

Capítulo 4

Estudos de casos

Este capítulo tem por objetivo avaliar a aplicação do *framework DAQSystem* em sistemas de coleta de sinais através de uma série de estudos de caso. Para isso, o *framework DAQSystem* foi utilizado no desenvolvimento de um software de visualização de sinais que opera com três equipamentos diferentes. Ao software de visualização, foi dado o nome de *BioScope* e a Figura 4.1 apresenta as etapas de execução do estudo de caso.

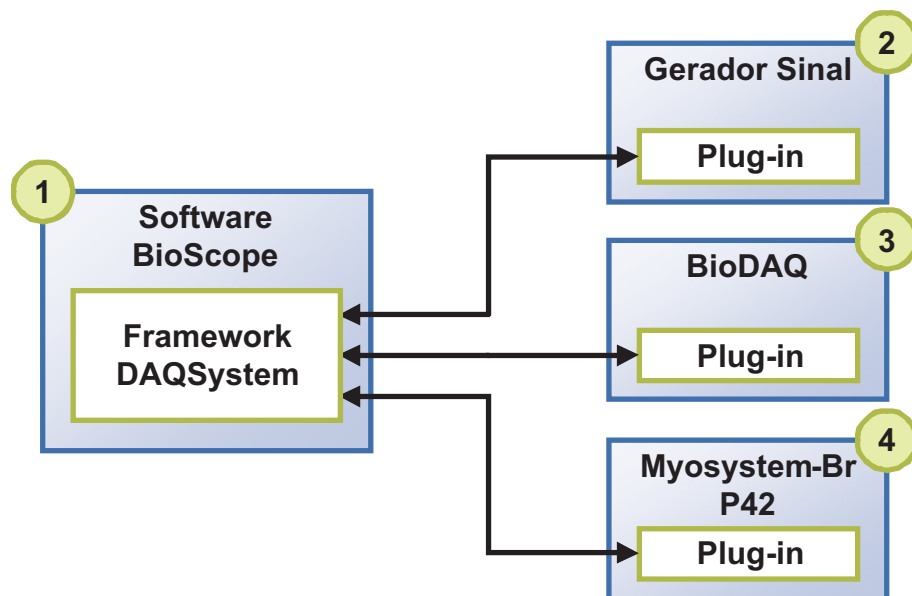


Figura 4.1: Diagrama esquemático de interconexão de um aplicativo (*BioScope*) a três equipamentos.

O primeiro passo executado foi o desenvolvimento do aplicativo *BioScope*. Este aplicativo utiliza o *framework DAQSystem* como unidade de controle do SAD. O aplicativo é capaz acessar um único dispositivo em um determinado momento, entretanto, pode alternar a utilização entre diferentes equipamentos.

Numa segunda etapa, o aplicativo foi conectado ao sistema de aquisição de dados virtual fornecido pelo *plug-in GeradorSinal*, apresentado no capítulo anterior. O *plug-in* gera uma série de sinais conhecidos e bem determinados, permitindo verificar se os gráficos gerados pelo aplicativo *BioScope* estão corretos.

O terceiro passo executado é a conexão do aplicativo *BioScope* com um equipamento real. O equipamento *BioDAQ*, desenvolvido no Laboratório de Engenharia Biomédica e Automática da Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia, foi utilizado nesta etapa. Desenvolveu-se o *plug-in* para o equipamento e foram realizados alguns experimentos.

Num quarto momento, o aplicativo *BioScope* foi conectado ao equipamento *MyoSystem-Br P42* (Datahominis Tecnologia Ltda, Uberlândia - MG), utilizado para captura de sinais eletromiográficos. Novamente, foi desenvolvido um novo *plug-in* para o sistema de aquisição, sendo avaliado o seu funcionamento.

Após a conclusão desses passos, foram realizados dois experimentos para avaliar a facilidade de atualização destes sistemas. No primeiro experimento, o equipamento *BioDAQ* recebeu um novo recurso, e foram verificadas as influências dessa atualização no software *BioScope*.

Em seguida, o software *BioScope* foi modificado para permitir a utilização de mais de um equipamento simultaneamente. Com isso, foi possível avaliar as influências que as modificações no software *BioScope* gerou nos equipamentos e seus respectivos *plug-ins*. Este experimento também foi utilizado para demonstrar a capacidade do *framework DAQSystem* ser utilizado na integração de diversos equipamentos. As próximas seções apresentam mais detalhes da implementação e da execução dos processos descritos.

4.1 O desenvolvimento do aplicativo *BioScope*

O software *BioScope* foi criado para permitir a visualização gráfica de um ou mais sinais captados por um equipamento qualquer. Para isso, utiliza o *framework DAQSystem* para acesso aos equipamentos.

A Figura 4.2 mostra um diagrama da interface com o usuário fornecida pelo sistema. Esta interface pode ser dividida em duas regiões: A região à esquerda contém controles que permitem a configuração do equipamento e o controle do processo de aquisição; a região à direita apresenta uma série de gráficos, um para cada canal ativado, que representam os sinais coletados pelo equipamento.

A área de configuração é dividida em três painéis. O primeiro painel deve fornecer acesso às configurações do equipamento. Através deste painel, o usuário poderá escolher qual o equipamento será utilizado e configurar o processo de aquisição. O segundo painel é utilizado para fornecer acesso às configurações de cada canal do equipamento selecionado no primeiro painel. O último painel é utilizado para controlar o processo de aquisição (Iniciar/Parar).

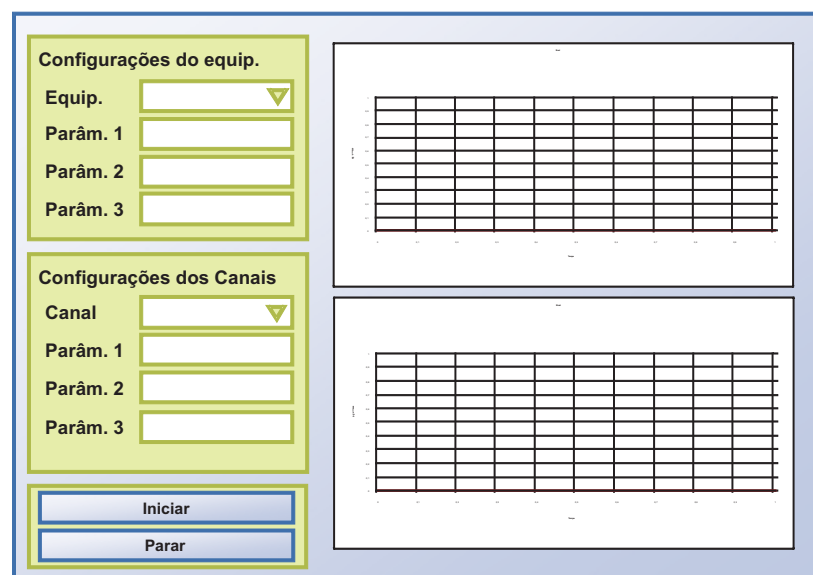


Figura 4.2: Exemplo de interface para o aplicativo *BioScope*.

A Figura 4.3 apresenta um diagrama de classes simplificado do aplicativo *BioScope*.

Nele, tem-se a janela principal do aplicativo modelada na classe *FormPrincipal*. Esta classe é composta um sistema de aquisição (Classe *DAQ* do *Framework*) e uma série de gráficos que irão apresentar a forma de onda dos diversos canais. É através do objeto *DAQ* do *framework DAQSystem* que o aplicativo acessa o sistema de aquisição de dados. Como *DAQ* é uma interface para um dispositivo qualquer, o aplicativo é capaz de operar com vários dispositivos diferentes, desde que estes implementem esta interface, ou seja, forneçam o *plug-in* do equipamento.

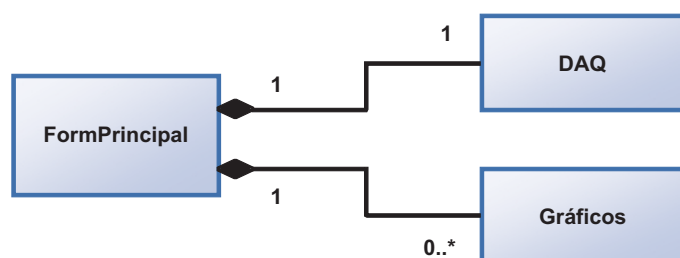


Figura 4.3: Diagrama de classes do software de visualização de sinais.

O formulário principal gerencia uma série de gráficos em um número que varia de acordo com a situação. O número de gráficos é igual ao número de canais ativados no equipamento, o que varia de acordo com o equipamento e a interação do usuário. Por este motivo, a relação de composição entre a classe *FormPrincipal* e os *Gráficos* é de um para muitos. Mais detalhes sobre o funcionamento destes gráficos serão mostrados adiante no texto.

Para escolher e configurar o equipamento que o aplicativo irá utilizar, o painel contém uma caixa de combinação com a lista dos equipamentos conectados ao computador e um controle *PropertyGrid* que irá exibir os parâmetros de configuração do dispositivo. O método *ProcurarDispositivos*, da classe *DAQFactory*, é utilizado para preencher a caixa de combinação com a lista dos equipamentos conectados ao computador. Isso é feito utilizando um código semelhante ao da Listagem 3.1 do capítulo anterior, sendo pesquisados os *plug-ins* no mesmo diretório do aplicativo. Quando o usuário escolhe um dos sistemas na caixa de combinação, o objeto *DAQ* é criado utilizando o método *CriarObjeto* da classe *DAQFactory*. Este objeto é associado ao *PropertyGrid*, através da propriedade

SelectedObject, para permitir a edição dos parâmetros de configuração do equipamento.

O segundo painel, para configurações dos canais, tem funcionamento semelhante ao primeiro painel. Ele possui uma caixa de combinação, contendo uma lista com os nomes dos canais do equipamento e um controle *PropertyGrid* para a configuração dos parâmetros dos canais. A caixa de combinação é preenchida com os nomes dos diversos canais, através da propriedade *Canal* do objeto *DAQ*. A maneira como isso feito é semelhante a demonstrada na Listagem 3.3 do capítulo anterior. O *PropertyGrid* também é associado ao canal selecionado, fornecendo um método de edição de seus parâmetros.

O último painel fornece dois botões: Um para iniciar e outro parar o processo de aquisição de dados. O botão “Iniciar” ativa o processo de aquisição do equipamento. Para isso, utiliza o método *IniciarAquisicao* do objeto *DAQ*. O botão “Parar” utiliza o método *PararAquisicao* da classe *DAQ* para interromper o processo de aquisição.

Na região direita, são exibidos uma série de gráficos com os sinais coletados. Cada gráfico corresponde ao sinal obtido por um canal do equipamento, sendo exibidos apenas os gráficos dos canais que estão habilitados. O software ajusta o número máximo de canais de acordo com a quantidade de canais disponíveis no equipamento.

Os gráficos apresentam o sinal captado através de uma curva no domínio do tempo. O eixo das abscisas representa o tempo, enquanto que o eixo das ordenadas corresponde à amplitude. O gráfico exibe uma janela fixa de dez segundos do sinal, sendo os valores mínimos e máximo deste eixo atualizados a cada dez segundos, de acordo com as amostras recebidas do sinal. O eixo das ordenadas é ajustado de acordo com a capacidade de medida de cada canal, obtida através das propriedades *ValorMax* e *ValorMin* do objeto *Canal*.

As amostras do sinal, utilizadas para a geração dos gráficos, são obtidas através dos *buffers* de cada canal, utilizando comandos semelhantes ao da Listagem 3.7 do capítulo anterior. Os gráficos são desenhados à medida que as amostras dos sinais são captadas pelo equipamento, fornecendo uma visualização em tempo real do sinal coletado.

Ao iniciar a execução do *BioScope*, é apresentada a janela para o usuário mostrada na Figura 4.4. Nela, tem-se os três painéis mencionados, e uma região em branco, à direita, reservada para os gráficos. Todos os controles dos painéis estão vazios, pois nenhum

equipamento foi ativado. Ao clicar na caixa de combinação com a lista dos equipamentos não é possível encontrar nenhum equipamento, já que não foram incluídos os *plug-ins* para o sistema. Nas próximas seções, serão adicionados novos *plug-ins* que fornecerão os dados a serem exibidos pelo aplicativo.

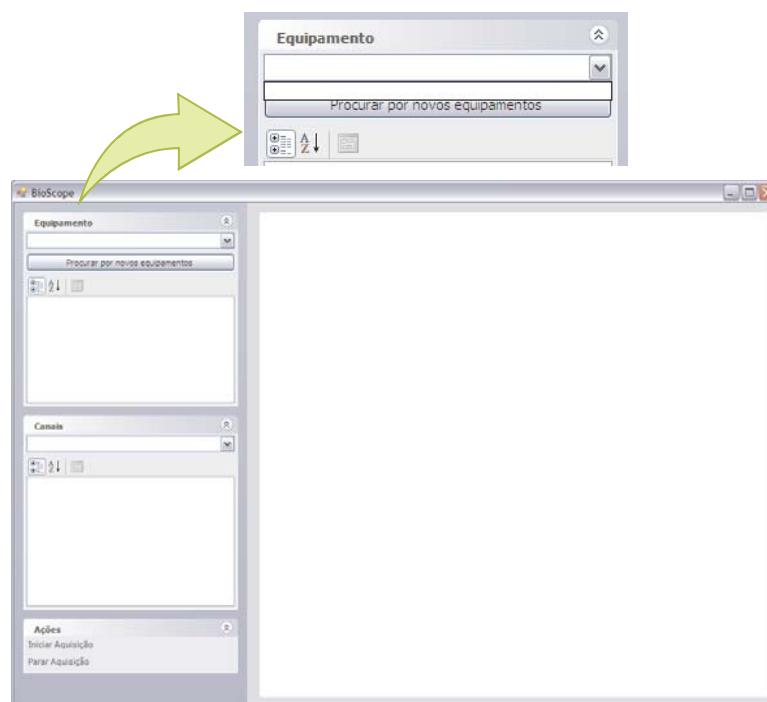


Figura 4.4: Aplicativo *BioScope* em execução. O detalhe mostra que nenhum equipamento foi encontrado.

4.2 O sistema virtual *Gerador Sinal*

O *plug-in GeradorSinal*, apresentado no capítulo anterior e cujo o código encontra-se no CD que acompanha este documento, pode ser utilizado com o aplicativo *BioScope*. Este *plug-in* simula o acesso a um equipamento de quatro canais, coletando ondas senoidais. Muitas empresas que vendem sistemas de aquisição de dados fornecem esse tipo de sistema virtual em conjunto com seus equipamentos. Estes equipamentos virtuais permitem a realização de testes sem o equipamento físico.

A Figura 4.5 apresenta um diagrama de blocos do equipamento virtual que o *plug-in* acessa. Os quatro canais estão conectados a um gerador de ondas senoidais, e cada canal pode fornecer um ganho diferente para o equipamento.

