuração e o uso das portas são feitos por software, através da leitura e escrita nos seguintes registradores: PxDIR, PxIN, PxOUT, PxSEL, P1IE, P2IE, P1IES, P2IES, P1IFG e P2IFG.

Cada *bit* nos registradores PxDIR indica qual a direção de tráfego de sinais para o pino correspondente da porta (por exemplo, o pino 0 de P1DIR indica se o pino P1.0 será de entrada ou de saída). A seis portas possuem registradores PxDIR (P1DIR a P6DIR). A direção selecionada para o pino independe da função determinada para o mesmo. Para configurar o

pino como entrada, deve-se levar o *bit* correspondente em PxDIR para nível baixo, e para configurar para saída, deve-se levar o *bit* para nível alto.

Os pinos das seis portas do MSP430 são multiplexados com outros periféricos encontrados no microcontrolador. Por exemplo, o pino P6.0 pode ser usado como I/O ou como uma das entradas do conversor A/D. Para selecionar a funcionalidade desejada para o pino, escreve-se no registrador PxSEL correspondente (P1SEL a P6SEL), sendo que o nível lógico 0 indica que o pino será usado para I/O, e o nível lógico 1 indica que o o pino será usado para o periférico designado pelo fabricante.

Os registradores PxSEL possuem duas particularidades. Primeiro, quando o pino é selecionado para trabalhar com o periférico correspondente, deve-se ajustar a direção de tráfego de sinal para o pino (através de PxDIR), pois ela não é escolhida automaticamente. Segundo, para cada *bit* em nível alto nos registradores P1SEL e P2SEL, são desabilitadas as interrupções dos pinos correspondentes. Por exemplo, utilizando o periférico designado para o pino P1.0, este não será capaz de causar interrupções devido a I/O.

### 5.2.2 CONVERSOR ANALÓGICO DIGITAL

Dentre os perifféricos presentes na família MSP430x14x, tem-se o conversor analógico-digital de 12 bits, o ADC12. Ele realiza e armazena até 16 conversões A/D sem intervenção da CPU, e possui gerador de tensão de referência, sensor de temperatura integrado no *chip* e 16 registradores de conversão e controle. Algumas características do ADC12 são:

- Taxa de conversão superior a 200 mil amostras por segundo;
- Velocidade programável (otimizando a relação entre consumo e velocidade de conversão);
- Conversão iniciada por software, pelo *Timer* A (seção 4.5) ou *Timer* B;
- Seleção por *software* de referência interna ou externa;
- Quatro diferentes modos de conversão: único canal, único canal repetidamente, seqüência de canais e seqüência repetida de canais;
- *Sample-and-hold* com tempo de amostragem programável controlado por software ou por temporizador;

- Oito canais de conversão que podem ser configurados individualmente;
- Registrador de interrupções, que indica qual dos canais gerou interrupção no conversor.

O processo de conversão possui limites de tensão superior e inferior ($V_{R+}$ e $V_{R-}$), dentro dos quais opera o ADC12. Os limites são configuráveis por *software*. Uma leitura digital máxima (para 12 *bits*, 0FFFh) indica que o sinal de entrada foi igual a $V_{R+}$, e uma leitura mínima (0000h) indica um sinal de entrada igual $V_{R-}$. O resultado da conversão ($N_{ADC}$) pode ser expresso pela seguinte fórmula:

$$N_{ADC} = 4095 \cdot \frac{(Tensão\ de\ Entrada) - V_{R-}}{V_{R+} - V_{R-}} \quad (1)$$

O sinal ADC12CLK temporiza a conversão, gerando o período de amostragem necessário. Este sinal é obtido de 4 possíveis fontes de relógio: ACLK, MCLK, SMCLK ou um oscilador interno, ADC12OSC. A fonte escolhida pode ser dividida por 1, 2, 4 ou 8, gerando o sinal ADC12CLK. A figura 4.1 ilustra a configuração deste sinal.

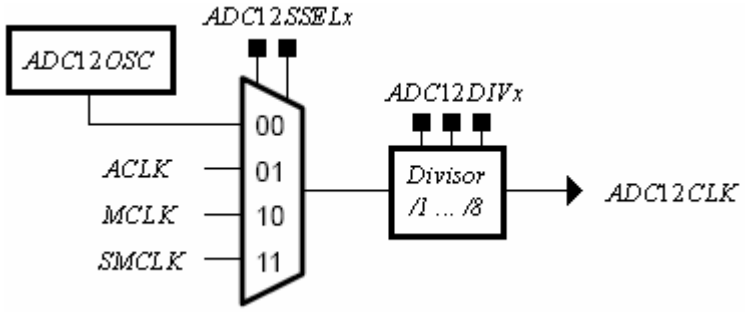


*Figura 9 – Seleção do relógio do conversor analógico digital*

O núcleo do ADC12 pode ser configurado pelos registradores ADC12CTL0, ADC12CTL1, ADC12MCTLx e ADC12MEMx. A seguir, temos uma descrição dos itens mais importantes de cada um.

### 5.2.3 REGISTRADORES DE CONTROLE DO CONVERSOR ANALÓGICO DIGITAL

O registrador ADC12TL0 possui 15 bits. A denominação de cada um deles é ilustrada abaixo.

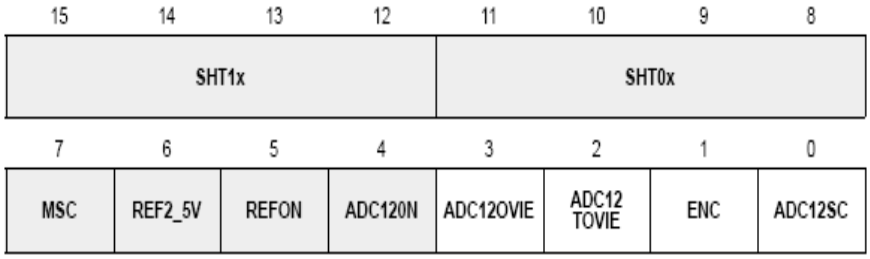


*Figura 10 – O registrador ADC12CTL0*

Os *bits* SHT1x definem o número de ciclos do sinal de relógio do conversor (ADC12CLK) que serão utilizados para realizar a amostragem nas memórias 8 a 15 do ADC12. Os *bits* SHT0x têm a mesma função para as memórias 0 a 7 do ADC12. O *bit* MSC permite múltipla amostragem e conversão. O *bit* REFON habilita a geração de tensão de referência, e o *bit* REF2_5 indica se o gerador de tensão de referência será de 1,5 V ou 2,5 V. O *bit* ADC12ON habilita o ADC12, o *bit* ENC (*Enable conversion*) habilita o processo de conversão, e o *bit* ADC12SC (*Start Conversion*) inicia o processo de conversão. Os *bits* marcados com a cor cinza só podem ser modificados quando ENC = 0.

A denominação dos *bits* do registrador ADC12CTL1 pode ser acompanhada na ilustração abaixo.


*Figura 11 – O registrador ADC12CTL1*

Os *bits* CSTARTADDx (*Conversion Start Adress*) determinam qual registrador de memória do ADC12 será utilizado no caso de um único canal, ou o primeiro registrador no caso de uma seqüência de canais. Os *bits* ADC12SSELx (*ADC12 Clock Source Select*) selecionam a fonte do relógio usada pelo sinal ADC12CLK. Os *bits* ADC12DIVx (*Clock Divider*) determinam o divisor do relógio usado na conversão para gerar o sinal ADC12CLK.

Os *bits* CONSEQx selecionam o modo de conversões (00 = um único canal, 01 = um único canal repetidamente, 10 = uma seqüência de canais e 11 = seqüência repetida de canais). O *bit* ADC12BUSY indica que um processo de amostragem e conversão em andamento. Novamente, os *bits* marcados com a cor cinza só podem ser modificados quando ENC = 0.

A denominação dos *bits* do registrador ADC12MCTLx pode ser acompanhada na ilustração abaixo.

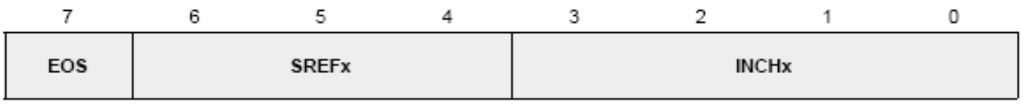


*Figura 12 – O registrador ADC12MCTLx*

O *bit* EOS (*End of conversion*) indica qual será o o último canal amostrado em uma seqüência de canais. Os *bits* SREFx indicam as referências de tensão para cada canal. Os *bits* INCHx determinam a seleção do canal correspondente à memória em questão. O registrador só pode ser modificado se ENC = 1.

Os registradores ADC12MEMx são justamente os responsáveis pelo armazenamento de dados advindos do processo de conversão. Eles possuem 16 *bits*, mas como o conversor tem resolução de 12 *bits*, os *bits* 12 a 15 não são utilizados, permanecendo em zero.

### 5.2.4 O CONTADOR/TEMPORIZADOR TIMER A

O *Timer A* é um contador/temporizador de 16 *bits*, assíncrono, com quatro modos de operação. Ele funciona a partir de diversas fontes de clock interno e externo, selecionáveis por *software*, e tem capacidade de interrupção. Pode operar nos modos de contagem progressiva/regressiva, captura (medição do período de sinais), comparação (geração de pulsos que possuem largura programável) e PWM (geração de sinais de freqüência e ciclo ativo programáveis). O *Timer A* pode trabalhar em conjunto com outros periféricos, podendo, por exemplo, iniciar uma conversão A/D.

O princípio de operação é simples. O registrador contador de 16 *bits* TAR (*Timer A Register*) é incrementado ou decrementado de acordo com o modo de operação, na borda de subida ou de descida de um sinal de relógio. Este pode ser selecionado entre quatro diferentes fontes: TACLK (clock externo), ACLK, SMCLK ou INCLK.

O registrador TAR pode ser acessado por *software*, e também pode gerar interrupção, quando este excede o seu limite de contagem. O modo de operação do TAR é definido através dos *bits* MCx do registrador TACTL. Quando iguais a 00h, a contagem do TAR é congelada, mantendo o último valor. Quando iguais a 01h, o TAR conta de 0 até o valor guardado no registrador TACCR0, reiniciando em 0. Quando iguais a 10h, o TAR é incrementado repetidamente de 0 a 0FFFFh (valor máximo de contagem). Quando iguais a 11h, o contador TAR vai de 0 até o valor em TACCR0, e é decrementado de volta até 0, gerando um período com o dobro de TACCR0.

A denominação dos *bits* do registrador TACTL é ilustrada abaixo.