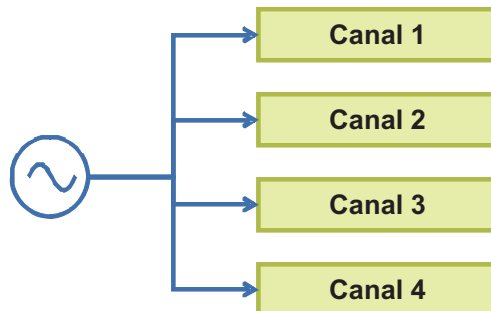


Figura 4.5: Diagrama de blocos do equipamento virtual acessado pelo *plug-in GeradorSinal*.

O *plug-in* foi copiado para o diretório do aplicativo *BioScope* e o aplicativo foi executado. A caixa de combinação com os equipamentos passou a apresentar dois novos itens (Figura 4.6), os dois itens correspondem aos dois equipamentos fornecidos pelo *plug-in GeradorSinal*. Ao selecionar o equipamento *GeradorSinal #1*, as propriedades do equipamento e de seu primeiro canal são exibidas nos editores de propriedades. Através do editor de propriedades do equipamento, foi alterada a frequência da senóide gerada para 3 Hz.

A Figura 4.7 mostra os passos para a configuração dos canais do sistema. O canal de número 1 foi selecionado no painel de canais e o editor de propriedades passa a exibir os seus parâmetros de configuração. Em seguida, o canal foi ativado ao modificar o valor da propriedade *IsEnabled* para *True*. O aplicativo *BioScope* detecta essa alteração e gera o gráfico correspondente ao canal na região direita. Continuando com o processo de configuração, o canal de número 3 foi selecionado. Ele também foi ativado, fazendo com que o aplicativo *BioScope* crie um novo gráfico para o canal. A propriedade *Ganho* também foi alterada para o valor 2.

Com a conclusão do processo de configuração, foi iniciada a aquisição de dados, clicando-se na área “Iniciar Aquisição” do terceiro painel. O aplicativo começa então,

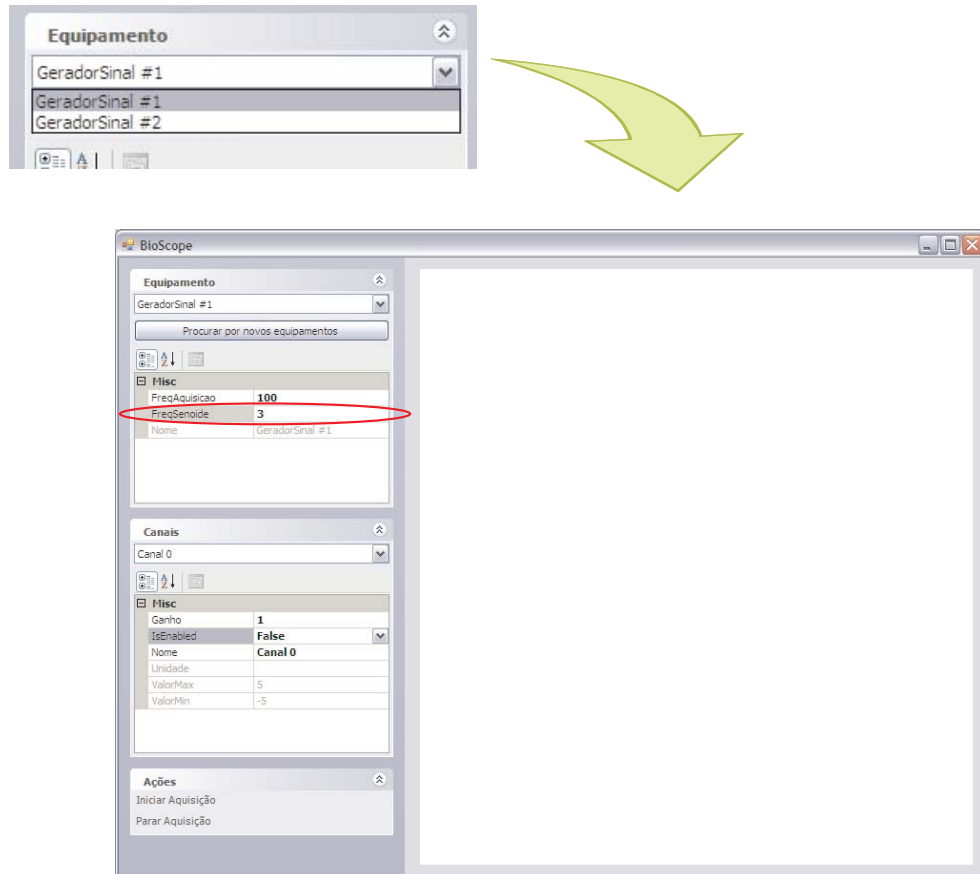


Figura 4.6: Ativando o equipamento *GeradorSinal #1* no aplicativo *BioScope*.

a exibir o sinal coletado nos dois gráficos, conforme pode ser observado na Figura 4.8. O canal 1 exibe uma senóide com 3Hz de frequência e amplitude unitária. O canal 3 também exibe uma senóide de 3Hz, mas com amplitude duas vezes maior que a do canal 1. Isso ocorre devido ao ajuste de ganho que foi feito neste canal na fase de configuração. Com esses experimentos, foi possível verificar o funcionamento do software *BioScope* com o *plug-in GeradorSinal*.

4.3 O equipamento *BioDAQ*

Numa terceira etapa, o software *BioScope* foi interfaceado com o equipamento *BioDAQ*. A Figura 4.9 apresenta uma foto da placa de circuito impresso do equipamento,

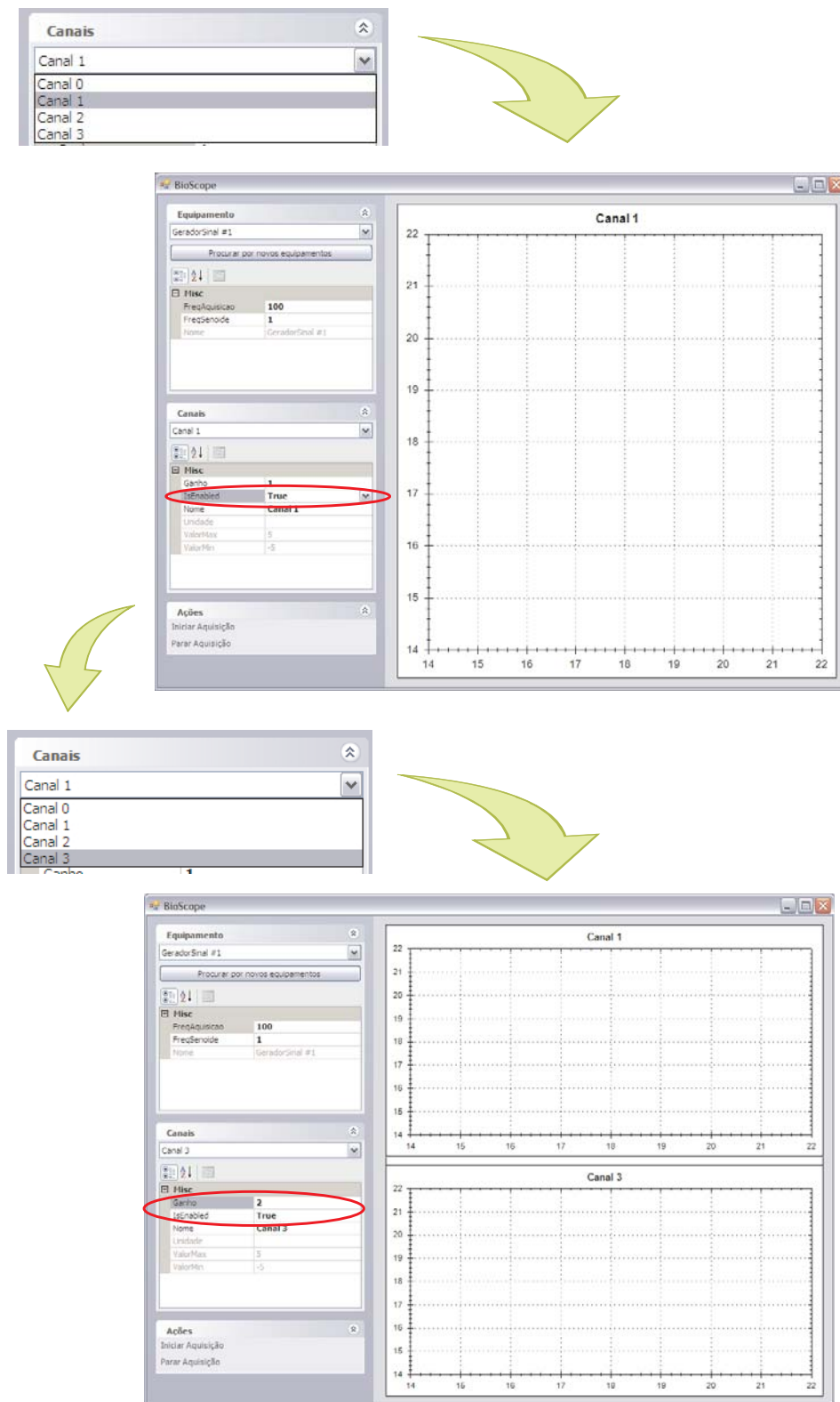


Figura 4.7: Configurando os canais do equipamento *GeradorSinal #1* no aplicativo *BioScope*.

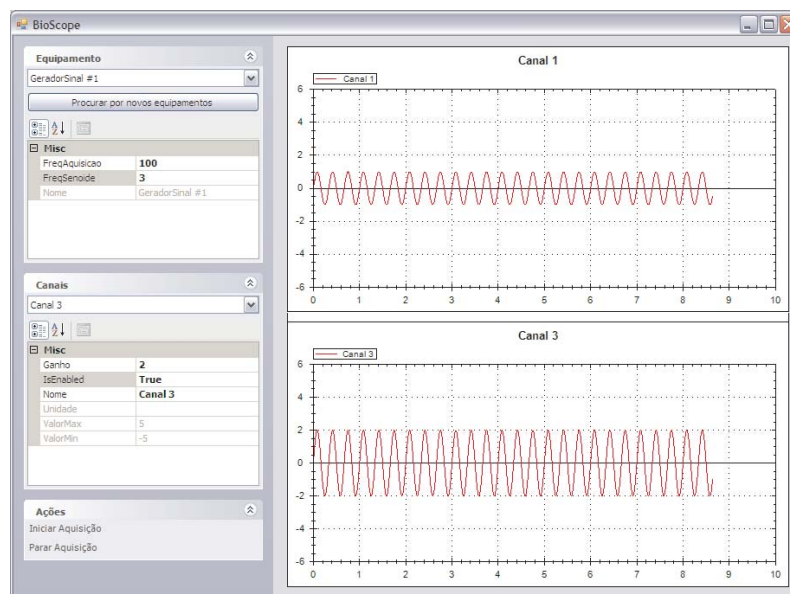


Figura 4.8: Processo de aquisição no aplicativo *BioScope* utilizando o *plug-in GeradorSinal*.

destacando seus principais componentes.

O equipamento é composto por uma interface USB para comunicação com o computador. A interface transfere os comandos enviados pela aplicação de software para o microcontrolador, responsável por gerenciar todo o sistema. O microcontrolador utiliza um canal digital isolado para obter as amostras de até três conversores AD. Esta isolamento é obtida pelo circuito situado na região direita. Na região inferior esquerda existem alguns circuitos de apoio para o restante do sistema.

O dispositivo é composto por três conversores AD, onde cada conversor é capaz de obter amostras de até oito canais diferentes, totalizando um máximo de 24 canais. Na imagem da foto, apenas o conversor central foi montado no equipamento. O conversor utilizado é capaz de medir tensões entre 0 e 5V com resolução de 16bits. O equipamento não inclui nenhum recurso de condicionamento de sinal.

A frequência de aquisição do equipamento pode ser modificada por software, podendo assumir os seguintes valores: 125Hz, 250Hz, 500Hz, 1khz, 2khz e 4khz. Essa frequência é controlador por um temporizador interno do microcontrolador.

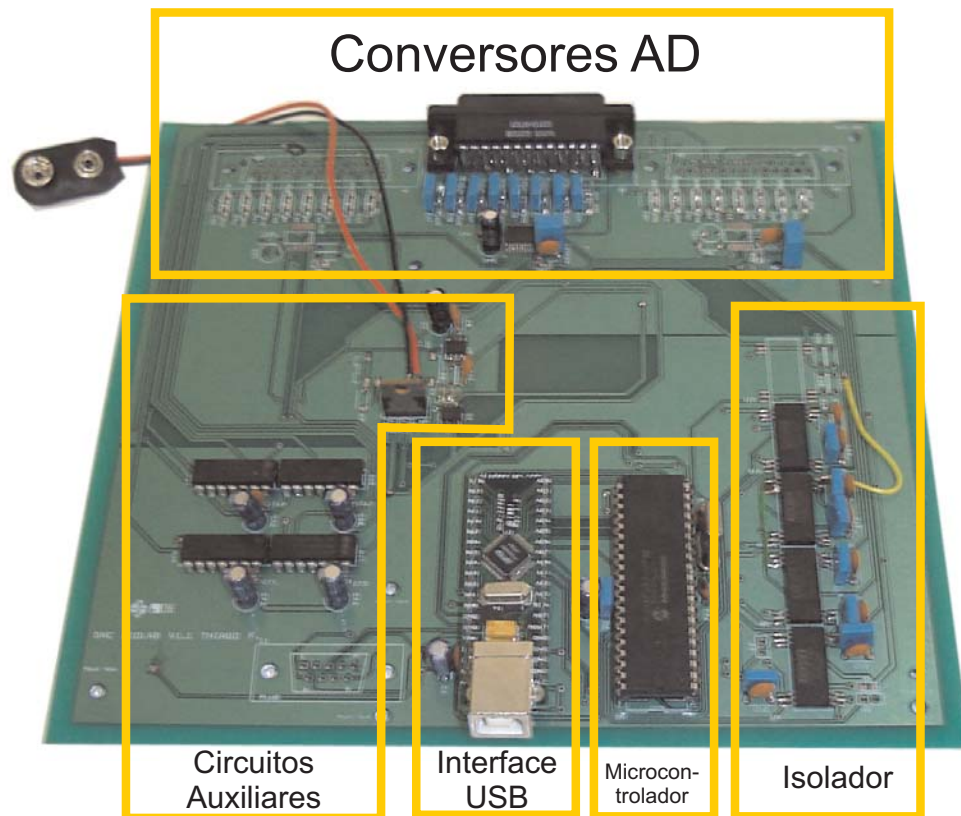


Figura 4.9: Foto da placa de circuito impresso principal do equipamento *BioDAQ* destacando seus principais componentes.

Para que o *BioDAQ* possa ser utilizado com o aplicativo *BioScope*, é preciso desenvolver seu *plug-in*. Esse *plug-in* deve utilizar rotinas de acesso ao sistema operacional ou bibliotecas fornecidas pelo fabricante do SAD para comunicar-se com o equipamento. No caso do *BioDAQ*, o fabricante da interface USB utilizada (FTDI Chip, Glasgow, UK) fornece uma biblioteca, desenvolvida em C, para acesso às suas funções. Como essa biblioteca não foi desenvolvida utilizando a plataforma *.net*, é necessária a utilização de algum sistema de interoperabilidade para que o *plug-in* acesse as funções da biblioteca.

A plataforma *.net* possui um recurso denominado *It Just Works* (DILASCIA, 2005) que pode fornecer essa camada de interoperabilidade. Esse recurso permite misturar código da plataforma *.net* com código nativo, utilizando a linguagem C++. Dessa forma, é possível criar uma classe da plataforma *.net* capaz de acessar estruturas e funções escritas em C,

gerando a camada de compatibilidade.

A Figura 4.10 mostra como o *plug-in* irá acessar as funções da biblioteca escrita em C. Ele utiliza a biblioteca *FTDIWrapper*, desenvolvida utilizando a linguagem C++ *.net*. A biblioteca *FTDIWrapper* possui classes adaptadoras, que funcionam como uma ponte entre o código da plataforma *.net* e o código nativo do sistema. A biblioteca então utiliza as rotinas do sistema operacional para comunicar-se com o equipamento.

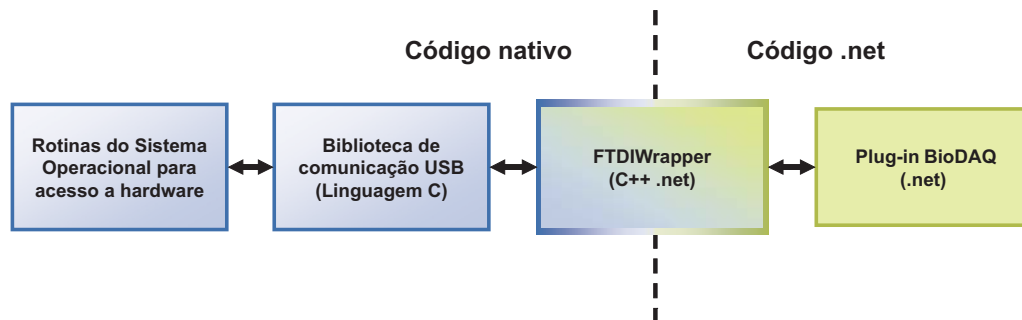
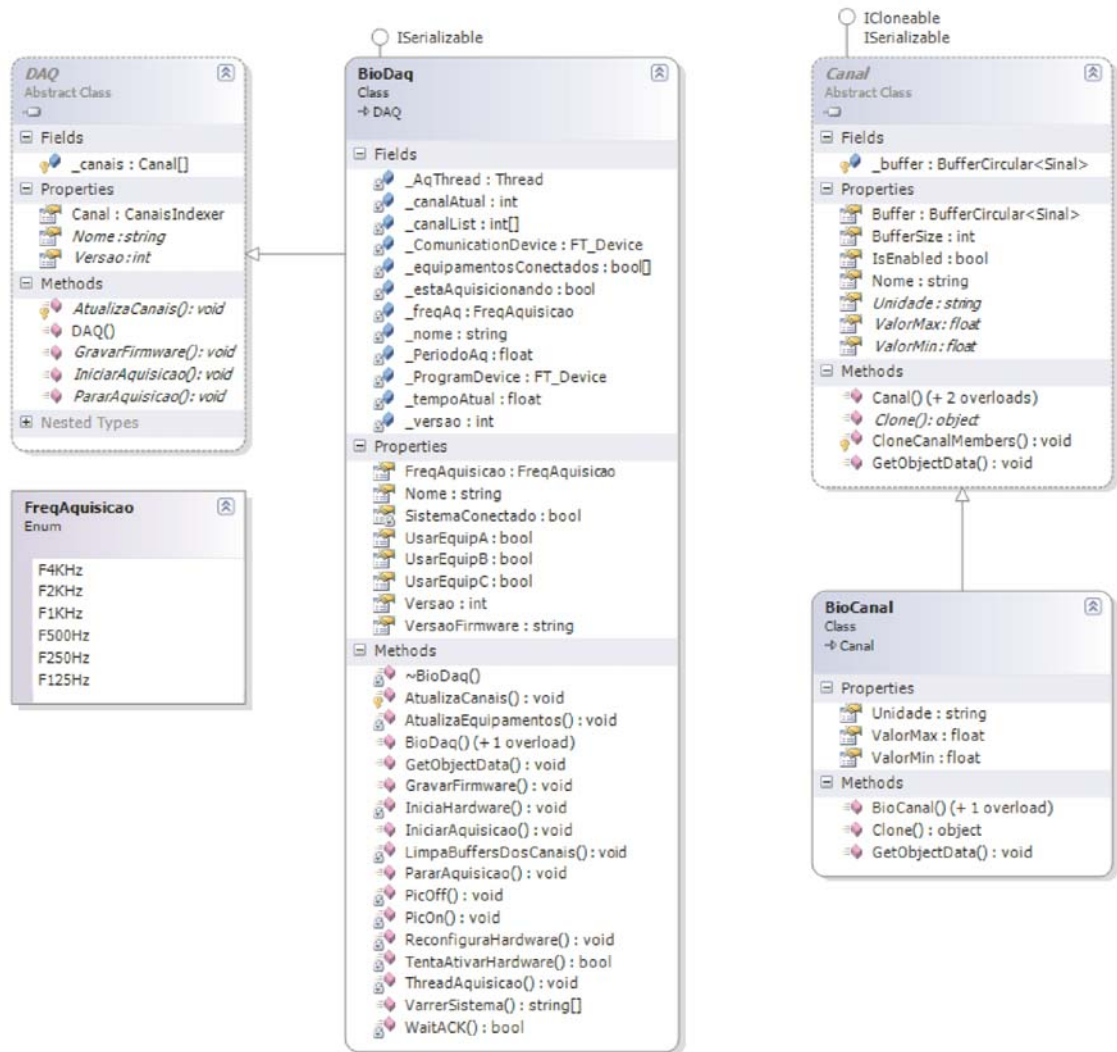


Figura 4.10: Diagrama de blocos mostrando como o *plug-in* acessa o equipamento.

A Figura 4.11 mostra o diagrama de classes do *plug-in* gerado pela IDE. Conforme foi visto no capítulo anterior, para a construção do *plug-in* são necessárias pelo menos duas classes: a classe *BioDAQ* que modela o equipamento, e a classe *BioCanal*, que fornece acesso aos recursos de cada canal do equipamento.

A classe *BioDAQ* é implementada seguindo as regras para criação de *plug-ins* descritas no capítulo anterior. Ela utiliza um objeto *FT_Device*, fornecido pela biblioteca *FTDIWrapper*, para comunicar-se com o equipamento. A classe fornece métodos que permitem ao *plug-in* enviar vetores de bytes com comandos para o equipamento. Ela também fornece métodos que obtêm vetores de bytes do equipamento com as respostas aos comandos e amostras dos sinais coletados.

A classe implementa ainda algumas propriedades utilizadas para configurar alguns recursos únicos do equipamento. A propriedade *FreqAquisicao* define qual o intervalo entre as amostras. Como mencionado anteriormente, o equipamento só é capaz de operar com algumas frequências de aquisição específicas. Essas frequências foram definidas na enumeração também chamada *FreqAquisicao*, e a propriedade utiliza essa enumeração

Figura 4.11: Diagrama de classes do *plug-in BioDAQ*.

como seu tipo de dado.

Além da propriedade *FreqAquisicao*, existem propriedades para ativar ou desativar a utilização dos sistemas de conversão AD (*UsarEquipA*, *UsarEquipB* e *UsarEquipC*). A propriedade *VersaoFirmware* fornece a versão do software de controle embutido no microcontrolador do equipamento e pode ser utilizada para indicar a necessidade de sua atualização. Os demais membros e métodos fornecem a funcionalidade interna do sistema.

Quando os valores dessas propriedades são modificados, o *plug-in* envia um comando para o equipamento, atualizando o seu estado. Por exemplo, ao modificar a propriedade *FreqAquisicao*, a classe *BioDAQ* gera um vetor de bytes contendo o comando e os parâmetros para a configuração da frequência de aquisição no equipamento. A classe então utiliza o seu membro *_communicationDevice*, da classe *FT_Device*, para transmitir o vetor de bytes para o equipamento, alterando a frequência de aquisição no mesmo.

A classe *BioCanal* segue a implementação do *plug-in*, modelando os canais do equipamento. Ela sobrescreve o a propriedade *Unidade*, retornando a string “V” representando a unidade Volts, que os canais do equipamento são capazes de coletar. Os canais são capazes de medir valores entre 0 e 5 Volts, de forma que as propriedades *ValorMin* e *ValorMax* retornam estes valores respectivamente. A construção dessas classes conclui o desenvolvimento do *plug-in* que poderá ser utilizado com o software *BioScope* para visualização de sinais.

O arquivo *dll* do *plug-in* foi copiado para o diretório do aplicativo *BioScope*. Essa operação foi feita sem fechar o aplicativo *BioScope*. O equipamento foi conectado a uma das portas USB do computador, e em seguida clicou-se no botão “Procurar Dispositivos”. O *BioScope* então voltou para o seu estado inicial, com todos os painéis vazios, entretanto, ao listar os dispositivos conectados ao equipamento foi encontrado um novo equipamento (Figura 4.12).

O equipamento *BioDAQ* foi selecionado, ativando o sistema. Os painéis foram atualizados de acordo com os recursos do equipamento, como mostra a Figura 4.13.

Com o equipamento já em operação com o aplicativo *BioScope*, foram realizados alguns testes para avaliar a capacidade do sistema de coletar um sinal real. Nestes testes, um

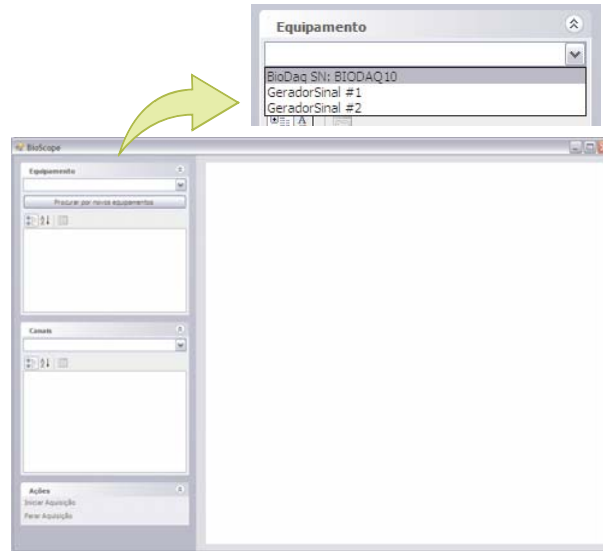


Figura 4.12: *BioScope* após a pesquisa por novos equipamentos. O detalhe mostra que o aplicativo detectou a presença do dispositivo *BioDAQ*

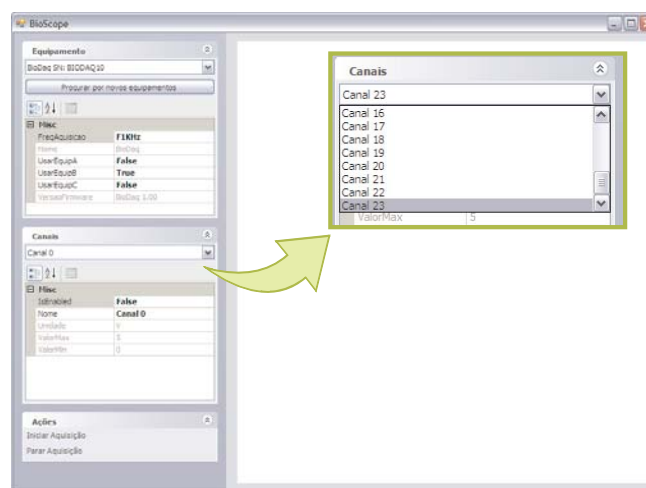


Figura 4.13: *BioScope* após ativar o equipamento *BioDAQ*. O detalhe mostra a lista com os canais do equipamento, totalizando 24 canais (Numerados de 0 até 23).

dos canais do *BioDAQ* foi conectado a um gerador de sinal. Um osciloscópio digital foi conectado em paralelo, permitindo comparar o sinal captado pelo *BioScope*, com o obtido no osciloscópio. A Figura 4.14 apresenta um diagrama de blocos da conexão e a Figura

4.15 mostra uma foto da montagem do experimento.

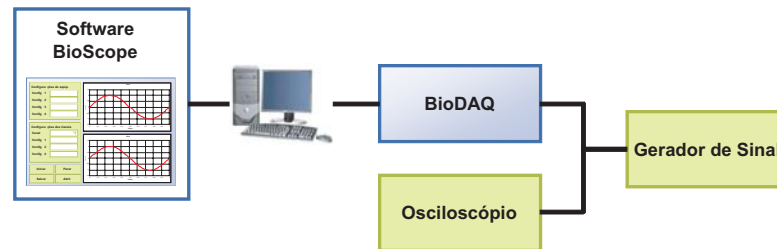


Figura 4.14: Conexão dos equipamentos durante o experimento com o *BioDAQ*.

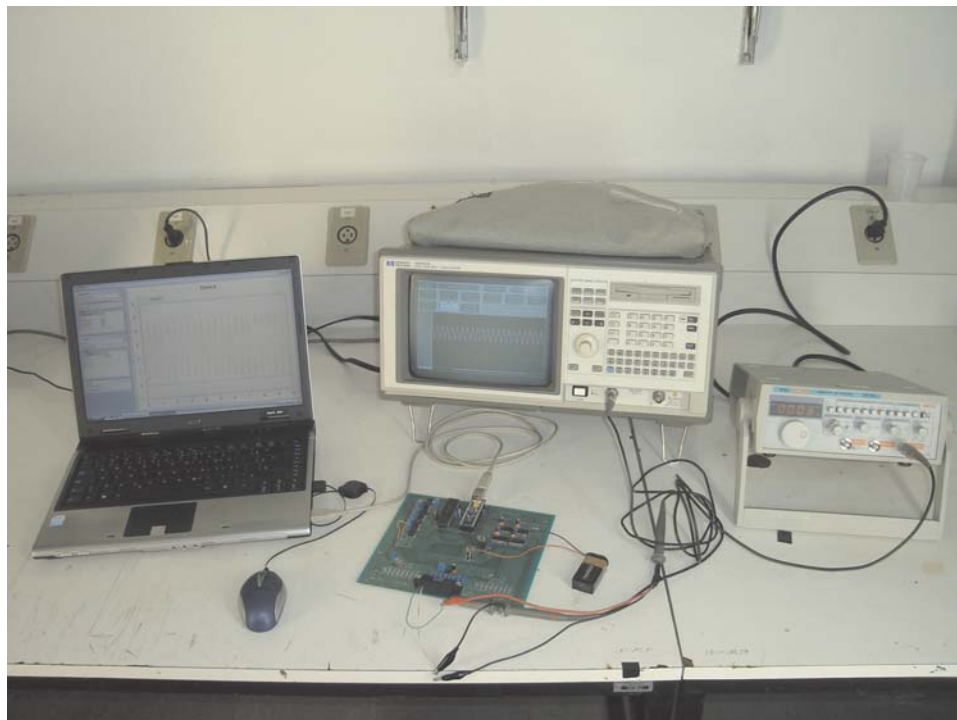


Figura 4.15: Montagem do experimento com o *BioDAQ*.