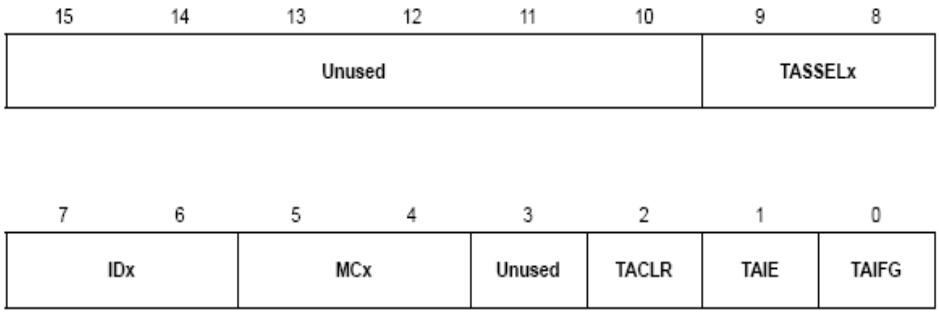


*Figura 13 – O registrador TACTL*

Os *bits* TASSELx determinam a fonte de relógio do *Timer A*. Os *bits* IDx determinam o divisor da fonte de relógio. A funcionalidade dos *bits* MCx já foi descrita acima.

### 5.2.5 INTERFACE SERIAL UNIVERSAL

O MSP430 oferece dois módulos independentes de interface serial síncrona/assíncrona universal (USART0 e USART1). Através deles, pode-se comunicar o microcontrolador com dispositivos, como computadores e outros microcontroladores.

Os dois módulos de comunicação serial do MSP 430 permitem a troca de pacotes seriais de 7 ou 8 *bits* de dados, geração e detecção de paridade e seleção do número de *bits* de parada (1 ou 2). Para cada módulo, existe um registrador de deslocamento para a transmissão e recepção dos dados, e um circuito gerador de relógio (*baud-rate*) para diversas velocidades de comunicação. A USART também oferece funcionalidades extras, como detecção de erros de comunicação e suporte a endereçamento para sistemas multiprocessados.

#### 5.2.5.1 MODO ASSÍNCRONO

Através deste modo de operação da USART, pode-se enviar dados do microcontrolador para o computador, desde que ambos concordem sobre o formato do pacote serial e a taxa de transmissão.

O pacote serial, no modo assíncrono, começa com um *bit* de partida (*Start*), seguido pelo *byte* a ser enviado (começando pelo *bit* menos significativo do caractere, o LSB), um *bit* de endereçamento e um de paridade (opcionais), e os *bits* de parada (1 ou 2). A USART pode trabalhar tanto com 7 quanto com 8 *bits*. Todas essas opções são definidas por *software*, através da escrita no registrador UxCTL.

A taxa de transmissão é definida nos registradores UxBR0 e UxBR1, onde o último é o *byte* mais significativo. O número formado por esses dois registradores divide o relógio escolhido por *software* para o módulo serial, gerando a *baud rate* desejada. Os registradores UxRXBUF e UxTXBUF são responsáveis pela leitura e escrita de dados seriais, respectivamente. Nos registradores ME1 e ME2, habilitam-se a transmissão e a recepção de cada um dos módulos seriais. Finalmente, os registradores IE1 e IE2 habilitam as interrupções causadas pela chegada ou pelo envio de dados nas USARTs 0 e 1.

### 5.2.6 PROGRAMAÇÃO DO MICROCONTROLADOR MSP430

As funções de execução do provedor de serviço (**WFPExecute**) recebem dados do aplicativo, que indicam as taxas de amostragem desejadas e pedidos de conversão. Basicamente, o microcontrolador MSP430 é responsável por receber esses dados de configuração e conversão do PC pela porta serial, processá-los, converter os sinais fornecidos nos canais de A/D e enviá-los pela porta serial.

O relógio utilizado para o MSP430 provém de um cristal externo de 8 MHz. Além disso, foi selecionada a USART0 para comunicar com a porta serial do PC. A porta P6 recebe os 8 canais de conversão A/D. Essas definições dos pinos encontram-se no *header* Definicoes.c (Anexo x), bem como as definições de *bytes* do protocolo.

No módulo Funcoes.c (também em anexo), temos uma pequena biblioteca para configurar os principais periféricos usados no projeto. A seguir, uma explicação mais detalhada deste arquivo.

#### 5.2.6.1 O ARQUIVO *FUNCOES.C*

As funções definidas nessa biblioteca são **Configurar_Clock**, **Configurar_AD_1**, **Configurar_TimerA**, **Configurar_UART** e **Configurar_AD_2**. Com exceção da última, essas funções são chamadas logo após feito o *boot* do microcontrolador. A função **Configurar_Clock** segue orientações do *User's Guide* do MSP430 para configurá-lo para utilizar um cristal externo, de 8 MHz, no caso. Assim, a função liga o oscilador externo do microcontrolador, testa por 50 µs se existe um sinal de relógio (através do *flag* OFIFG, que detecta falta de sinal de relógio), e se funcionar, configura o relógio para operar a 8 MHz, a partir do cristal externo.

A função **Configurar_AD_1** define aspectos da operação do ADC12 que independem das taxa de conversão escolhidas pelo usuário, como, por exemplo, o tempo de conversão, de modo que o *Timer A* inicie as conversões e estas sejam concluídas o mais rápido possível. Além disso, ela define que o usuário da placa de aquisição deverá escolher do pino P6.0 em diante para receber os dados (no caso de 7 canais, por exemplo, o usuário deverá usar os pinos P6.0 a P6.6). A função **Configurar_Timer_A** define que o *Timer A* deverá trabalhar com um relógio de 8 MHz e gerar interrupções. A função **Configurar_UART** define um pacote serial com 8 *bits* de dados, um *bit* de parada e sem *bits* de paridade e de endereço, gera uma *baud rate* de aproximadamente 115200 bps, habilita a transmissão e a recepção de dados, além da interrupção causada pela recepção.

A função **Configurar_AD_2** é chamada depois que o PC já mandou uma palavra de configuração. Seguindo o protocolo definido na seção 4.5.1, a função joga no registrador TACCR0 o valor dado pelo terceiro (LSB) e quarto (MSB) *bytes* do pacote de configuração. Em seguida, é calculado o número de canais utilizado, a fim de escolher os pinos

correspondentes para funcionarem como entrada analógica, e não como I/O. É guardado também na variável "int_AD" (passada por referência) o valor do último canal, que irá gerar a interrupção do ADC12. Em seguida, o canal 0 do ADC12 é atribuído à memória 0, o canal 1 para a memória 1, e assim por diante até o último canal usado. Finalmente, o vetor fim_contagem[8] recebe o valor das contagens de cada um dos canais, em função do canal de maior taxa.

#### 5.2.6.2 O ARQUIVO *PROGRAMA FINAL.C*

Neste módulo estão inseridos as variáveis globais, o laço principal e as interrupções chamadas pela porta P1, pelo *Timer A* e pelo ADC12. Dentre as variáveis globais, temos quatro *buffers*: um de 13 *bytes* para receber palavras de configuração (conf[PAC_CONF]); um *buffer* circular de 1000 *bytes* para armazenar os dados obtidos pelo conversor (pacote[PAC_DATA]); e dois *buffers* de oito *bytes* para guardar as contagens atual e final, que decidem quando armazenar os dados dos 8 canais. Além disso, definem-se ponteiros para conf[PAC_CONF] e pacote[PAC_DATA]: *pConf, de escrita em conf[PAC_CONF], e *pPacote e *pEnvio, para escrita e leitura em pacote[PAC_DATA], respectivamente. Em seguida, são chamadas as funções de configuração dos periféricos usados no projeto, e o programa entra em um laço infinito que analisa os ponteiros usados.

Na primeira condição analisada, é verificado se o primeiro *byte* de configuração recebido é igual ao START_BYTE. Na segunda condição, é verificado se um pacote de configuração foi recebido, e se for o caso, é calculado o CRC correspondente e comparado com o CRC enviado. Se ambos se igualarem, o segundo *byte* é analisado para saber se foi mandado um pacote de configuração ou de pedido de conversão, e são tomadas as devidas providências.

Na terceira condição analisada no laço infinito, os ponteiros pPacote e pEnvio são comparados, pois se as conversões já começaram, o primeiro estará à frente do segundo, e se a variável "amostras" for nula, isso indica que completou-se o pedido de conversão. Nesse caso, o microcontrolador envia serialmente os dados para o PC.

A rotina de interrupção da recepção na USART é bem simples, simplesmente armazenando o *byte* recebido na posição apontada pelo ponteiro pConf. A análise do que é recebido fica por conta laço principal.

Na rotina de interrupção do ADC12, realizam-se laços condicionais para saber se os dados convertidos devem ser armazenados ou não (*buffer* contagem[8]), e para saber se o processo de conversão A/D deve ser terminado (variável global "amostras"). Se os dados forem guardados, o ponteiro pPacote e a variável "amostras" são atualizados.

A rotina de interrupção do *Timer A* é a mais simples. Pelo vetor de interrupções TAIV (*Timer_A Interrupt Vector Register*), é possível saber se a rotina foi chamada devido ao *overflow* do *timer* (TAIV = 0Ah). Nesse caso, o registrador TACCR0 é atualizado, e o processo de conversão é iniciado.

## 5.3 O CIRCUITO DE CONEXÃO MSP430 – PORTA SERIAL

A conversão de níveis de tensão CMOS para RS-232C e vice-versa foi feita pelo *transceiver* HIN232. Como o MSP430 trabalha com baixos níveis de tensão, chegando a suportar no máximo 4,3 V [24], foi inserido um divisor de tensão na linha que chega na recepção da USART0 do microcontrolador (pino P3.5), gerando um nível de 2,5V. De resto, o circuito é o mesmo que aquele mostrado na Figura 8**.**

# CAPÍTULO 6

# APLICATIVO E RESULTADOS PRÁTICOS

Concluídas as camadas física e de acesso ao *hardware*, deve-se então criar um aplicativo para chamar as funções oferecidas na API da arquitetura WOSA/XFS, de modo a verificar o real funcionamento do sistema. Para tal, foi criado um aplicativo com interface amigável, que ilustra a utilização das funções básicas de abertura de sessão e de acesso exclusivo à placa de aquisição. Nele estão inseridos *displays* gráficos onde são plotadas as amostras obtidas dos sinais de entrada.