Após a montagem física, o aplicativo *BioScope* foi utilizado para configurar o equipamento. A Figura 4.16 ilustra os passos realizados durante este processo. O equipamento foi configurado para realizar a aquisição em 1khz através da propriedade *FreqAquisicao*. Note que a propriedade *FreqAquisicao* exibe uma lista com as frequências que o equipamento suporta. Isso ocorre, porque foi usada uma enumeração no tipo de dado e, por

reflexão, o controle *PropertyGrid* obtém os valores da enumeração. Em seguida, foi selecionado o canal 8 e este foi ativado de maneira análoga ao que foi feito com o *GeradorSinal*. Ao final do processo, o aplicativo apresenta o gráfico que irá conter o sinal coletado e está pronto para iniciar o processo de aquisição.

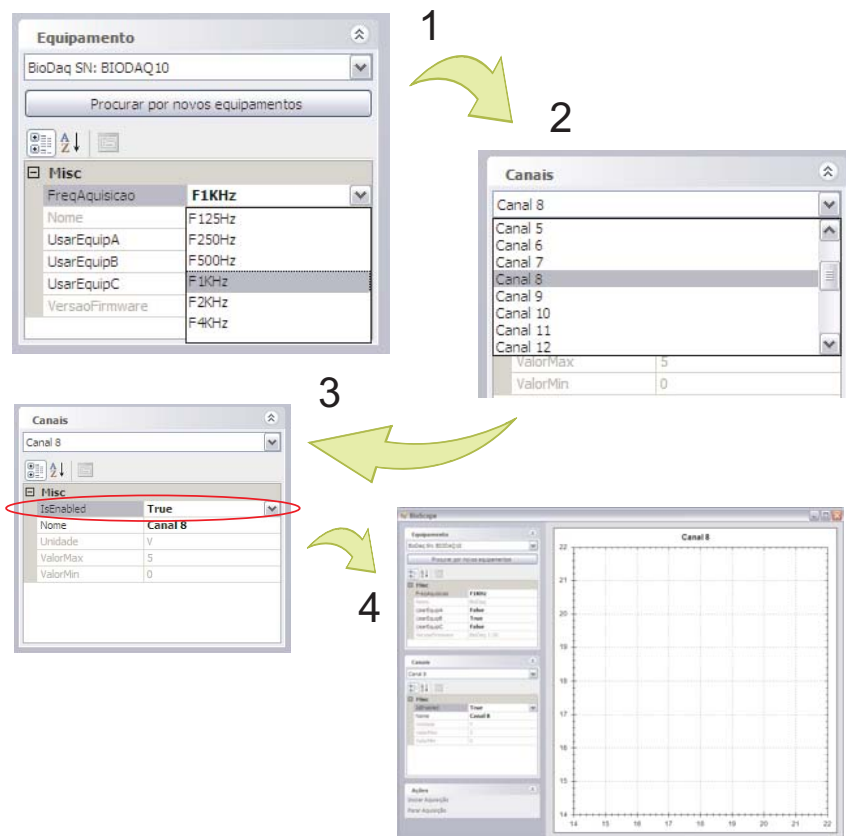


Figura 4.16: Passos ilustrando a configuração do *BioDAQ*. (1) Configurando a frequência de aquisição, (2) selecionando o canal 8, (3) ativando o canal, (4) resultado final com o gráfico do canal 8 na região direita.

O gerador de sinal foi ajustado para gerar uma onda senoidal com frequência de 3 Hz, com 4,2 V de amplitude pico a pico e offset de 2,5V. O osciloscópio foi configurado com a escala de 4V por divisão para amplitude do sinal e 1 segundo por divisão no tempo. Em seguida, foi iniciado o processo de aquisição de dados no software. Foram verificados os gráficos produzidos, comparando com os valores definidos no gerador de sinal e com os obtidos através do osciloscópio. A Figura 4.17 mostra o sinal capturado no software e o

obtido no osciloscópio, onde é possível verificar que ambos estão em conformidade com os parâmetros do sinal gerado. Em seguida, foi modificada a frequência e a forma de onda dos sinais, sendo verificada a ocorrência das mudanças nos dois equipamentos (Figura 4.18).

4.4 *Myosystem-Br1 P42*

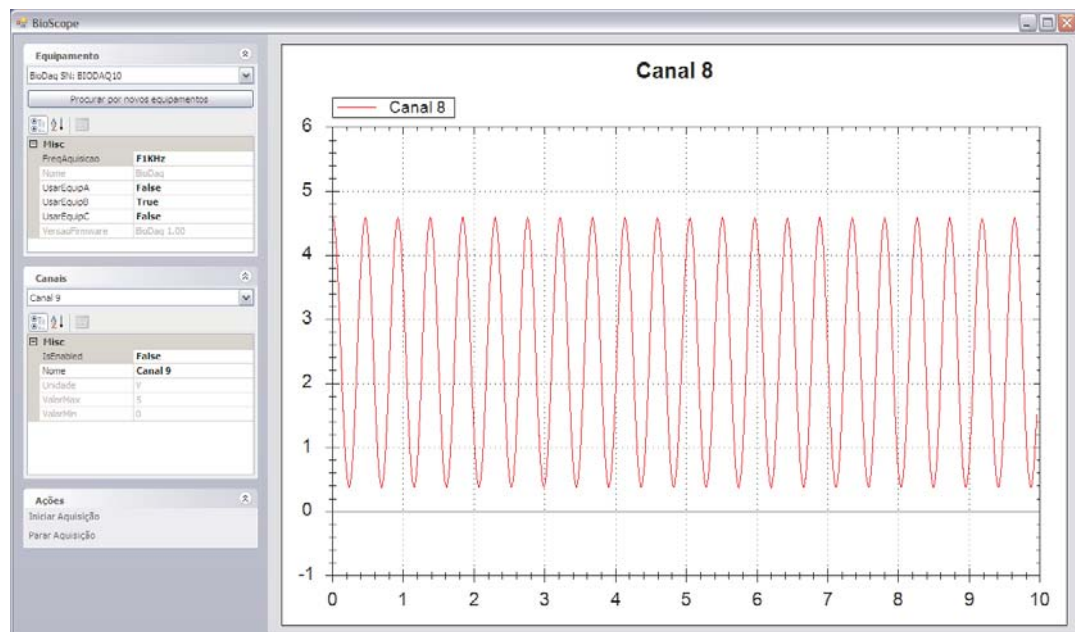
O Myosystem-Br1 P42 (Datahominis Tecnologia Ltda, Uberlândia - MG) é um equipamento para coleta de sinais eletromiográficos, criado para auxiliar estudos da atividade muscular realizados por diversos profissionais da área de saúde. O sistema possui quatro canais dedicados a sinais EMG e dois canais auxiliares para conexão com outros sensores (como células de carga e eletrogoniômetros). A Figura 4.19 apresenta uma foto do equipamento.

Assim como o *BioDAQ*, a comunicação entre computador e o equipamento também é feita utilizando o barramento USB. O equipamento possui um microcontrolador que recebe os comandos da interface USB e controla os diversos elementos do sistema. Esse microcontrolador também está conectado a um conversor AD de 16 bits capaz de obter as amostras do sinal presente nos seis canais do sistema.

O equipamento pode realizar a aquisição de dados utilizando frequências de aquisição de 2 ou 4 kHz, sendo esta configurada através do software. Ele ainda possui um gatilho externo, que pode ser utilizado para sincronizar a aquisição com outro dispositivo.

Ao contrário do *BioDAQ*, este equipamento inclui condicionadores de sinais em seus canais, sendo que os canais EMG e auxiliares possuem condicionadores diferentes. Para os canais EMG, o equipamento possui um filtro passa alta com frequência de corte situada em 15Hz, um filtro passa baixa sintonizado em 1kHz, além de dois amplificadores programáveis. O ganho desses amplificadores pode ser configurado por software, assumindo valores entre 1 e 800 vezes, quando combinados. Para a captação dos sinais EMG pelo equipamento, são utilizados eletrodos ativos que também fornecem uma pré-amplificação de 20 vezes. Dessa forma, é possível obter um ganho total de 16000 vezes. Os canais EMG são isolados galvânicamente do restante do sistema, o que é necessário para a segurança

a)



b)

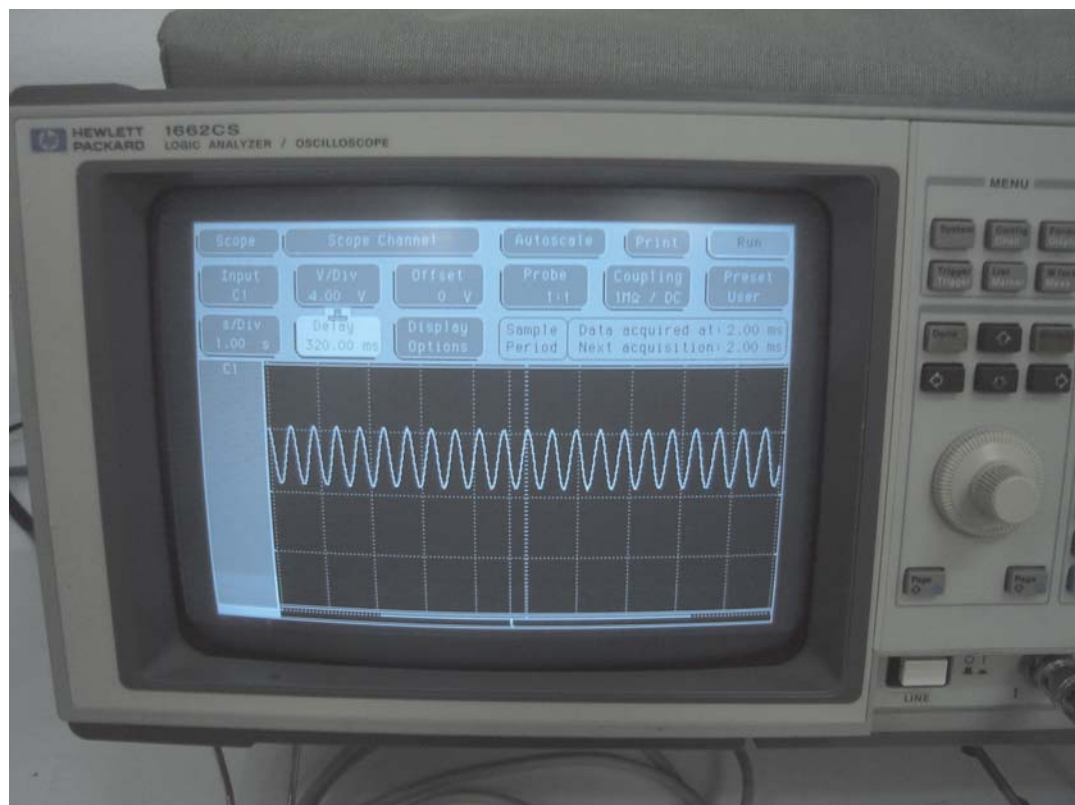


Figura 4.17: a) Sinal obtido pelo *BioScope* através do sistema de aquisição *BioDAQ*. b) Sinal obtido pelo osciloscópio digital.

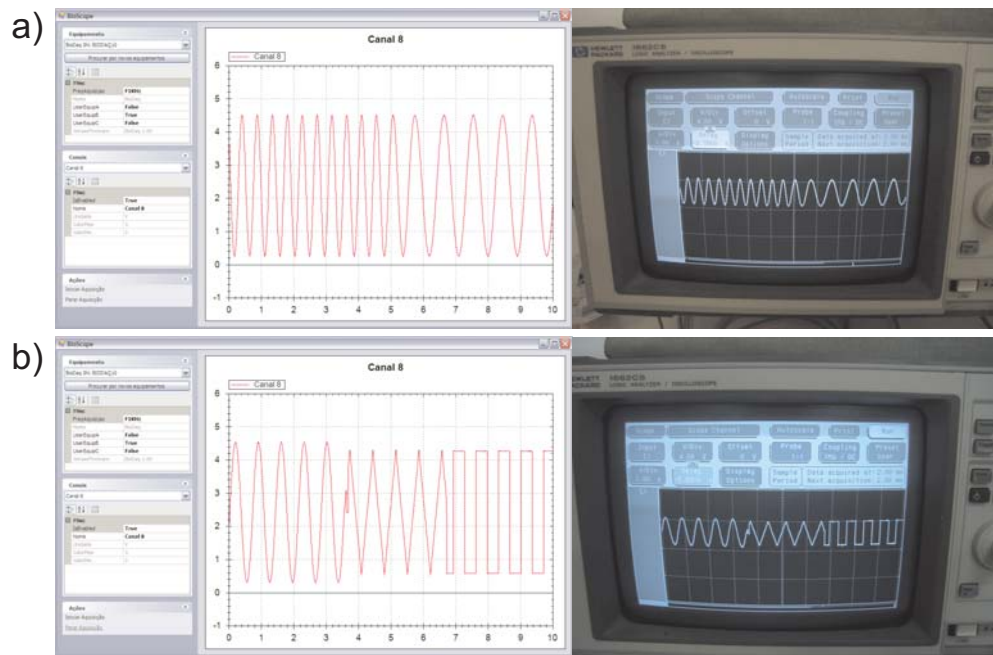


Figura 4.18: Comparação entre o sinal obtido com o *BioScope* e o osciloscópio digital ao variar a frequência (a) e a forma de onda (b) do sinal.



Figura 4.19: Myosystem-Br1 P42.

ao voluntário.

Os canais auxiliares são utilizados para a conexão de diversos sensores como, por exem-

plo, células de carga. Estes canais possuem apenas um único amplificador programável com ganho máximo de 100 vezes, e não possuem o filtro passa alta, permitindo a coleta de sinais DC. O filtro passa baixa possui frequência de corte situada em 1khz.

Novamente, para que o software *BioScope* possa utilizar este equipamento para a visualização de sinais eletromiográficos é preciso construir um *plug-in* do equipamento. O desenvolvimento do *plug-in* ocorre da mesma forma que o *BioDAQ*. Como o sistema utiliza a mesma biblioteca para a comunicação com a interface USB, serão reutilizadas as classes da biblioteca *FTDIWrapper*. A Figura 4.20 apresenta o diagrama de classes do *plug-in MyoSystem-Br1 P42*.

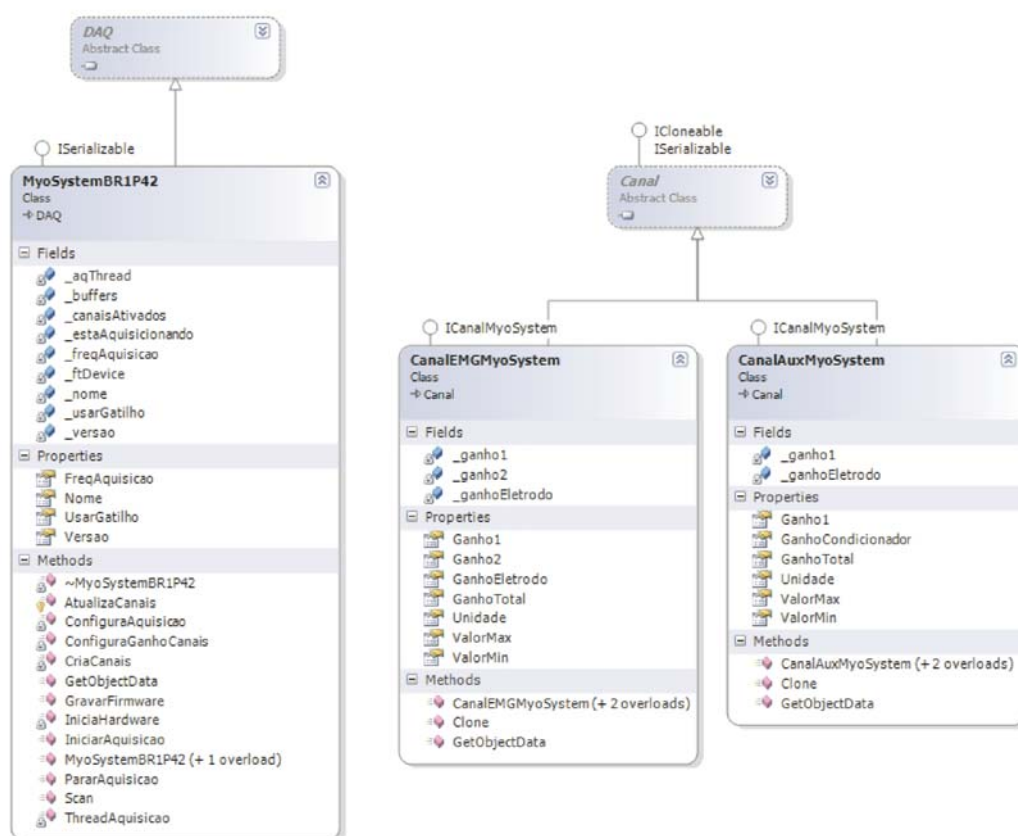


Figura 4.20: Diagrama de classes do *plug-in MyoSystem-Br1 P42*.

A classe *MyoSystemBR1P42* representa o equipamento, herdando de *DAQ*. Ela sobreescreve os membros herdados de *DAQ* e adiciona os recursos únicos do equipamento

através de propriedades. A propriedade *FreqAquisicao* configura o intervalo de tempo entre duas amostras e a propriedade *UsarGatilho* define se a aquisição será sincronizada com um equipamento externo.

Como o equipamento possui dois tipos de canais diferentes foram criadas as classes *CanalEMGMyoSystem* e *CanalAuxMyosystem*. A classe *CanalEMGMyoSystem* modela os canais EMG. Ela possui as propriedades *Ganho1*, *Ganho2* e *GanhoEletrodo*. Essas propriedades são utilizadas para definir os ganhos dos dois amplificadores programáveis, presentes no condicionador de sinal do canal, e do pré-amplificador contido no eletrodo. A propriedade *GanhoTotal* é uma propriedade que informa o ganho total que o sistema fornece no canal. A classe ainda sobrescreve as propriedades de *Canal* fornecendo o valor máximo e mínimo que o canal é capaz de medir de acordo com os ganhos e a unidade de medida.

A classe *CanalAuxMyoSystem* representa os canais auxiliares. O ganho do canal pode ser definido na propriedade *Ganho* e é possível indicar o ganho embutido em um condicionador externo ao equipamento através da propriedade *GanhoCondicionador*. O restante dos elementos da classe são as sobrecargas dos elementos herdados da classe *Canal*.

Após o desenvolvimento, o *plug-in* foi copiado para o diretório do aplicativo *BioScope*. O equipamento foi conectado ao computador e o aplicativo *BioScope* foi iniciado, exibindo a sua janela principal. Ao listar os equipamentos disponíveis no computador, foi apresentado um novo item referente ao *Myosystem-Br1 P42* (Figura 4.21).

Ao selecionar o *MyoSystem-Br1 P42* na lista de equipamentos é estabelecida a comunicação com o mesmo. A Figura 4.22 mostra que os painéis foram atualizados, passando a exibir as propriedades referentes aos parâmetros de configuração únicos do *MyoSystem-Br1 P42*.

A lista de canais também foi atualizada com os nomes dos canais do equipamento. O *Myosystem-Br1 P42* possui a peculiaridade de possuir dois tipos de canais diferentes (EMG e Auxiliares), ainda assim o aplicativo é capaz de reconhecer e configurar esses dois tipos de canais. A Figura 4.23 mostra como o editor de propriedades é modificado à medida que um tipo diferente de canal é selecionado.

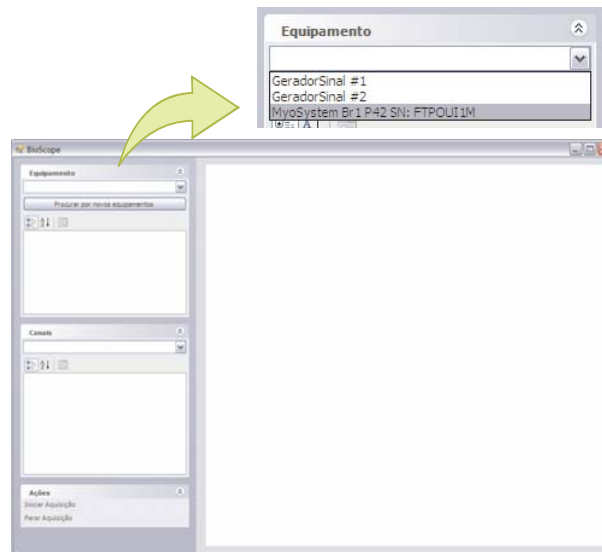


Figura 4.21: *MyoSystem-Br1 P42* na lista de equipamentos do aplicativo *BioScope*.

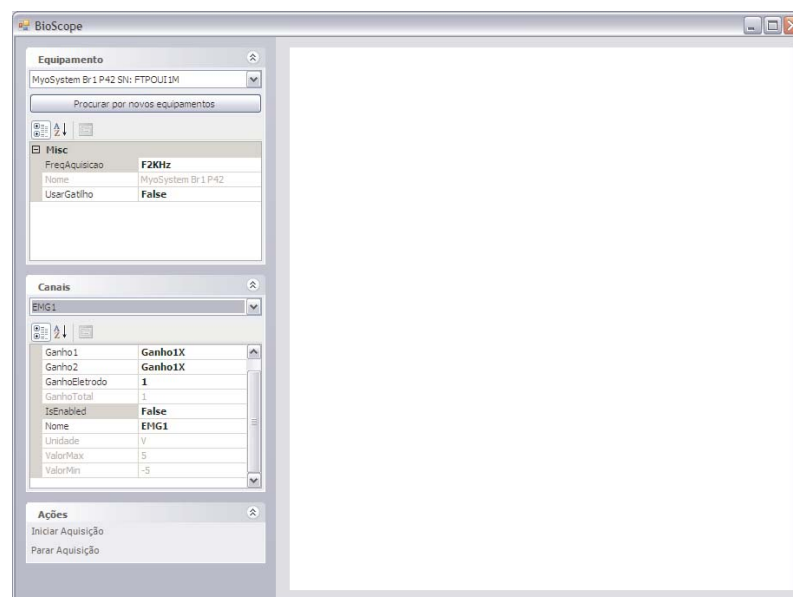


Figura 4.22: Painéis apresentando as propriedades do *MyoSystem-Br1 P42*.

O *MyoSystem-Br1 P42* foi submetido aos mesmos experimentos realizados anteriormente com o *BioDAQ*. Um dos canais auxiliares do equipamento foi conectado a um gerador de sinais e, em paralelo, foi conectado um osciloscópio para comparação. O gerador de sinal foi configurado para gerar uma onda senoidal de 3 Hz com 3 volts de pico a

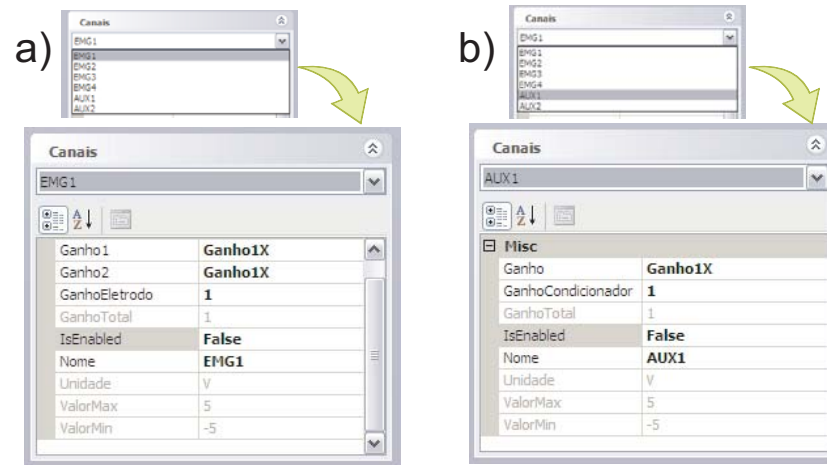


Figura 4.23: Editor de propriedades do canal no aplicativo *BioScope*. Em (a) foi selecionado um canal EMG e em (b) foi selecionado um canal auxiliar.

pico e offset de 2,5 volts. A configuração do osciloscópio permaneceu a mesma do experimento anterior, sendo utilizada as escalas de 4V por divisão na amplitude e 1 segundo por divisão no tempo. A Figura 4.24 apresenta uma imagem da montagem do experimento.



Figura 4.24: Montagem do experimento com o equipamento *Myosystem-Br1 P42*.

No aplicativo *BioScope* foi configurada a frequência de aquisição do equipamento em 2 khz. O sinal foi conectado ao canal auxiliar 1 do equipamento, sendo este ativado e configurado no aplicativo para fornecer ganho unitário. A Figura 4.25 ilustra os passos

de configuração do equipamento.

O processo de aquisição foi então iniciado, exibindo o sinal coletado. Este sinal encontrava-se em conformidade com o sinal definido no gerador de sinal e com o exibido no osciloscópio. Os procedimentos do experimento anterior foram refeitos com o *Myosystem-Br1 P42*, variando a frequência e a forma de onda no gerador de sinal. O sinal obtido no sistema e no osciloscópio podem ser vistos na Figura 4.26.

O *BioScope* e o *Myosystem-Br1 P42* também foi utilizado para visualização de um sinal EMG. Neste experimento, foram utilizados os seguintes materiais:

- O equipamento *MyoSystem-Br1 P42*.
- Um computador com o aplicativo *BioScope* e o *plug-in* do equipamento *MyoSystem-Br1 P42* instalados.
- Um eletrodo ativo duplo diferencial com ganho de 20 vezes.
- Um eletrodo de referência.
- Gel condutivo.
- Fita adesiva para fixação dos eletrodos.

Para a fixação dos eletrodos, foi realizada uma limpeza do local onde eles seriam posicionados. O eletrodo ativo foi fixado no músculo flexor de punho, e um eletrodo de referência no ombro do voluntário. O eletrodo ativo foi posicionado de modo que os seus contatos (fios de 10mm de comprimento) estavam situados sobre uma reta imaginária perpendicular ao sentido das fibras musculares. Antes da fixação do eletrodo de referência, foi acrescentada uma pequena quantidade de gel no eletrodo, visando melhorar o contato elétrico deste com a pele. Em seguida, os eletrodos foram conectados ao equipamento para dar início ao processo de coleta. A Figura 4.27 apresenta o voluntário, após a fixação dos eletrodos para a realização do experimento.

O eletrodo foi conectado ao canal EMG 1 do equipamento. No aplicativo *BioScope*, foi ativado o equipamento *Myosystem-Br1 P42*, definida a frequência de aquisição em 2

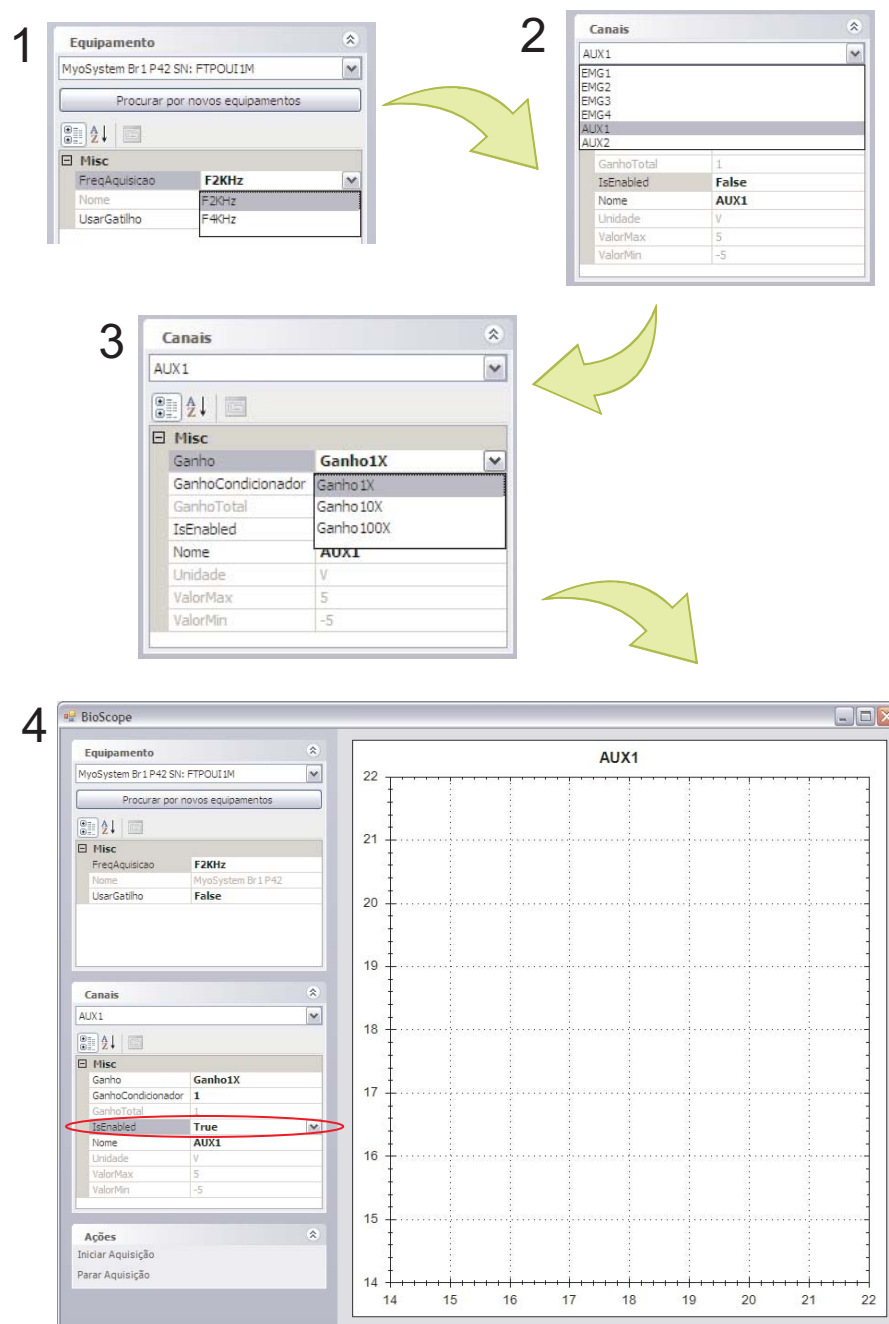


Figura 4.25: Passos de configuração do *Myosystem-Br1 P42*: (1) Configurando a frequência de aquisição em 2khz; (2) Selecionando o canal auxiliar; (3) Configurando o ganho do canal; (4) Ativando o canal. Ao final do processo, o gráfico do canal auxiliar é exibido na região direita.