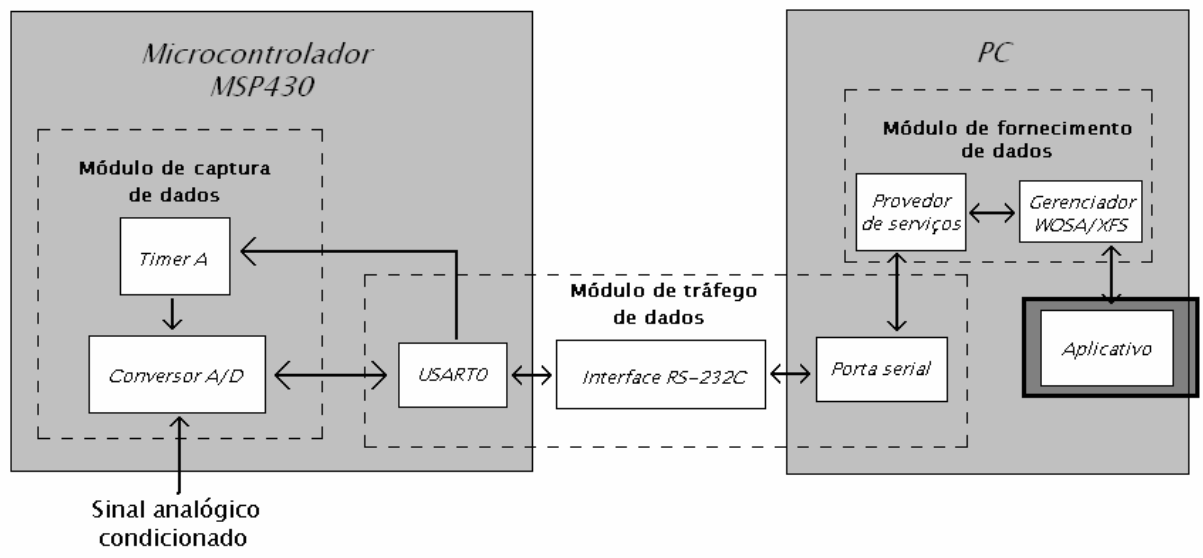


*Figura 14 – Seção descrita no Capítulo 6*

## 6.1 APLICATIVO

O programa contém uma única janela, que pode ser vista na Figura 6.1. Os botões da esquerda realizam as chamadas das funções **WFSStartup**, **WFSOpen**, **WFSLock**, **WFSUnlock**, **WFSClose** e **WFSCleanUp**, apresentando caixas de mensagens de acordo com o resultado das funções. Nessas caixas, verificam-se mensagens indicativas de sucesso ou erro, seja por causa de pedidos feitos antes mesmo de a sessão ser aberta, seja por causa de falhas internas do sistema XFS (na desalocação de memória, por exemplo).

Do lado direito da janela do aplicativo, temos campos de inserção relacionados à chamada da função **WFSExecute** para configuração da placa e para pedido de conversão. A caixa de verificação garante uma aquisição contínua do número de amostras marcado no campo de edição acima, e o botão do canto inferior esquerdo interrompe esse tipo de aquisição. As chamadas das funções foram feitas segundo as especificações da arquitetura WOSA/XFS.

É importante lembrar que, como foi visto na sessão 4.14, ao chamar as funções **WFSLock** e **WFSExecute**, o aplicativo é responsável por desalocar a estrutura de resultados passada por referência na chamada da função, bem como outras posições de memória alocadas anteriormente (como os resultados da conversão, por exemplo).

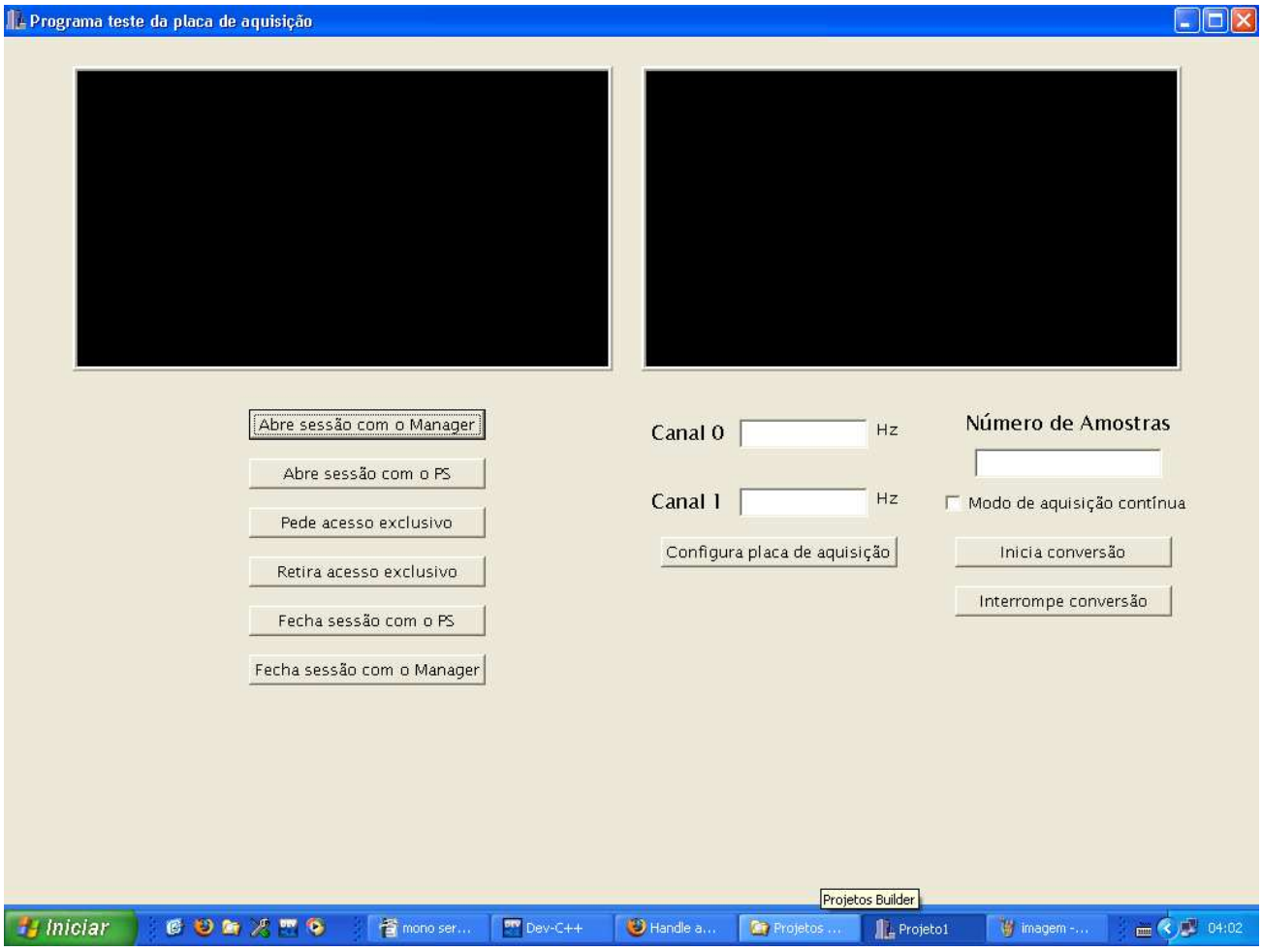


*Figura 15 – Janela do aplicativo*

Para realizar corretamente uma sessão de aquisição de dados, deve-se começar abrindo sessões com o *Manager* e com o provedor de serviços, e pedindo acesso exclusivo, apertando os botões correspondentes, nessa ordem. Em seguida, as taxas de conversão para os dois canais devem ser inseridas nos campos adequados, e a configuração deve ser feita através do botão de configuração. Finalmente, o número total de amostras deve ser inserido no campo indicado, e o pedido de conversão deve ser feito pelo botão correspondente. Caso se deseje uma conversão no modo de aquisição contínua, deve-se marcar a caixa de verificação. Para terminar a sessão, deve-se garantir que o sistema não está adquirindo mais amostras (através do botão "Interrompe conversão"), retirar o acesso exclusivo e fechar as sessões com o provedor de serviços e o *Manager*, pelos botões correspondentes.

## 6.2 RESULTADOS

O sistema funciona de forma adequada, respondendo rapidamente aos pedidos do usuário. Não foi observado atraso entre o momento em que o usuário aperta cada botão e o momento em que surgem as respostas nas caixas de mensagens, bem como o tempo entre o pedido de conversão e a atualização gráfica dos dados, indicando o bom funcionamento dos módulos e da comunicação entre eles.

Os módulos de captura e tráfego de dados obtiveram resultados satisfatórios. Através de um gerador de funções, foi inserida nos canais 0 e 1 do ADC12 uma senóide de 120 Hz com níveis de tensão adequados, e a mesma foi observada nos *displays* gráficos, como mostra a Figura 17. Além disso, foram observadas através do *software* de *debug* do microcontrolador MSP430 (IAR Systems) correspondências entre os pacotes de configuração enviados pelo PC e os pacotes recebidos pelo MSP430, e entre os resultados das conversões A/D e os valores obtidos pelo aplicativo.

O módulo de fornecimento de dados também respondeu de forma adequada. Não foram observados erros de comunicação do aplicativo com o *Manager*, e deste com o provedor de serviços, bem como erros de alocação de memória, que é gerenciada pelo *Manager*.

Em compensação, percebeu-se que o aplicativo só responde quando há movimento do cursor do *mouse* sobre ele. Isso acontece porque o aplicativo não responde a chamadas de

funções, e sim a mensagens enviadas para o mesmo. Dessa forma, é melhor utilizar no desenvolvimento do aplicativo as chamadas assíncronas das funções utilizadas (por exemplo, **WFSAsyncOpen**), para que o aplicativo receba mensagens ao invés de retorno de funções, funcionando de forma mais adequada para usuário.

As figuras 15 e 16 mostram os resultados obtidos após ter sido realizado o procedimento adequado para a conversão de dados. Nele, verifica-se seqüencialmente a abertura de sessão, a obtenção de acesso exclusivo e a configuração da placa. A figura 17 mostra o sistema no seu funcionamento esperado, adquirindo uma senóide de 120 Hz (obtida a partir de um gerador de funções) e plotando as amostras obtidas. A Figura 18, por sua vez, mostra algumas mensagens de erro retornadas pelo sistema XFS quando o procedimento não é obedecido.