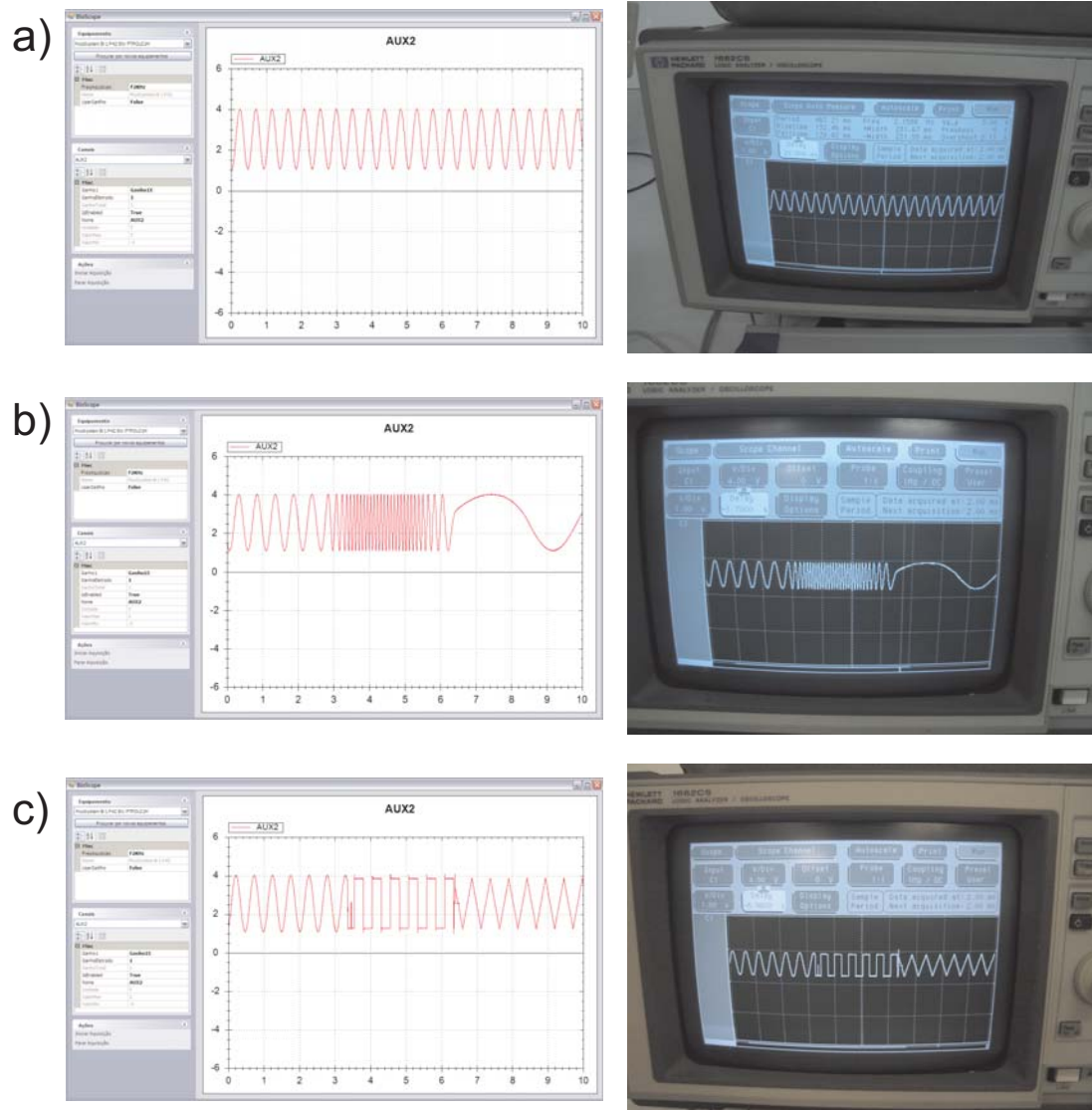


Figura 4.26: Formas de ondas capturadas pelo *BioScope* através do *Myosystem-Br1 P42* comparadas com as obtidas pelo osciloscópio: a) Uma senóide pura; b) Variando a frequência da senóide; c) Variando a forma do sinal.



Figura 4.27: Eletrodos fixados no voluntário

khz e ativado o canal EMG 1. Os dois ganhos do canal, foram ajustados em 100 vezes e 4 vezes, respectivamente. O ganho do eletrodo foi definido em 20 vezes, totalizando um ganho de 8000 vezes. Em seguida, foi iniciado o processo de aquisição, sendo requisitado ao voluntário a realização de uma contração muscular. O sinal capturado pelo sistema é apresentado na Figura 4.28, onde a escala de amplitude está em microvolts.

4.5 Atualizações no sistema

Uma das dificuldades no desenvolvimento de software ocorre quando surge alguma alteração após o seu desenvolvimento, exigindo a manutenção de código. A manutenção é um processo dispendioso, principalmente se o software for mal projetado (BROOKS, 1987). O *framework DAQSystem* facilita a manutenção do software e essa situação será simulada, em duas situações. Na primeira, um novo recurso será adicionado ao equipamento *BioDAQ* e serão verificadas as mudanças ocorridas no aplicativo *BioScope*. Na segunda situação, o aplicativo *BioScope* será modificado e serão verificadas as influências geradas nos três equipamentos.

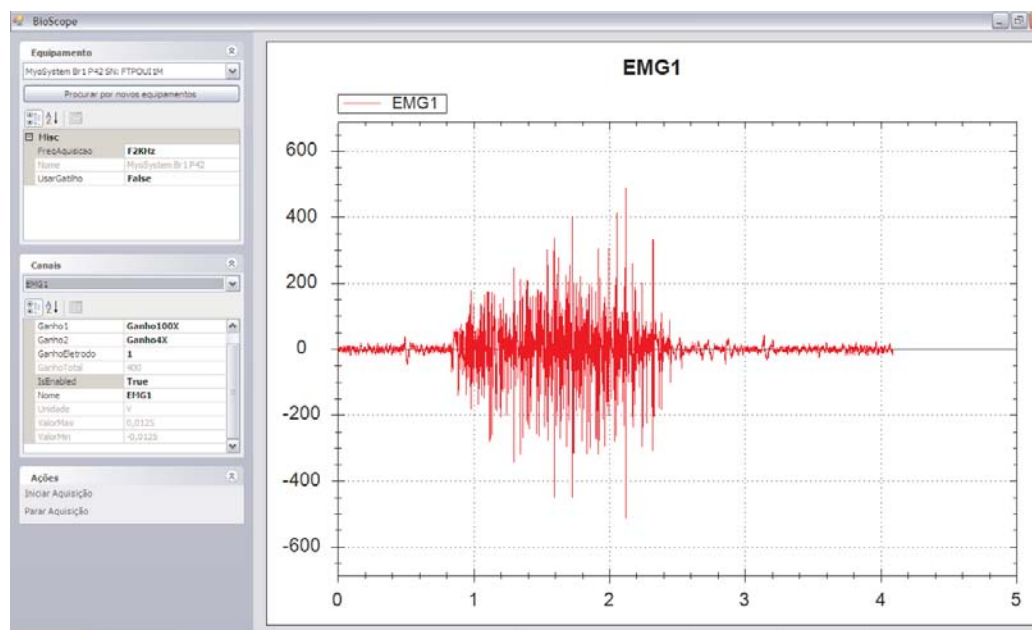


Figura 4.28: Aplicativo *BioScope* exibindo o Sinal EMG capturado através do equipamento *Myosystem-Br1 P42*

4.5.1 Atualizando o *BioDAQ*

Para simular uma atualização em um equipamento, foi adicionado ao *BioDAQ* um gatilho externo para sincronização com outros equipamentos. Este gatilho é ativado através de um sinal digital que chega até o microcontrolador. A modificação no equipamento foi feita atualizando o seu *firmware*, acrescentando novas instruções, para que o disparo do processo aquisição possa ser feito através de um gatilho externo.

Em seguida, é necessário modificar o *plug-in*, para incluir o novo recurso. Foi então adicionada a propriedade *UsarGatilho* à classe *BioDAQ*, fornecendo acesso ao recurso. A Figura 4.29 mostra a nova propriedade na classe *BioDAQ*. Em seguida, o *plug-in* foi recompilado e o arquivo *dll* gerado foi copiado para o diretório do aplicativo *BioScope*. Ao conectar-se ao equipamento, o aplicativo já reconhece o novo recurso e exibe a propriedade *UsarGatilho* nas configurações do equipamento (Figura 4.30).

Note que não foi necessária nenhuma alteração no aplicativo *BioScope* ou nos outros *plug-ins*. O aplicativo *BioScope* sequer foi recompilado, demonstrando que uma modi-

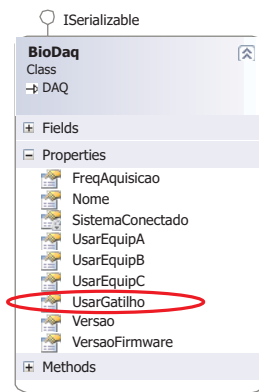


Figura 4.29: Classe *BioDAQ* com a propriedade *UsarGatilho*.

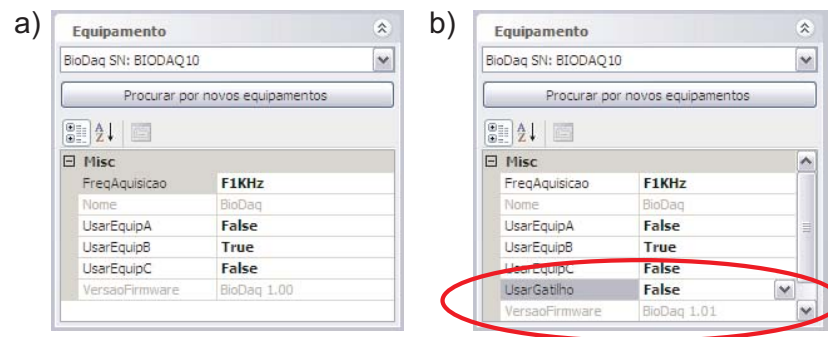


Figura 4.30: Propriedades do equipamento *BioDAQ* antes(a) e depois (b) da modificação. Note a mudança na versão do firmware retornada (foi modificada de 1.00 para 1.01) e a nova propriedade, *UsarGatilho*.

ficção do equipamento ocorre de maneira independente do software de processamento. Todo o sistema continuou funcional, podendo ser realizadas aquisições com este, ou com os outros equipamentos, normalmente.

4.5.2 Atualizando o *BioScope*

Num segundo momento, software *BioScope* foi modificado para avaliar o efeito que modificações no software de processamento geram nos equipamentos. Uma nova versão foi criada para permitir o acesso a mais de um equipamento simultaneamente. Para

isso, foi adicionado um novo painel, que apresentará o nome dos diversos equipamentos ao lado de caixas de checagem (Figura 4.31). Para ativar um equipamento, o usuário deverá marcar a sua respectiva caixa de checagem. Na configuração do equipamento, aparecerão apenas os equipamentos ativados, e o restante da operação continua como na versão anterior do aplicativo.

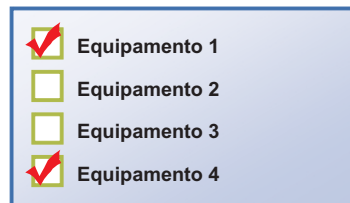


Figura 4.31: Exemplo de painel para seleção de dispositivos utilizados.

O *framework DAQSystem* facilita o desenvolvimento de softwares que integrem vários equipamentos. Isso ocorre porque o aplicativo “enxerga” todos os equipamentos, como sendo do mesmo tipo. Dessa forma, para atender o novo requisito proposto, foi modificada a estrutura do software, conforme o diagrama de classes da Figura 4.32. A diferença é que agora o relacionamento entre o *FormPrincipal* e o *DAQ* é de um para muitos.

O aplicativo mantém a referência para os objetos *DAQ* numa lista. A configuração dos equipamentos é feita passando a referência dos objetos *DAQ* e *Canais* para os controles *PropertyGrid*. Ao iniciar ou parar o processo de aquisição, o aplicativo simplesmente percorre a lista de objetos *DAQ* executando as funções *IniciarAquisicao* ou *PararAquisicao*.

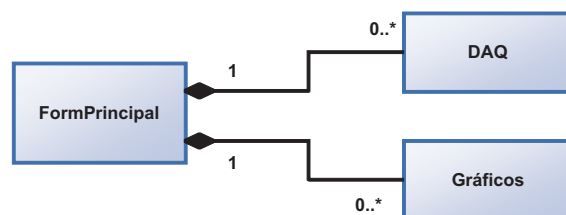


Figura 4.32: Diagrama de classes do aplicativo *BioScope* após a modificação.

A Figura 4.33 mostra o software após a atualização. O aplicativo possui um novo painel na região superior, denominado “Equipamentos Utilizados”. Este painel é utilizado para

ativar ou desativar os diversos equipamentos, sendo atualizado, assim que o software é iniciado ou quando o usuário pressiona o botão “Procurar por novos equipamentos”.

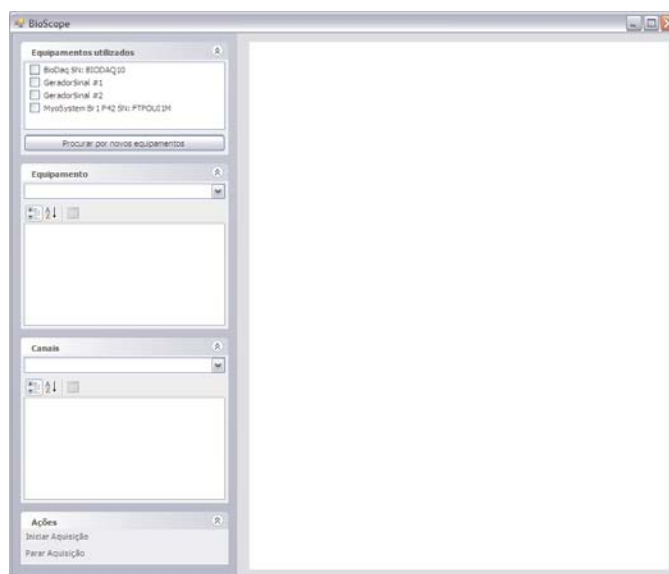


Figura 4.33: Aplicativo *BioScope* sendo executado após a modificação. O seu novo painel apresenta os vários equipamentos encontrados.

Para utilizar um dos equipamentos, o usuário marca a caixa de checagem ao lado do nome do equipamento. Para desconectar o equipamento, a operação inversa deve ser feita, desmarcando a caixa de checagem do equipamento. A Figura 4.34 apresenta um exemplo de utilização. No momento da execução, os equipamentos *BioDAQ* e *Myosystem-Br1 P42* estavam conectados ao computador e foram ativados. Observe que a lista com os equipamentos apresenta apenas os dois equipamentos ativados.

À medida que o usuário seleciona um dos sistemas, as propriedades do equipamento são exibidas no *PropertyGrid* situado logo abaixo. O painel de configurações de canais também é atualizado, permitindo a edição das configurações dos diversos equipamentos. A Figura 4.35 demonstra os passos para a configuração dos dois sistemas. Primeiro, o equipamento *BioDAQ* foi selecionado. Os controles dos painéis foram atualizados, exibindo as propriedades específicas do *BioDAQ*. O usuário pode então configurar os diversos parâmetros deste equipamento da mesma maneira que faria na versão anterior. Em se-

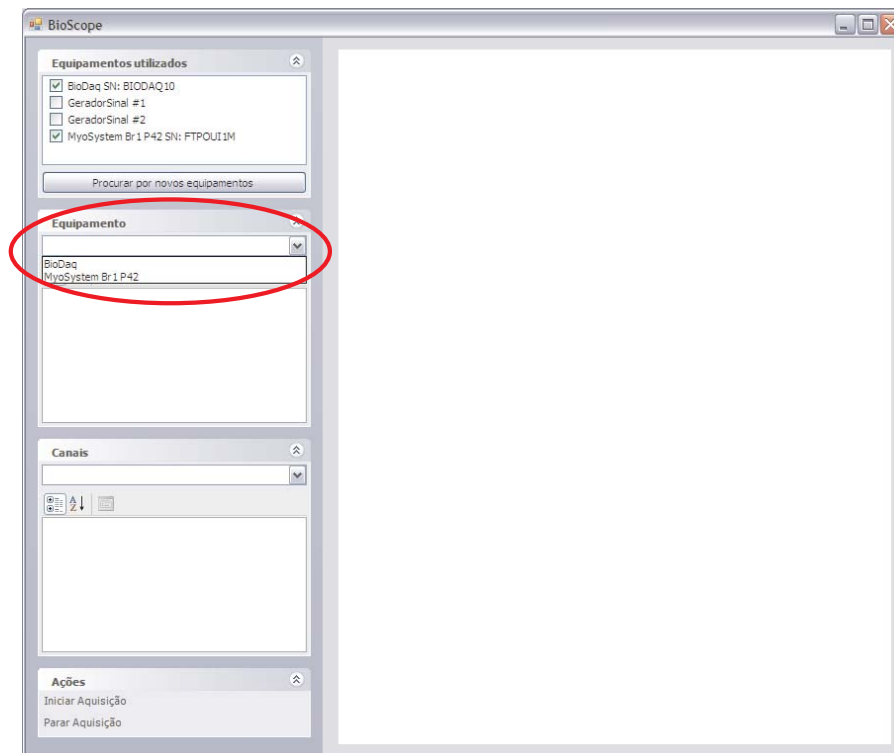


Figura 4.34: Conectando vários equipamentos ao aplicativo *BioScope*.

guida, pode-se selecionar o equipamento *Myosystem-Br1 P42* na lista, e o aplicativo atualizará os painéis com os recursos do equipamento, fornecendo acesso aos seus parâmetros de configuração. O aplicativo permite alternar entre os diversos equipamentos ativados em qualquer ordem, e quantas vezes forem necessárias.

À medida que os canais são ativados, são criados os gráficos na região direita. Note que o título do gráfico exibe não só o nome do canal, mas inclui também o nome do equipamento, para facilitar o reconhecimento. O último painel mantém a sua funcionalidade, permitindo ao usuário controlar o processo de aquisição de dados.

Para demonstrar uma aquisição conjunta, canal 8 do *BioDAQ* e o canal auxiliar 2 do *Myosystem-Br1 P42* foram conectados em paralelo com o gerador de sinal. O *BioDAQ* foi configurado para realizar uma aquisição em 1khz, e seu canal 8 foi ativado no *BioScope*. O *Myosystem-Br1 P42* foi configurado para realizar uma coleta em 2khz e o canal auxiliar 2 foi ativado sendo configurado com ganho unitário. Em seguida, a aquisição foi iniciada. A

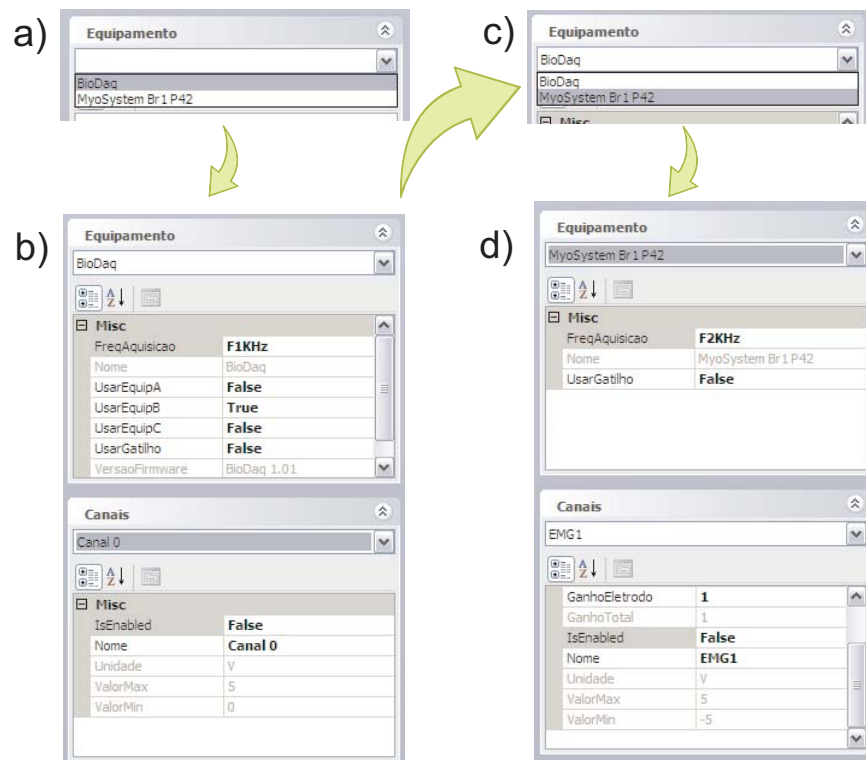


Figura 4.35: Painéis do aplicativo *BioScope* sendo atualizados de acordo com o equipamento selecionado. Selecionando o equipamento *BioDAQ* (a), o painel passa a exibir as propriedades únicas desse equipamento (b). Ao selecionar o equipamento *MyoSystem Br1 P42* (c), o painel se modifica, passando a exibir as propriedades do *MyoSystem Br1 P42* (d).

Figura 4.36 mostra o sinal coletado pelos dois canais fornecidos de ambos os equipamentos. Note que, apesar do sinal aplicado aos dois canais ser o mesmo, eles estão defasados. Isso ocorre porque os equipamentos não foram sincronizados, ou seja, o início da coleta foi feita separadamente para cada equipamento. A escala de cada sinal também é diferente, mas, pelos valores dos eixos dos gráficos, é possível verificar que eles possuem a mesma amplitude e frequência.

Observe que, neste experimento, foram realizadas alterações significativas no aplicativo *BioScope*, mas ainda assim, o sistema continuou funcionando sem a necessidade de modificar os *plug-ins* já existentes. Eles sequer foram recompilados no processo, demons-

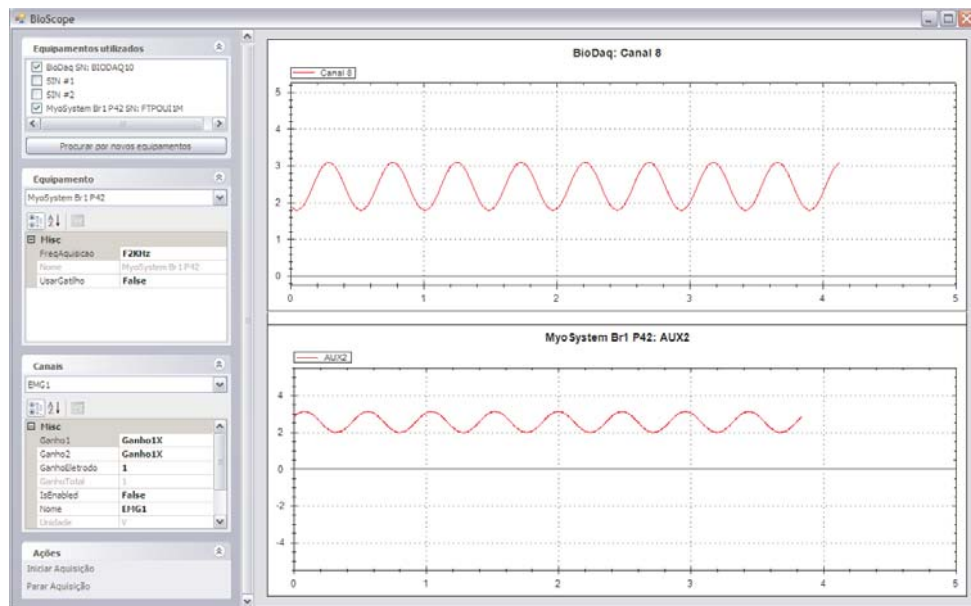


Figura 4.36: Aplicativo *BioScope* coletando sinais de dois equipamentos diferentes.

trando novamente a capacidade do *framework* de isolar o desenvolvimento do *plug-in* da unidade de processamento.

Capítulo 5

Conclusões e trabalhos futuros

Este trabalho apresentou, inicialmente, uma introdução sobre o desenvolvimento de instrumentos biomédicos baseados em computadores, levantando questões como a falta de compatibilidade entre equipamentos diferentes com um determinado software, dificuldades de manutenção e utilização de vários equipamentos diferentes simultaneamente por um único software. Como solução para aqueles problemas, propõe-se a utilização de uma arquitetura, baseada em *plug-ins*, para realizar o interfaceamento entre aplicativos de software e equipamentos para coleta de dados. A estrutura básica da arquitetura foi implementada na forma de um *framework*, sobre o qual podem ser desenvolvidos aplicativos de captura e processamento de sinais ou conectados a equipamentos de coleta de sinais. O *framework* foi desenvolvido de forma a isolar o desenvolvimento do aplicativo do desenvolvimento do equipamento, permitindo maior flexibilidade e menor acoplamento entre aquelas unidades. Tal arquitetura pode diminuir consideravelmente o impacto de alterações em uma unidade no funcionamento da outra. O *framework* também fornece uma interface única para conexão com equipamentos distintos a um único aplicativo de software. A técnica de reflexão, utilizada neste trabalho, permite ao *framework* realizar a conexão entre o aplicativo e o equipamento de maneira dinâmica e com total acesso a seus recursos.

A implementação do *framework* foi realizada seguindo as seguintes etapas:

- Projeto e desenvolvimento do *framework*.

- Projeto e desenvolvimento do software exemplo para visualização de sinais utilizando o *framework*.
- Projeto e desenvolvimento do *plug-in* para o equipamento *BioDAQ*, fornecendo a interface entre este equipamento e o *framework*.
- Projeto e desenvolvimento do *plug-in* para o eletromiográfico (*Myosystem-Br1 P42*), permitindo ao *framework* controlar este equipamento.

Durante o desenvolvimento do *framework*, houve uma preocupação maior com a compatibilidade entre sistemas. Em alguns momentos, houve uma troca entre desempenho ou utilização de memória em favor de uma maior compatibilidade. Por exemplo, os sinais coletados são armazenados em sua forma transformada, utilizando números em ponto flutuante de 32 bits. Isso gera um maior gasto de memória, já que alguns sistemas realizam a coleta utilizando valores inteiros de 16 bits. Entretanto, existe um ganho de compatibilidade, eliminando a necessidade de realizar essa transformação, que é diferente para cada equipamento, no módulo de processamento. Caso esta transformação seja feita no módulo de processamento, este passa a ser dependente do sistema de aquisição de dados, diminuindo a compatibilidade. Apesar de exigir um maior poder de processamento e quantidade de memória, o *framework* foi construído para operar satisfatoriamente com os computadores atuais.

A opção por utilizar classes abstratas na interface com os SADs complica o desenvolvimento dos *plug-ins*. Isso ocorre porque o *plug-in*, ao serializar seus membros, deve preocupar-se com os atributos da classe base. Para contornar isso é possível modificar o *framework* para utilizar interfaces puras, contendo apenas propriedades e métodos.

O desenvolvimento dos *plug-ins*, em algumas situações, exige a utilização de técnicas de interoperabilidade entre plataformas (Por exemplo, o acesso de uma biblioteca desenvolvida em C para realizar a comunicação com o dispositivo USB). Os métodos de realização dessa interoperabilidade são complexos e sua documentação está dispersa o que pode gerar algumas dificuldades no processo.

O sistema desenvolvido foi avaliado através de diversos experimentos que confirmaram

sua operação com aplicativos de software, que utilizaram o framework para várias tarefas, como por exemplo, captura e visualização de sinais em tempo real. O aplicativo foi capaz de configurar e capturar sinais utilizando equipamentos diferentes, acessando todos os seus recursos, bastando para isso a instalação de *plug-ins*. A adaptação ocorreu em tempo de execução, não sendo necessário recompilar, ou reiniciar o aplicativo para que ela ocorresse, comprovando que o sistema é capaz de adaptar-se ao substituir, alterar ou incluir um equipamento de aquisição de dados.

Outros experimentos mostraram que alterações nos equipamentos não são propagadas para o aplicativo e vice-versa. Isso pode ser observado em duas situações: ao se adicionar um novo recurso (gatilho externo) ao hardware *BioDAQ*, alterando-se seu *plug-in*, o software *BioScope* e os demais *plug-ins* não sofreram nenhuma alteração; alterando-se características do software *BioScope*, para suportar vários equipamentos simultaneamente, os *plug-ins* dos equipamentos não foram modificados.

O *framework* foi também utilizado na integração de vários equipamentos em um único software de processamento, através da modificação do aplicativo *BioScope*, fazendo com que este fosse capaz de captar sinais de fontes diferentes. Essa integração ocorreu de maneira bastante simples, devido à interface única com os equipamentos fornecida pelo *framework*.