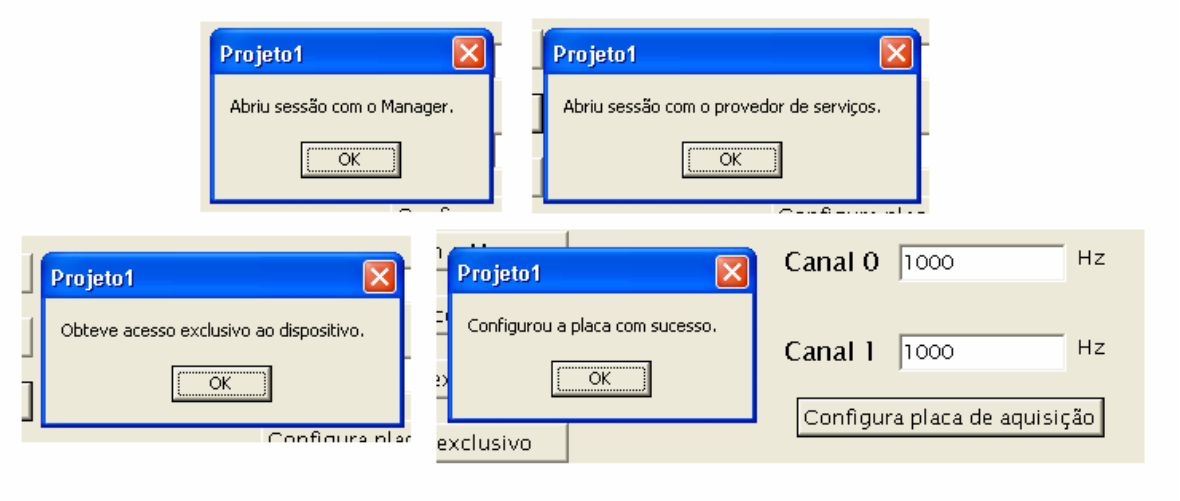


*Figura 16 – Abertura de sessão, acesso exclusivo e configuração da placa de aquisição*

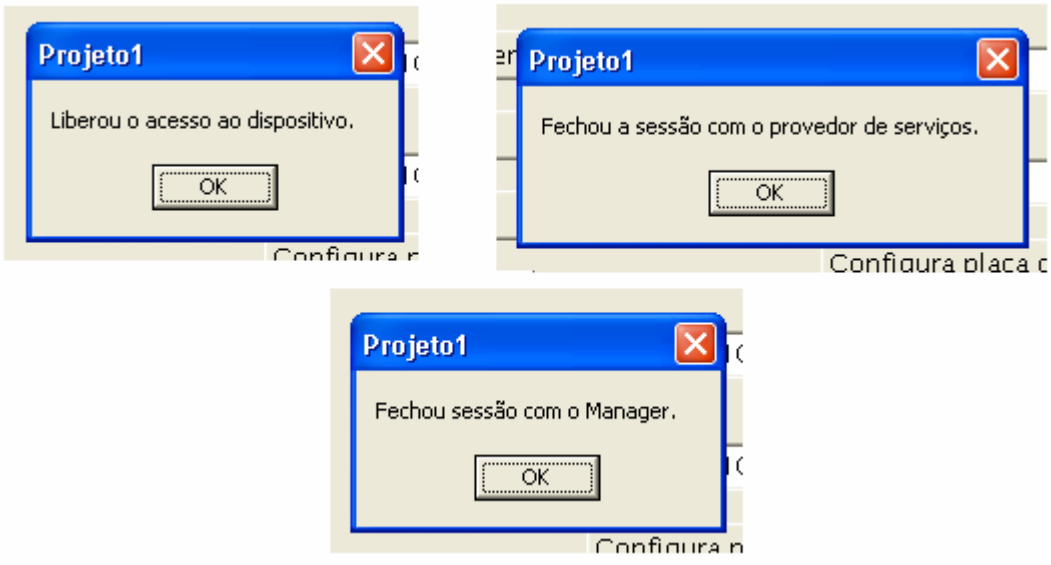


*Figura 17 – Liberando o acesso e fechando uma sessão*

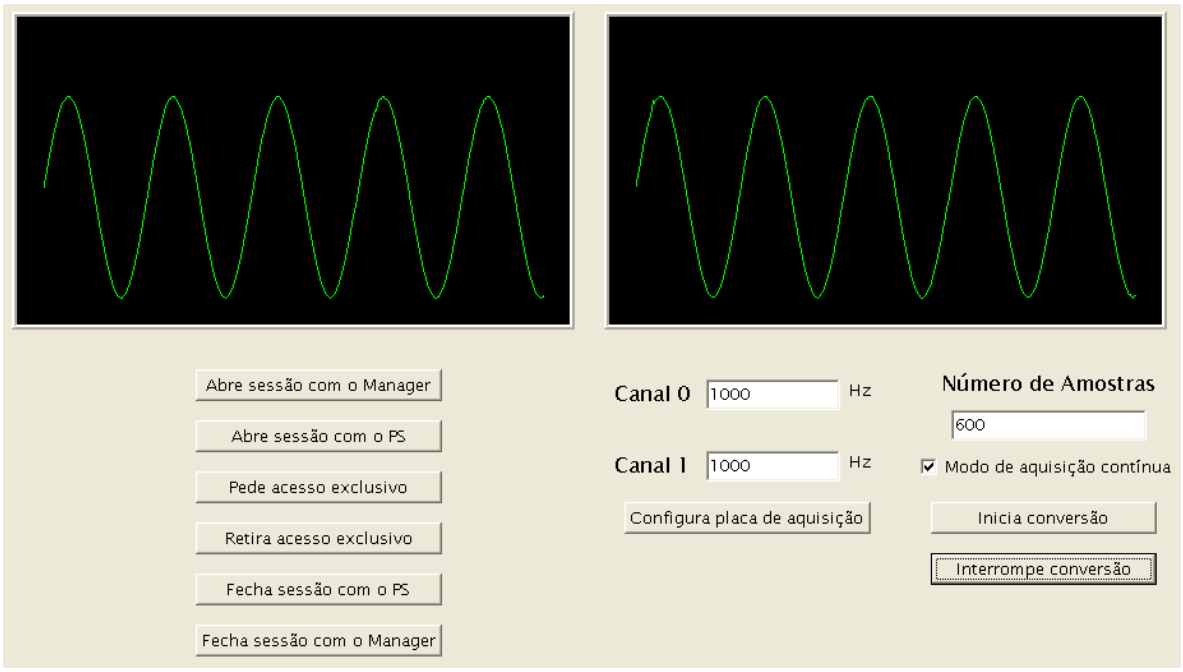


*Figura 18 – Aquisição contínua de uma senóide de 120 Hz*

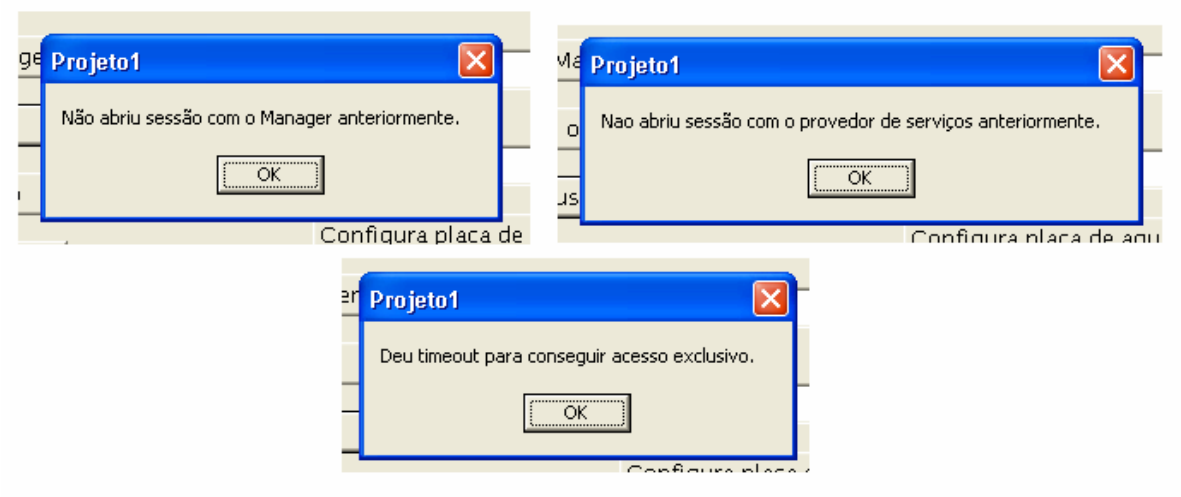


*Figura 19 – Mensagens de erro (não-abertura de sessões e falha em conseguir acesso exclusivo)*

O aplicativo desenvolvido é apenas uma demonstração da implementação adequada do sistema. Um aplicativo com funcionalidade real de aquisição de sinais poderia, por exemplo, remover os botões que abrem e fecham as sessões, bem como aqueles que pedem e liberam o acesso exclusivo, já que servem apenas para ilustrar o procedimento interno de comunicação entre as camadas do sistema. ‘Além disso, podem ser incluídas funções para gravação em arquivo dos dados obtidos, que é um procedimento necessário para o posterior processamento dos sinais obtidos.

# CAPÍTULO 7

## CONCLUSÃO E PERSPECTIVAS

O sistema de medição e aquisição de dados funciona bem, adquirindo e plotando amostras de diferentes sinais analógicos em tempo real. Nesse sistema, é possível conectar-se até oito sinais analógicos simultaneamente ao **módulo de captura de dados**. As amostras desses sinais são então enviadas através do **módulo de tráfego de dados** e, em seguida, são entregues ao aplicativo final pelo **módulo de fornecimento de dados.**

Um sistema desse tipo é de grande utilidade para o laboratório GPDS da UnB, possibilitando a análise de diferentes sinais biológicos, que são os objetos de pesquisa da área de Instrumentação Biomédica. Nesse laboratório, o uso do sistema também será relevante para os projetos da área de Automação Bancária, que deverão apenas adicionar funcionalidades específicas oferecidas pela arquitetura WOSA/XFS para os dispositivos relacionados. Dentro de dois meses, será constituído um ambiente de testes em automação bancária, com cerca de cinco máquinas ATM reais instaladas. Por fim, a utilização do sistema em qualquer das duas áreas, permitirá aos desenvolvedores concentrar seus esforços na aplicabilidade em si de seus estudos, promovendo maior evolução dos projetos em um mesmo período de dedicação.

A concepção do sistema em camadas, permitiu que fossem implementadas apenas as funções de interesse para o aplicativo. Dessa forma, de acordo com a funcionalidade a ser dada ao sistema, outras funções WFP da arquitetura WOSA/XFS deverão ser implementadas. Por exemplo, considerou-se a existência de um único dispositivo a ser acessado pelos aplicativos, o microcontrolador MSP430. Caso existam outros dispositivos, como uma impressora, pode-se então registrar as informações de configuração desse novo dispositivo, expandindo as suas funcionalidades do sistema.

A comunicação entre o PC e o MSP430 realiza-se de forma serial, a pequenas distâncias e através de um meio confiável (via cabo). Além disso, apenas um aplicativo deve conseguir acesso ao dispositivo por vez. Sendo assim, o protocolo de acesso ao meio e o de comunicação são simplificados, não exigindo confirmação de recebimento, retransmissão ou

código complexo de correção de erros. Apenas verifica-se o CRC (*Cyclic Redundancy Check*), uma forma tradicional de verificação de erros. Caso seja de interesse realizar a comunicação por um meio menos confiável, como ocorre na comunicação por radiofreqüência, deve-se então implementar um protocolo mais complexo. Uma outra possibilidade é alterar a forma de comunicação para que seja feita de forma USB (*Universal Serial Bus*), já que a mesma tem se mostrado muito útil em diversas aplicações, como medição e automação. A comunicação USB apresenta taxas de transmissão mais elevadas do que as obtidas por meio da porta serial comum. De acordo com a topologia do barramento, até 127 dispositivos podem ser conectados ao mesmo tempo em um mesmo barramento, ao contrário da porta serial clássica que suporta apenas um dispositivo conectado por vez.

É importante ressaltar que a implementação do sistema em questão muito contribuiu para o desenvolvimento de aspectos do conhecimento relevantes para a formação dos alunos. Entre eles, pode-se citar o aprendizado do funcionamento dos Sistemas Operacionais, a visualização das camadas nele envolvidas, bem como a interface entre elas, no que se refere à compreensão e adequação dos *softwares* desenvolvidos à arquitetura WOSA/XFS. Para viabilizar esse desenvolvimento, foi necessário desenvolver habilidades em programação C e C++, utilizando-se também funções específicas do Windows. Também foi preciso estudar o microcontrolador MSP430 e a forma de comunicação mais adequada para comunicá-lo com o PC. Além disso, não se pode deixar de citar o trabalho inicial de verificação das necessidades do laboratório GPDS, que motivou o estudo sobre aspectos em Instrumentação Biomédica e Automação Bancária, para a compatibilização das necessidades verificadas e o fim a que se propõe este trabalho.

# ANEXO – CÓDIGO DOS PROGRAMAS

- **PROVEDOR DE SERVIÇO**

  - XFSMSP.h

```
#ifndef _XFSMSP_H_
#define _XFSMSP_H_
#include <xfsspi.h>

#define WFS_CMD_MSP_CONFIG 1
#define WFS_CMD_MSP_CONVERT 2

typedef struct _wfs_msp_config
{
  ULONG taxas[8];
  int amostras;
} WFSMSPCONFIG, * LPWFSMSPCONFIG;

#endif /* _SERVICEPROVIDER_H_ */
```

  - SPMSP.h

```
#ifndef _SPMSP_H_
#define _SPMSP_H_
//#include <xfsspi.h>
#include "XFSMSP.h"

//Protocolo com o dispositivo
#define PAC_CONF        13
#define START_BYTE      0x21
#define CONFIG_BYTE     0xAA
#define CONVERT_BYTE    0x33
#define MSP_CLK         8e6

typedef struct _wfs_msp_execute
{
  LPWFSMSPCONFIG Config;
  HWND hWnd;
} WFSMSPEXECUTE, * LPWFSMSPEXECUTE;

extern HSERVICE session_memo[10];
extern HPROVIDER hServiceProvider;
extern HANDLE hLock, hCom;
extern int i, iLock, releaseOK;
extern BOOL config_serial;

//Funcoes - Threads
DWORD WINAPI OpenMessage( LPVOID lpParam );
DWORD WINAPI CloseMessage( LPVOID lpParam );
```

```
DWORD WINAPI LockMessage( LPVOID lpParam );
DWORD WINAPI UnlockMessage( LPVOID lpParam );
DWORD WINAPI ExecuteMessage( LPVOID lpParam );

//Funcoes chamadas pelo XFS Manager
__declspec (dllexport) HRESULT extern WINAPI WFPOpen( HSERVICE hService,
  LPSTR lpszLogicalName, HAPP hApp, LPSTR lpszAppID, DWORD dwTraceLevel,
  DWORD dwTimeOut, HWND hWnd, REQUESTID ReqID, HPROVIDER hProvider,
  DWORD dwSPIVersionsRequired, LPWFSVERSION lpSPIVersion,
  DWORD dwSrvcVersionsRequired, LPWFSVERSION lpSrvcVersion);

__declspec (dllexport) HRESULT extern WINAPI WFPClose ( HSERVICE hService,
  HWND hWnd, REQUESTID ReqID);

__declspec (dllexport) HRESULT extern WINAPI WFPLock ( HSERVICE hService,
  DWORD dwTimeOut, HWND hWnd, REQUESTID ReqID);

__declspec (dllexport) HRESULT extern WINAPI WFPUnlock ( HSERVICE hService,
  HWND hWnd, REQUESTID ReqID);

__declspec (dllexport) HRESULT extern WINAPI WFPUnloadService (  );

__declspec (dllexport) HRESULT extern WINAPI WFPExecute ( HSERVICE
hService,
  DWORD dwCommand, LPVOID lpCmdData, DWORD dwTimeOut, HWND hWnd, REQUESTID
ReqID);

#endif /* _SERVICEPROVIDER_H_ */
```

- WFPOpen.c

```
#include "SPMSP.h"

DWORD WINAPI OpenMessage( LPVOID lpParam )
{
    LPWFSRESULT lpWFSOpenTResult;
    HWND hWindow;

    lpWFSOpenTResult = (LPWFSRESULT)lpParam;
    hWindow = (HWND)(lpWFSOpenTResult->lpBuffer);
    SendMessage(hWindow,WFS_OPEN_COMPLETE,0,(LPARAM)lpWFSOpenTResult);
    return 0;
}

/**************************WFSOPEN*************************************
/
__declspec (dllexport) HRESULT WINAPI WFPOpen ( HSERVICE hService, LPSTR
lpszLogicalName,
HAPP hApp, LPSTR lpszAppID, DWORD dwTraceLevel, DWORD dwTimeOut, HWND hWnd,
REQUESTID ReqID, HPROVIDER hProvider, DWORD dwSPIVersionsRequired,
LPWFSVERSION
lpSPIVersion, DWORD dwSrvcVersionsRequired, LPWFSVERSION lpSrvcVersion)
{
    WORD dwHighVersionRequired, dwLowVersionRequired, dwSwap;
    LPWFSRESULT lpWFSOpenResult;
    WINDOWINFO CallWindow;
    HANDLE hOpenThread;
    DWORD dwOpenThreadID;

    CallWindow.cbSize = sizeof(WINDOWINFO);
    if(!GetWindowInfo(hWnd,&CallWindow))
```

```
{
  if(GetLastError()==ERROR_INVALID_HANDLE)
    return(WFS_ERR_INVALID_HWND);
}

lpSPIVersion->wVersion = 0x003;
lpSPIVersion->wLowVersion = 0x101;
lpSPIVersion->wHighVersion = 0x9903;
strcpy(lpSPIVersion->szDescription,"Terceira versão do SPI");
strcpy(lpSPIVersion->szSystemStatus,"Anything");

lpSrvcVersion->wVersion = 0x001;
lpSrvcVersion->wLowVersion = 0x001;
lpSrvcVersion->wHighVersion = 0x001;
strcpy(lpSrvcVersion->szDescription,"Primeira versão do Serviço");
strcpy(lpSrvcVersion->szSystemStatus,"Anything");

//Version Negotiation do SPI
dwHighVersionRequired = (WORD) (dwSPIVersionsRequired&0xFFFF);
dwSwap = (dwHighVersionRequired>>8);
dwHighVersionRequired = (dwHighVersionRequired<<8) + dwSwap;
dwLowVersionRequired = (WORD) (dwSPIVersionsRequired>>16);
dwSwap = (dwLowVersionRequired>>8);
dwLowVersionRequired = (dwLowVersionRequired<<8) + dwSwap;

if ((0x101 > dwHighVersionRequired)||
    (0x399 < dwLowVersionRequired))
{
    if (0x101 > dwHighVersionRequired)
    {
        return(WFS_ERR_SPI_VER_TOO_LOW);
    }
    if (0x399 < dwLowVersionRequired)
    {
        return(WFS_ERR_SPI_VER_TOO_HIGH);
    }
}

//Version Negotiation do Service Provider
dwHighVersionRequired = (WORD) (dwSrvcVersionsRequired&0xFFFF);
dwSwap = (dwHighVersionRequired>>8);
dwHighVersionRequired = (dwHighVersionRequired<<8) + dwSwap;
dwLowVersionRequired = (WORD) (dwSrvcVersionsRequired>>16);
dwSwap = (dwLowVersionRequired>>8);
dwLowVersionRequired = (dwLowVersionRequired<<8) + dwSwap;

if ((0x100 > dwHighVersionRequired)||
    (0x100 < dwLowVersionRequired))
{
    if (0x100 > dwHighVersionRequired)
    {
        return(WFS_ERR_SRVC_VER_TOO_LOW);
    }
    if (0x100 < dwLowVersionRequired)
    {
        return(WFS_ERR_SRVC_VER_TOO_HIGH);
    }
}
session_memo[i++]=hService;
hServiceProvider=hProvider;
```

```
WFMAllocateBuffer(sizeof(WFSRESULT),WFS_MEM_SHARE,(LPVOID)&lpWFSOpenResult)
;

    lpWFSOpenResult->RequestID = ReqID;
    lpWFSOpenResult->hService = hService;
    lpWFSOpenResult->hResult = WFS_SUCCESS;
    lpWFSOpenResult->lpBuffer = (LPVOID)(hWnd);

    hOpenThread = CreateThread(NULL, 0, OpenMessage, lpWFSOpenResult, 0,
&dwOpenThreadID);
    CloseHandle(hOpenThread);

    return(WFS_SUCCESS);
}
```

- WFPClose.c

```
#include "SPMSP.h"

DWORD WINAPI CloseMessage( LPVOID lpParam )
{
    LPWFSRESULT lpWFSCloseTResult;
    HWND hWindow;

    lpWFSCloseTResult = (LPWFSRESULT)lpParam;
    hWindow = (HWND)(lpWFSCloseTResult->lpBuffer);
    SendMessage(hWindow,WFS_CLOSE_COMPLETE,0,(LPARAM)lpWFSCloseTResult);
    if(WFMReleaseDLL(hServiceProvider)==WFS_SUCCESS)
      releaseOK = 1;
    return 0;
}

/***************************WFSCLOSE
************************************/

__declspec (dllexport) HRESULT WINAPI WFPClose ( HSERVICE hService, HWND
hWnd, REQUESTID ReqID)
{
    HANDLE hCloseThread;
    DWORD dwCloseThreadID;
    LPWFSRESULT lpWFSCloseResult;
    int i1;

    i1 = 0;
    while( (session_memo[i1]!=hService) && (i1<i) )
    {
      i1++;
    }
    if(i1==i)
    {
      return(WFS_ERR_INVALID_HSERVICE);
    }

    session_memo[i1] = 0;
    i--;


WFMAllocateBuffer(sizeof(WFSRESULT),WFS_MEM_SHARE,(LPVOID)&lpWFSCloseResult
);
    lpWFSCloseResult->RequestID = ReqID;
```

```
    lpWFSCloseResult->hResult = WFS_SUCCESS;
    lpWFSCloseResult->lpBuffer = (LPVOID)(hWnd);
    hCloseThread = CreateThread(NULL, 0, CloseMessage, lpWFSCloseResult,
        0, &dwCloseThreadID);
    CloseHandle(hCloseThread);
    return(WFS_SUCCESS);
}
```

- WFPLock.c

```
#include "SPMSP.h"

DWORD WINAPI LockMessage( LPVOID lpParam )
{
  DWORD dwWaitResult;
  LPWFSRESULT lpWFSLockTResult;
  HWND hWindow;
  DCB dcb;
//  char *pcCommPort = "COM1";
  BOOL fSuccess;

  lpWFSLockTResult = (LPWFSRESULT)lpParam;
  hWindow = (HWND)(lpWFSLockTResult->lpBuffer);
  dwWaitResult = WaitForSingleObject(hLock,(lpWFSLockTResult-
>u.dwCommandCode));
  switch (dwWaitResult)
  {
        // A thread conseguiu acesso ao semaforo
        case WAIT_OBJECT_0:
          iLock = 1;
          if(!config_serial)
          {
            hCom = CreateFile("COM1",
              GENERIC_READ | GENERIC_WRITE,
              0,    // must be opened with exclusive-access
              NULL, // no security attributes
              OPEN_EXISTING, // must use OPEN_EXISTING
              0,    // not overlapped I/O
              NULL  // hTemplate must be NULL for comm devices
            );
          fSuccess = GetCommState(hCom, &dcb);
          dcb.BaudRate = CBR_115200;     // set the baud rate
          dcb.ByteSize = 8;             // data size, xmit, and rcv
          dcb.Parity = NOPARITY;        // no parity bit
          dcb.StopBits = ONESTOPBIT;    // one stop bit
          fSuccess = SetCommState(hCom, &dcb);
          config_serial = 1;
          }
        break;
        // A thread nao conseguiu acesso ao semaforo devido a time-out
        case WAIT_TIMEOUT:
          lpWFSLockTResult->hResult = WFS_ERR_TIMEOUT;
        break;
  }
  SendMessage(hWindow,WFS_LOCK_COMPLETE,0,(LPARAM)lpWFSLockTResult);
  return 0;
}

/*****************************WFPLOCK************************************
********/
```

```
__declspec (dllexport) HRESULT WINAPI WFPLock ( HSERVICE hService, DWORD
dwTimeOut,
        HWND hWnd, REQUESTID ReqID)
{
  WINDOWINFO CallWindow;
  LPWFSRESULT lpWFSLockResult;
  HANDLE hLockThread;
  DWORD dwLockThreadID;
  int i1;

  CallWindow.cbSize = sizeof(WINDOWINFO);
  if(!GetWindowInfo(hWnd,&CallWindow))
  {
    if(GetLastError()==ERROR_INVALID_HANDLE)
      return(WFS_ERR_INVALID_HWND);
  }

  i1 = 0;
  while( (session_memo[i1]!=hService) && (i1<i) )
  {
    i1++;
  }

  if(i1==i)
  {
    return(WFS_ERR_INVALID_HSERVICE);
  }


WFMAllocateBuffer(sizeof(WFSRESULT),WFS_MEM_SHARE,(LPVOID)&lpWFSLockResult)
;
  lpWFSLockResult->RequestID = ReqID;
  lpWFSLockResult->hResult = WFS_SUCCESS;
  lpWFSLockResult->hService = hService;
  lpWFSLockResult->u.dwCommandCode = dwTimeOut;
  lpWFSLockResult->lpBuffer = (LPVOID)(hWnd);
  hLockThread = CreateThread(NULL, 0, LockMessage, lpWFSLockResult, 0,
&dwLockThreadID);
  CloseHandle(hLockThread);
  return(WFS_SUCCESS);

}
```

- WFPUnlock.c

```
#include "SPMSP.h"

DWORD WINAPI UnlockMessage( LPVOID lpParam )
{
  DWORD dwWaitResult;
  LPWFSRESULT lpWFSUnlockTResult;
  HWND hWindow;

  lpWFSUnlockTResult = (LPWFSRESULT)lpParam;
  hWindow = (HWND)(lpWFSUnlockTResult->lpBuffer);
  if(iLock == 0)
  {
    lpWFSUnlockTResult->hResult = WFS_ERR_NOT_LOCKED;
  }
  else
  {
```

```
    ReleaseSemaphore(hLock,1,NULL);
    config_serial = 0;
    CloseHandle(hCom);
  }
  SendMessage(hWindow,WFS_UNLOCK_COMPLETE,0,(LPARAM)lpWFSUnlockTResult);
  return 0;
}

/*********************************WFPUNLOCK*****************************
****/

__declspec (dllexport) HRESULT extern WINAPI WFPUnlock ( HSERVICE hService,
HWND hWnd,
        REQUESTID ReqID)
{
  WINDOWINFO CallWindow;
  LPWFSRESULT lpWFSUnlockResult;
  HANDLE hUnlockThread;
  DWORD dwUnlockThreadID;
  int i1;

  CallWindow.cbSize = sizeof(WINDOWINFO);
  if(!GetWindowInfo(hWnd,&CallWindow))
  {
    if(GetLastError()==ERROR_INVALID_HANDLE)
      return(WFS_ERR_INVALID_HWND);
  }

  i1 = 0;
  while( (session_memo[i1]!=hService) && (i1<i) )
  {
    i1++;
  }

  if(i1==i)
  {
    return(WFS_ERR_INVALID_HSERVICE);
  }


WFMAllocateBuffer(sizeof(WFSRESULT),WFS_MEM_SHARE,(LPVOID)&lpWFSUnlockResul
t);
  lpWFSUnlockResult->RequestID = ReqID;
  lpWFSUnlockResult->hResult = WFS_SUCCESS;
  lpWFSUnlockResult->lpBuffer = (LPVOID)(hWnd);
  lpWFSUnlockResult->hService = hService;
  hUnlockThread = CreateThread(NULL, 0, UnlockMessage, lpWFSUnlockResult,
      0, &dwUnlockThreadID);
  CloseHandle(hUnlockThread);
  return(WFS_SUCCESS);
}
```

- WFPExecute.c

```
#include "SPMSP.h"

DWORD WINAPI ExecuteMessage( LPVOID lpParam )
{
  LPWFSMSPEXECUTE lpWFSMSPExecuteResult;
  HWND hWindow;
  LPWFSRESULT lpWFSExecuteTResult;
```

```
LPWFSMSPCONFIG lpConfig;
CHAR *data_convert, data_config[PAC_CONF] = {0};
ULONG i1;
int i2;
BOOL fSuccess;
HRESULT resultado;
DWORD dwBytes;

lpWFSExecuteTResult = (LPWFSRESULT)lpParam;
lpWFSMSPExecuteResult = (LPWFSMSPEXECUTE)lpWFSExecuteTResult->lpBuffer;
hWindow = lpWFSMSPExecuteResult->hWnd;
lpConfig = lpWFSMSPExecuteResult->Config;

if(!iLock)
{
  lpWFSExecuteTResult->hResult = WFS_ERR_LOCKED;
}
else
{
  switch(lpWFSExecuteTResult->u.dwCommandCode)
  {
    case WFS_CMD_MSP_CONFIG:
      data_config[0] = START_BYTE;
      data_config[1] = CONFIG_BYTE;
      i2 = 0;

      for(i1 = 1; i1<8; i1++)
      {
        if((lpConfig->taxas[i2])<(lpConfig->taxas[i1]))
          i2 = i1;
      }

    //O 3o e o 4o byte do vetor de configuracao indicam
    //as contagens do Timer A, para gerar a taxa mais alta de conversao
      i1 = MSP_CLK/(lpConfig->taxas[i2]);
      data_config[2] = (i1&0xFF);        //byte menos significativo
      data_config[3] = (i1>>8);   //byte mais significativo
      i1 = lpConfig->taxas[i2];

      for(i2 = 0; i2<8; i2++)
      {
        if(lpConfig->taxas[i2])
          data_config[i2+4] = i1/(lpConfig->taxas[i2]);
      }

      data_config[12] = 0;               //CRC
      for(i2 = 0; i2<8; i2++)
        data_config[12] += data_config[i2];

      //Envia pacote de configuracao
      fSuccess = WriteFile(hCom,(LPCVOID)data_config,13,&dwBytes,NULL);
      lpWFSExecuteTResult->hResult = WFS_SUCCESS;
    break;

    case WFS_CMD_MSP_CONVERT:
      data_config[0] = START_BYTE;
      data_config[1] = CONVERT_BYTE;
      data_config[2] = lpConfig->amostras;
      data_config[3] = (lpConfig->amostras)>>8;

      data_config[12] = 0;               //CRC
      for(i2 = 0; i2<8; i2++)
```

```
          data_config[12] += data_config[i2];

        resultado = WFMAllocateBuffer((lpConfig->amostras)+1,
          WFS_MEM_SHARE,(LPVOID)&data_convert);

        if(resultado != WFS_SUCCESS)
        {
          lpWFSExecuteTResult->hResult = WFS_ERR_INTERNAL_ERROR;
        }
        else
        {
          fSuccess = WriteFile(hCom,(LPCVOID)data_config,13,&dwBytes,NULL);
          fSuccess = ReadFile(hCom,(LPVOID)data_convert,
            (DWORD)(lpConfig->amostras),
            &dwBytes,NULL);
          lpWFSExecuteTResult->lpBuffer = (LPVOID)data_convert;
          lpWFSExecuteTResult->hResult = WFS_SUCCESS;
        }
      break;
      default:
        lpWFSExecuteTResult->hResult = WFS_ERR_INVALID_COMMAND;
    }
  }

SendMessage(hWindow,WFS_EXECUTE_COMPLETE,0,(LPARAM)(lpWFSExecuteTResult));
  return 0;
}

/**********************************WFPEXECUTE*****************************
***/

__declspec (dllexport) HRESULT extern WINAPI WFPExecute
      ( HSERVICE hService, DWORD dwCommand, LPVOID lpCmdData,
      DWORD dwTimeOut, HWND hWnd, REQUESTID ReqID)
{
  int i1;
  static int load = 0;
  WINDOWINFO CallWindow;
  HANDLE hExecuteThread;
  DWORD dwExecuteThreadID;
  LPWFSRESULT lpWFSExecuteResult;
  WFSMSPEXECUTE WFSMSPExecute;

  CallWindow.cbSize = sizeof(WINDOWINFO);
  if(!GetWindowInfo(hWnd,&CallWindow))
  {
    if(GetLastError()==ERROR_INVALID_HANDLE)
      return(WFS_ERR_INVALID_HWND);
  }

  i1 = 0;
  while( (session_memo[i1]!=hService) && (i1<i) )
  {
    i1++;
  }

  if(i1==i)
  {
    return(WFS_ERR_INVALID_HSERVICE);
  }
```

```
  WFMAllocateBuffer(sizeof(LPWFSRESULT),WFS_MEM_SHARE,(LPVOID
*)&lpWFSExecuteResult);
  WFSMSPExecute.Config = (LPWFSMSPCONFIG) lpCmdData;
  WFSMSPExecute.hWnd = hWnd;
  lpWFSExecuteResult->u.dwCommandCode = dwCommand;
  lpWFSExecuteResult->RequestID = ReqID;
  lpWFSExecuteResult->hService = hService;
  lpWFSExecuteResult->lpBuffer = (LPVOID)&WFSMSPExecute;

  hExecuteThread = CreateThread(NULL, 0, ExecuteMessage,
    (LPVOID)lpWFSExecuteResult, 0, &dwExecuteThreadID);
  CloseHandle(hExecuteThread);
  return(WFS_SUCCESS);
}
```

- WFPUnloadService.c

```
#include "SPMSP.h"

__declspec (dllexport) HRESULT extern WINAPI WFPUnloadService()
{
  if(releaseOK)
  {
    releaseOK = 0;
    return WFS_SUCCESS;
  }
  return WFS_ERR_NOT_OK_TO_UNLOAD;
}
```

- WFPDLLMain.c

```
#include "SPMSP.h"

HSERVICE session_memo[10];
HPROVIDER hServiceProvider;
HANDLE hLock, hCom;
int i = 0, iLock = 0, releaseOK = 0;
BOOL config_serial = 0;

/**********************************DLLMAIN*********************************
**/

BOOL APIENTRY DllMain (HINSTANCE hInst     /* Library instance handle. */ ,
                       DWORD reason        /* Reason this function is being
called. */ ,
                       LPVOID reserved     /* Not used. */ )
{
    switch (reason)
    {
      case DLL_PROCESS_ATTACH:
        hLock = CreateSemaphore(NULL,1,1,"MSP");
      break;

      case DLL_PROCESS_DETACH:
      break;

      case DLL_THREAD_ATTACH:
      break;

      case DLL_THREAD_DETACH:
```

```
      break;
    }

    return TRUE;
}
```

- **MSP430**

  - Definicoes.h

```
# include "msp430x14x.h"

#define AD_DIR          P6DIR
#define AD_SEL          P6SEL

#define PAC_DATA        1000
#define PAC_CONF        13
#define START_BYTE      0x21
#define CONFIG_BYTE     0xAA
#define CONVERT_BYTE    0x33


void Configurar_Clock(void);
void Configurar_AD_1(void);
void Configurar_UART(void);
void Configurar_AD_2(char *fim_contagem, char *conf, int *taxa, int
*int_AD);
```

  - Funcoes.c

```
#include "Definicoes.h"

#define Configurar_TimerA() (TACTL = TASSEL_2 + MC_0 + ID_0 + TAIE)

void Configurar_Clock(void)
{
  int i;
  BCSCTL1 &= ~XT2OFF; // XT2on
  do
  {
    IFG1 &= ~OFIFG; // Clear OSCFault flag
    for (i = 0xFF; i > 0; i--); // Time for flag to set
  }
  while ((IFG1 & OFIFG)); // OSCFault flag still set?
  BCSCTL2 |= SELM_2 + SELS + DIVS_0; // MCLK = 8MHz, SMCLK = 8MHz.
}

void Configurar_AD_1(void)
{
  ADC12CTL0 &= ~(ADC12SC + ENC);
  AD_DIR &= 0x00;
  ADC12CTL0 |= SHT0_0 + MSC + REF2_5V + REFON + ADC12ON;
  ADC12CTL1 = CSTARTADD_0 + SHS_0 + SHP + ADC12DIV_0 + ADC12SSEL_2 +
CONSEQ_1;
}

void Configurar_UART(void)
{
```

```
  P3DIR = 0xFF;
  P3SEL = 0x30;
  UCTL0 |= SWRST + CHAR;
  UTCTL0 |= SSEL1; // Escolhe SMCLK para atuar na UART
  UBR00 = 69; // Divisor é 69 para Baud Rate de 115,2kbps
  UBR10 = 0x00;
  UMCTL0 = 0x00; // Modulação, parte fracionária da divisão.
  ME1 |= UTXE0 + URXE0; //Habilita módulos de transmissão e recepção
  U0CTL &= ~SWRST;
  IE1 |= URXIE0;
}

void Configurar_AD_2(char *fim_contagem, char *conf, int *taxa, int
*int_AD)
{
  char f, confADaux, *pConfAD;//confAD
  int confAD;  //Para configurar registradores de 16 bits (ADC12CTL0)

  ADC12CTL0 &= ~(ADC12SC + ENC); //Nao se configura o AD com ENC = 1
//Zerar o buffer de contagens dos canais,
//para permitir novas taxas de amostragens
  for(f=0;f<8;f++)
    fim_contagem[f]=0;

//O 3o e o 4o byte do vetor conf indicam as contagens do Timer A,
//para gerar a taxa mais alta de conversao
  *taxa = conf[2]+(conf[3]<<8);
  TACCR0 = *taxa;

  confAD = 0;
  f = 0;
  while((conf[f + 4]!=0)&&(f!=8))
    f++;

  confAD = f; //Guardar em confAD o numero de canais usado
  confADaux = 1;
  for(f=0;f<confAD;f++)
    confADaux *= 2;
  confADaux -= 1;   //confADaux = 2^confAD - 1
  AD_SEL = confADaux;

  confADaux++;
  confADaux /= 2;   //confADaux = 2^(confAD - 1)
  *int_AD = confADaux;

//Atribui o canal 0 para a memoria 0, canal 1 para a memoria 1 etc.,
//até o canal final
  pConfAD = (char *)&ADC12MCTL0;
  for(f=0;f<confAD;f++)
    *(pConfAD++) = f + SREF_1;
  pConfAD--;
  *pConfAD |= EOS;

//O vetor fim_contagem guarda o fim de contagem de cada um
//dos canais usados
  for(f=0;f<8;f++)
  {
    fim_contagem[f] = conf[f+4];
  }
}
```

- Programa Final.c

```
#include "Funcoes.c"

char conf[PAC_CONF] = {0}, pacote[PAC_DATA] = {0},
//START_BYTE, CONFIG_BYTE, 0xFF,0,1,2,0,0,0,0,0,0,0xCD},
contagem[8], fim_contagem[8],
crc = 0, *pConf, *pPacote, *pEnvio;
int amostras = 0, taxa = 0, AD_ie = 0;//, ok = 0;//flag

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

void main(void)
{
  char f;
  WDTCTL = WDTPW + WDTHOLD; // Parar WDT
  P1DIR = 1;
  Configurar_Clock();
  Configurar_AD_1();
  Configurar_UART();
  Configurar_TimerA();

  pConf = conf;
  pPacote = pacote;
  pEnvio = pacote;
  P1OUT = 0;
  _EINT();
  while(1)
  {
    if(pConf == (conf + 1))
    {
      if(conf[0] != START_BYTE)
        pConf = conf;
    }
    if(pConf == (conf + PAC_CONF))
    {
      pConf = conf;
      for(f=0;f<11;f++)
        crc += conf[f];
      if (crc == conf[12])
      {
        if(conf[1] == CONFIG_BYTE)  //Termina de configurar o AD
        {
          Configurar_AD_2(fim_contagem, conf, &taxa, &AD_ie);
          P1OUT = 1;
        }
        if(conf[1] == CONVERT_BYTE)  //Pede conversao
        {
          amostras = conf[2] + (conf[3]<<8);
          for(f = 0; f < 8; f++)
            contagem[f] = 0;
          IE1 &= ~URXIE0;
          TACTL |= MC_1;
          ADC12IE = AD_ie;
          P1OUT = 0;
        }
      }
      crc = 0;
    }
    if((pPacote != pEnvio)&&(!amostras))
    {
      do
```

```
      {
        //Espera até buffer de transmissão estar vazio
        while ((TXEPT&U0TCTL)==0);
        //Envia
        TXBUF0=*(pEnvio++);
        while ((TXEPT&U0TCTL)==0);
        if(pEnvio == (pacote + PAC_DATA))
          pEnvio = pacote;
      }while(pPacote != pEnvio);
      IE1 |= URXIE0;
    }
  }
}

#pragma vector = USART0RX_VECTOR
__interrupt void USART0ISR (void)
{
  if(pConf == (conf + PAC_CONF))
    pConf = conf;
  *(pConf++) = RXBUF0;
}

#pragma vector=ADC_VECTOR
__interrupt void ADC12ISR (void)
{
  int i = 0, *pADC12MEM;

  pADC12MEM = (int*)&ADC12MEM0;
  while(fim_contagem[i]!=0)
  {
    if(pPacote == (pacote + PAC_DATA))
      pPacote = pacote;
    contagem[i] += 1;
    if(!amostras)
    {
      Configurar_TimerA();
      ADC12IE=0;
      break;
    }
//Decide se o canal deve ser armazenado. Caso afirmativo,
//guarda o canal e zera o contador.
    if(contagem[i]==fim_contagem[i])
    {
      *(pPacote++) = (*(pADC12MEM)>>8)|(i<<4);
      *(pPacote++) = *(pADC12MEM++);
      contagem[i] = 0;
      amostras -= 2;
    }
    i++;
  }
  ADC12CTL0 &= ~ENC;
  ADC12IFG = 0; //Tira pedido de interrupção.
}

#pragma vector=TIMERA1_VECTOR
__interrupt void Timer_A(void)
{

  if( TAIV == 0xA )
  {
    TACCR0 = taxa;
    ADC12IFG = 0;
```

```
    ADC12CTL0 |= ADC12SC + ENC;
      // Inicia a conversão da seqüência de canais.
  }
}
```

- **APLICATIVO C++BUILDER**

  - Teste.h

```
//---------------------------------------------------------------------------
--

#ifndef TestH
#define TestH
//---------------------------------------------------------------------------
--
#include <Classes.hpp>
#include <Controls.hpp>
#include <StdCtrls.hpp>
#include <Forms.hpp>
#include <Dialogs.hpp>
#include <GraphicPanel.h>
#include <ExtCtrls.hpp>
//---------------------------------------------------------------------------
--
class TForm1 : public TForm
{
__published:      // IDE-managed Components
        TButton *Startup;
        TButton *CleanUp;
        TGraphicPanel *GraphicPanel1;
        TGraphicPanel *GraphicPanel2;
        TButton *Lock;
        TButton *Unlock;
        TButton *Execute_Cfg;
        TButton *Execute_Conv;
        TButton *Open;
        TButton *Close;
        TEdit *txCanal0;
        TEdit *txCanal1;
        TLabel *Label1;
        TLabel *Label2;
  TEdit *amostras;
        TLabel *Label3;
        TLabel *Label4;
        TLabel *Label5;
  TCheckBox *AqCont;
  TButton *ExecuteConvStop;
        void __fastcall StartupClick(TObject *Sender);
        void __fastcall OpenClick(TObject *Sender);
  void __fastcall LockClick(TObject *Sender);
  void __fastcall UnlockClick(TObject *Sender);
  void __fastcall CloseClick(TObject *Sender);
  void __fastcall CleanUpClick(TObject *Sender);
  void __fastcall Execute_ConvClick(TObject *Sender);
        void __fastcall Execute_CfgClick(TObject *Sender);
  void __fastcall ExecuteConvStopClick(TObject *Sender);

private:    // User declarations
```

```
public:           // User declarations
        __fastcall TForm1(TComponent* Owner);
};
//---------------------------------------------------------------------------
extern PACKAGE TForm1 *Form1;
//---------------------------------------------------------------------------
#endif
```

- Projeto1.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop
USEFORM("Test.cpp", Form1);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        try
        {
                 Application->Initialize();
                 Application->CreateForm(__classid(TForm1), &Form1);
     Application->Run();
        }
        catch (Exception &exception)
        {
                 Application->ShowException(&exception);
        }
        return 0;
}
```

- Test.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <stdio.h>

#pragma hdrstop

#include "XFSMSP.h"
#include "XFSADMIN.h"
#include "Test.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
//#pragma link "CPort"
#pragma link "GraphicPanel"
#pragma resource "*.dfm"
TForm1 *Form1;
HSERVICE hService;
HRESULT resultado;
WFSMSPCONFIG config;
BOOL convert = 0;
//---------------------------------------------------------------------------
```

```
__fastcall TForm1::TForm1(TComponent* Owner)
        : TForm(Owner)
{
  GraphicPanel1->DefineHandle();
  GraphicPanel1->AtualizaPar();
  GraphicPanel1->LigaBarra();
  GraphicPanel1->MudaCorFundo(clBlack);

  GraphicPanel2->DefineHandle();
  GraphicPanel2->AtualizaPar();
  GraphicPanel2->LigaBarra();
  GraphicPanel2->MudaCorFundo(clBlack);
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::StartupClick(TObject *Sender)
{
  WFSVERSION WFSVersion;

  resultado = WFSStartUp(0x00010002,&WFSVersion);

  switch(resultado)
  {
    case WFS_SUCCESS:
      ShowMessage("Abriu sessão com o Manager.");
    break;
    case WFS_ERR_API_VER_TOO_HIGH:
      ShowMessage("Versão muito alta da API.");
    break;
    case WFS_ERR_API_VER_TOO_LOW:
      ShowMessage("Versão muito baixa da API.");
    break;
    case WFS_ERR_ALREADY_STARTED:
      ShowMessage("Já foi aberta a sessão com o Manager.");
    break;
    case WFS_ERR_INTERNAL_ERROR:
      ShowMessage("Aconteceu um erro interno no subsistema XFS.");
    break;
    default:
      ShowMessage("Problema em WFSStartup.");
  }

}

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

void __fastcall TForm1::OpenClick(TObject *Sender)
{
  WFSVERSION WFSServiceVersion,WFSSPIVersion;

  resultado = WFSOpen("microcontroller",WFS_DEFAULT_HAPP,NULL,NULL,
    WFS_INDEFINITE_WAIT,0x010002,&WFSServiceVersion,
    &WFSSPIVersion,&hService);

  switch(resultado)
  {
    case WFS_SUCCESS:
      ShowMessage("Abriu sessão com o provedor de serviços.");
    break;
    case WFS_ERR_NOT_STARTED:
```

```
      ShowMessage("Não abriu sessão com o Manager anteriormente.");
    break;
    case WFS_ERR_INTERNAL_ERROR:
      ShowMessage("Erro interno no sistema XFS.");
    break;
  }
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::LockClick(TObject *Sender)
{
  LPWFSRESULT lpLock, lpExecute;

  resultado = WFSLock(hService,5000L,&lpLock);

  switch(resultado)
  {
    case WFS_SUCCESS:
      ShowMessage("Obteve acesso exclusivo ao dispositivo.");
    break;
    case WFS_ERR_TIMEOUT:
      ShowMessage("Deu timeout para conseguir acesso exclusivo.");
    break;
    case WFS_ERR_INVALID_HSERVICE:
      ShowMessage("Nao abriu sessão com o provedor de serviços
anteriormente.");
    break;
    case WFS_ERR_NOT_STARTED:
      ShowMessage("Não foi aberta a sessão com o Manager.");
    break;
  }
  resultado = WFSFreeResult(lpLock);
  if(resultado!=WFS_SUCCESS)
    ShowMessage("Erro desalocando memória em WFSLock().");
}
//---------------------------------------------------------------------------
--


void __fastcall TForm1::UnlockClick(TObject *Sender)
{
  //resultado = WFMFreeBuffer(config);
  resultado = WFSUnlock(hService);
  switch(resultado)
  {
    case WFS_SUCCESS:
      ShowMessage("Liberou o acesso ao dispositivo.");
    break;
    case WFS_ERR_NOT_LOCKED:
      ShowMessage("Nao garantiu o acesso anteriormente.");
    break;
    case WFS_ERR_INVALID_HSERVICE:
      ShowMessage("Nao abriu sessão com o provedor de serviços
anteriormente.");
    break;
    case WFS_ERR_NOT_STARTED:
      ShowMessage("Não foi aberta a sessão com o Manager.");
    break;
  }
}
```

```
//---------------------------------------------------------------------------
--

void __fastcall TForm1::CloseClick(TObject *Sender)
{
  resultado = WFSClose(hService);
  switch(resultado)
  {
    case WFS_SUCCESS:
          ShowMessage("Fechou a sessão com o provedor de serviços.");
    break;
    case WFS_ERR_NOT_STARTED:
      ShowMessage("Não foi aberta a sessão com o Manager.");
    break;
  }
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::CleanUpClick(TObject *Sender)
{
  resultado = WFSCleanUp();
  switch(resultado)
  {
    case WFS_SUCCESS:
      ShowMessage("Fechou sessão com o Manager.");
    break;
    case WFS_ERR_NOT_STARTED:
      ShowMessage("Não havia sessão para fechar.");
    break;
  }
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::Execute_ConvClick(TObject *Sender)
{
  unsigned int i;
  static bool AmostraFlag=false;
  static unsigned int canal;
  unsigned int canal_temp;
  static unsigned int amostra_temp;
  unsigned int amostra;
  unsigned char *buffer;
  LPWFSRESULT lpExecute;

  config.amostras = StrToInt(amostras->Text);
  convert = TRUE;
  do
  {
    resultado = WFSExecute(hService, WFS_CMD_MSP_CONVERT,
      (LPVOID)&config,WFS_INDEFINITE_WAIT,&lpExecute);
    switch(resultado)
    {
      case WFS_ERR_LOCKED:
        ShowMessage("Não garantiu o acesso exclusivo anteriormente.");
      break;
      case WFS_ERR_NOT_STARTED:
        ShowMessage("Não abriu sessão com o provedor de serviços
anteriormente.");
      break;
      case WFS_SUCCESS:
```

```
        buffer = (unsigned char *)(lpExecute->lpBuffer);

        for(i=0;i<(unsigned int)StrToInt(amostras->Text);i++)
        {
          canal_temp = (((unsigned int)(buffer[i]))>>4)&0x0F;
          if((canal_temp == 0||canal_temp == 1)&&(!AmostraFlag))
          {
            amostra_temp =  (((unsigned int)(buffer[i]))<<8)&0x0F00;
            AmostraFlag=true;
            canal=canal_temp;
          }
          else if(AmostraFlag)
          {
            amostra = amostra_temp + (((unsigned int)(buffer[i]))&0x00FF);
            AmostraFlag=false;
            switch(canal)
            {
              case 0:
                GraphicPanel1->InserePonto((amostra*GraphicPanel1-
>Height)/4095-GraphicPanel1->Height/2);
                //GraphicPanel1->MudaCorLinha(clBlue);
                GraphicPanel1->Plota();
//                GraphicPanel1->LigaBarra();
              break;

              case 1:
                GraphicPanel2->InserePonto((amostra*GraphicPanel2-
>Height)/4095-GraphicPanel2->Height/2);
                GraphicPanel2->Plota();
              break;
            }
          }
        }
        resultado = WFMFreeBuffer(buffer);
        if(resultado!=WFS_SUCCESS)
          ShowMessage("Erro desalocando memória dos resultados.");
        resultado = WFSFreeResult(lpExecute);
        if(resultado!=WFS_SUCCESS)
          ShowMessage("Erro desalocando memória em WFSExecute(convert).");
      break;
    }
  }while(convert&&AqCont->Checked);
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::Execute_CfgClick(TObject *Sender)
{
  LPWFSRESULT lpExecute;
  int i;
  //resultado =
WFMAllocateBuffer(sizeof(LPWFSMSPCONFIG),WFS_MEM_SHARE,(LPVOID
*)(&config));
  config.taxas[0] = (ULONG)StrToInt64(txCanal0->Text);
  config.taxas[1] = (ULONG)StrToInt64(txCanal1->Text);
  for(i=2;i<8;i++)
    config.taxas[i] = 0;

  resultado = WFSExecute(hService, WFS_CMD_MSP_CONFIG,
    (LPVOID)&config,WFS_INDEFINITE_WAIT,&lpExecute);

  switch(resultado)
```

```
  {
    case WFS_SUCCESS:
      ShowMessage("Configurou a placa com sucesso.");
    break;
    case WFS_ERR_LOCKED:
      ShowMessage("Não garantiu o acesso exclusivo anteriormente.");
    break;
    case WFS_ERR_NOT_STARTED:
      ShowMessage("Não abriu sessão com o provedor de serviços
anteriormente.");
    break;
    case WFS_ERR_INTERNAL_ERROR:
      ShowMessage("Erro interno do sistema XFS.");
    break;
  }
  resultado = WFSFreeResult(lpExecute);
  if(resultado!=WFS_SUCCESS)
    ShowMessage("Erro desalocando memória em WFSExecute(config).");
}
//---------------------------------------------------------------------------
--

void __fastcall TForm1::ExecuteConvStopClick(TObject *Sender)
{
  convert = FALSE;
}
//---------------------------------------------------------------------------
--
```