Dessa forma, pode-se concluir que, os objetivos traçados inicialmente foram cumpridos e que, através da uso de sistemas adaptativos e de *plug-ins*, é possível minimizar os problemas de falta de compatibilidade entre equipamentos, dificuldades de manutenção e integração dos mesmos.

5.1 Trabalhos futuros

Os experimentos realizados com o *framework DAQSystem* demonstraram que ele é capaz de atender às expectativas iniciais. Entretanto, podemos citar alguns trabalhos que poderão ser realizadas a partir deste:

- Integração do *framework* com outras ferramentas e linguagens de desenvolvimento

para aproveitar os recursos destas no desenvolvimento de sistemas de processamento de sinais.

- Avaliar se a interface fornecida pelo *framework* é capaz de acessar outros equipamentos e, caso contrário, modificá-la.
- Implementar interfaces para utilização do *framework* com em sistema de geração de sinais.
- Criação de um sistema de *plug-ins*, semelhante aos já desenvolvidos, para as unidades de processamento. Com isso, seria possível construir uma plataforma, semelhante a plataforma Eclipse, para aquisição e processamento de sinais que seja totalmente personalizável.
- Desenvolvimento um banco de *plug-ins* para os diversos equipamentos existentes. Com isso, os novos aplicativos de processamento baseados no *framework* seriam automaticamente compatíveis com estes equipamentos.

Apêndice A

Tutorial para o desenvolvimento de *plug-ins*

O desenvolvimento de um *plug-in* para o *framework DAQSystem*, consiste basicamente no desenvolvimento de uma biblioteca de classes da plataforma *.net*, que pode ser feito utilizando uma das diversas linguagens daquela plataforma (C#, C++, Visual Basic, Visual J#). Neste tutorial será utilizada a linguagem C# para o desenvolvimento, entretanto os procedimentos são semelhantes para as outras linguagens, existindo apenas pequenas diferenças na sintaxe.

A.1 Configurando o ambiente de desenvolvimento

Para iniciar o desenvolvimento, é preciso ter a versão de desenvolvimento da plataforma *.net* 2.0. A melhor maneira de obtê-la é instalando uma das versões da IDE *Visual Studio* (Microsoft, Redmond, WA). Existem diversas versões da IDE, incluindo a versão gratuita *Express* (MICROSOFT, 2007c) que pode ser obtida através do site da Microsoft. Para este tutorial será utilizada a IDE Visual C# Express, entretanto, procedimentos similares poderão ser realizados utilizando outras versões da IDE.

Como estamos desenvolvendo uma biblioteca de classes, é preciso que um aplicativo a utilize, para testá-la. O aplicativo *BioScope* pode ser utilizado para esta finalidade, dessa

forma, em vez de começar uma solução do zero, será utilizada uma solução contendo o aplicativo *BioScope*. Portanto deve-se abrir a solução *DevPlugin* (incluída no CD que acompanha este documento) contendo o *BioScope*.

Com o ambiente de desenvolvimento configurado, o primeiro passo é a criação de um novo projeto, que deve ser uma biblioteca de classes. Para fazer isso na IDE, deve-se utilizar a ferramenta *Solution Explorer*. Deve-se clicar com o botão direito no primeiro item, aquele que contém o nome da solução, e selecionar a opção *Add*, seguida de *New Project* (Figura A.1). Uma nova janela é apresentada, onde deve-se escolher a opção *Class Library*. Neste exemplo vamos utilizar o nome *GeradorSinal* para o projeto, o que deve ser digitado na caixa de texto e, em seguida, deve-se clicar no botão “OK”. A IDE gera o novo projeto, criando uma nova classe. Salve o projeto e compile para verificar se o ambiente está funcionando.

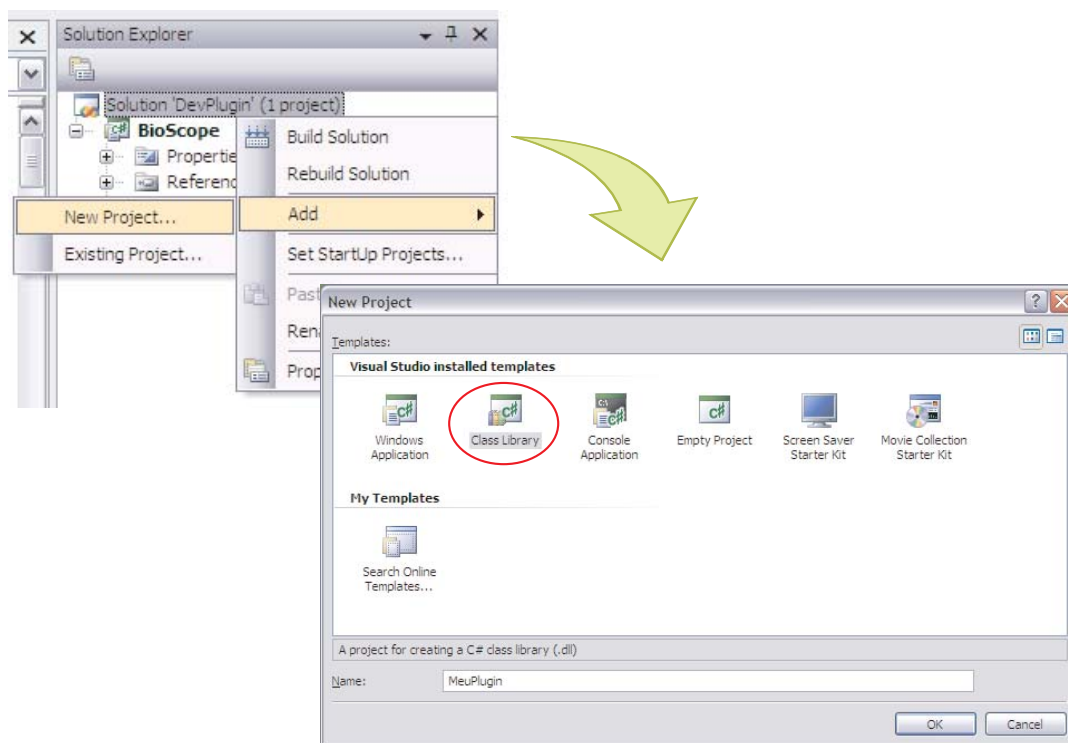


Figura A.1: Criando um novo projeto no *Visual C# Express*.

Como descrito ao longo do texto desta dissertação, o aplicativo *BioScope* procura por

plug-ins no mesmo diretório onde encontra-se seu executável. Dessa forma, é preciso configurar a IDE para enviar os arquivos gerados pelo projeto do *plug-in* para o mesmo diretório do *BioScope*. Para isso, basta clicar com o botão direito no projeto *GeradorSinal* e selecionar a opção *Properties*. As opções do projeto surgirão na tela. Deve-se então, na aba *Build*, modificar a opção *Output Path* para a mesma do aplicativo *BioScope* (Figura A.2). Na opção *Output Path* deve-se colocar o valor “..\BioScope\bin\Debug\”.

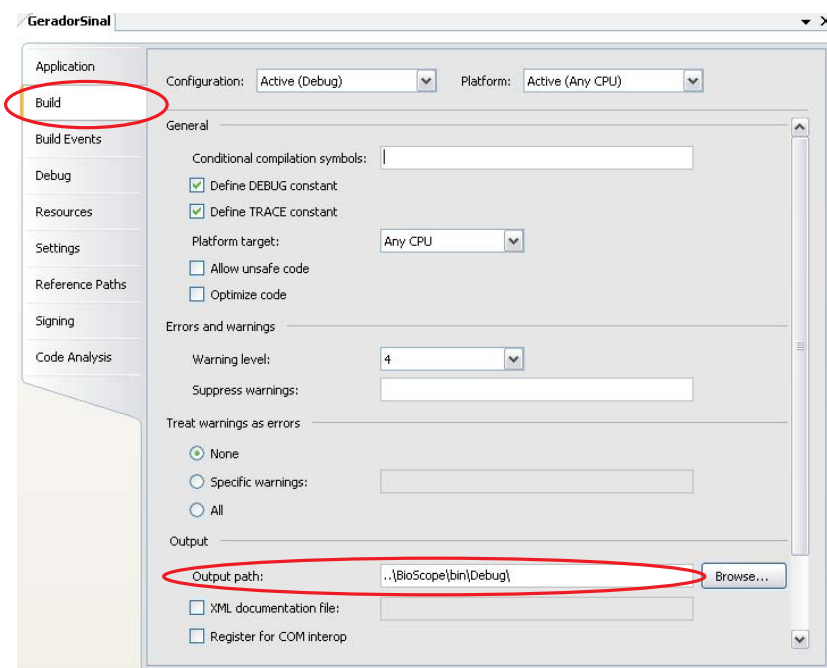


Figura A.2: Janela de configuração do projeto. Em destaque, a opção para onde são enviados os arquivos gerados pelo projeto.

Em seguida, é preciso adicionar a referência ao *framework DAQSystem* e algumas outras bibliotecas auxiliares. Inicialmente, deve-se copiar os arquivos dll do *framework* para o diretório do projeto: *Biolab.DAQSystem.dll* e *Biolab.Collections.dll*. Em seguida, deve-se utilizar a ferramenta *Solution Explorer* para adicionar a referência dessas bibliotecas ao projeto. Para isso, deve-se clicar com o botão direito em *References* e em seguida selecionar na opção *Add Reference* (Figura A.3). Surgirá uma nova janela que permite escolher as referências que serão adicionadas. Deve-se então clicar na aba *Browse* e selecionar os dois arquivos dll pertencentes ao *framework*.

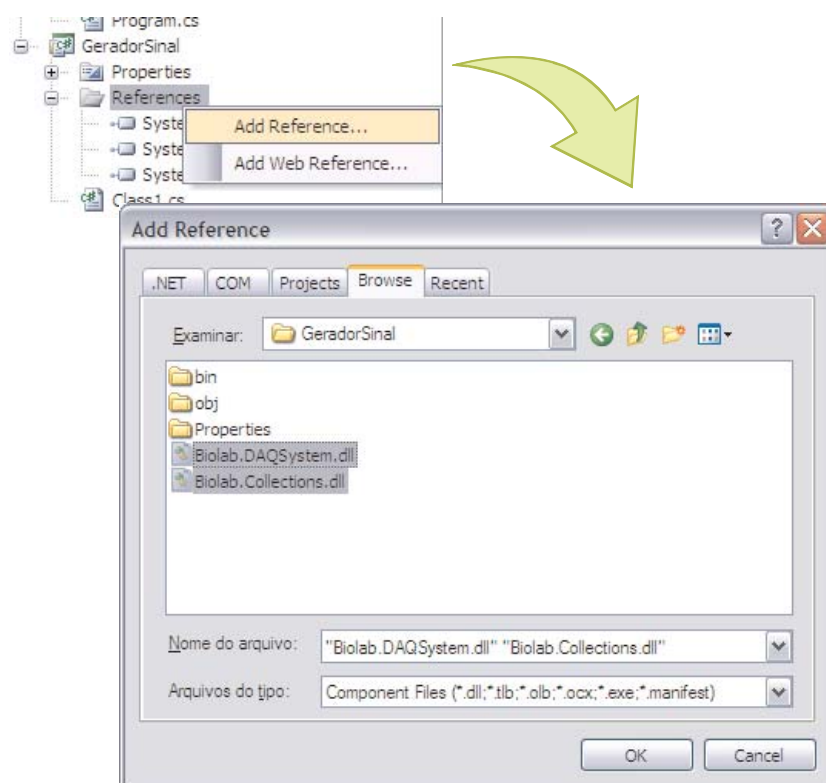


Figura A.3: Passos para adicionar as referências do *framework* ao projeto.

Esses passos concluem a configuração do ambiente de desenvolvimento. A partir deste momento é possível iniciar a criação de um novo *plug-in*.

A.2 Desenvolvendo as classes do *plug-in*

Cada *plug-in* pode fornecer acesso a vários equipamentos. Para cada equipamento que o *plug-in* suporta deve ser criada uma classe derivada de *DAQ*. Como cada equipamento é composto de diversos canais de entrada, também será preciso desenvolver uma classe derivada de *Canal*. Portanto, um *plug-in* que fornece acesso a um único equipamento será composto por pelo menos duas classes: uma classe para modelar o equipamento e outra classe que modela seus canais.

Para ilustrar o desenvolvimento de um *plug-in*, será demonstrado o desenvolvimento do plugin do equipamento virtual *Gerador Sinal*, apresentado no Capítulo 3 e 4. O *Gerador*

Sinal é um equipamento de aquisição de dados virtual, que gera ondas senoidais bem definidas. A Figura A.4 mostra o diagrama de blocos do equipamento.

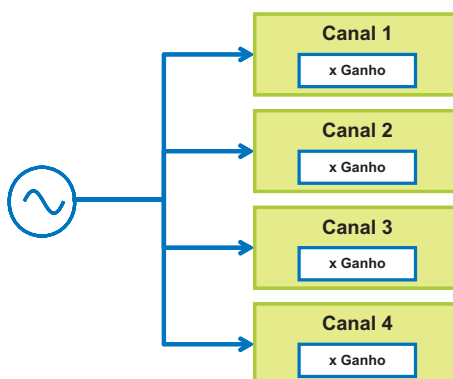


Figura A.4: Diagrama de blocos do equipamento virtual acessado pelo *plug-in Gerador-Sinal*.

O equipamento é composto por um gerador de ondas senoidais conectado a quatro canais. É possível definir qual a frequência da onda senoidal gerada, além da frequência de amostragem que será utilizada. O mesmo sinal gerado chega a todos os canais, entretanto, cada canal possui a capacidade de definir um ganho independente para este sinal.

Após obter as especificações do sistema de aquisição passamos para o desenvolvimento das subclasses *DAQ* e *Canal*, conforme apresentado no Capítulo 3 deste trabalho. Para facilitar o desenvolvimento, foi criado um gabarito básico para essas duas classes que atende aos requisitos necessários. O *plug-in* pode ser então desenvolvido a partir desse gabarito. Os gabaritos geram código a partir de algumas informações fornecidas pelo usuário, eles são arquivos zip que devem ser copiados para o diretório Meus Documentos\Visual Studio 2005\Templates\Item Templates. No CD que acompanha este documento é possível encontrar os gabaritos para as subclasses *DAQ* e *Canal*, após copiar os dois arquivos para o diretório especificado, pode-se facilmente criar as classes necessárias para acesso ao equipamento.

Numa primeira etapa, será criada a classe derivada de *DAQ*, que será utilizada para acessar o equipamento. Para isso, deve-se clicar com o botão direito no item referente ao projeto, na ferramenta *Solution Explorer* (Figura A.5). Um menu de opções surgirá,

onde deverão ser selecionadas as opções *Add* e *New Item*. Uma janela será apresentada, listando uma série de gabaritos que poderão ser utilizados para a criação deste novo item. No final da lista poderão ser encontrados os gabaritos para a criação de objetos *DAQ* e *Canal*. A opção *Novo DAQ* deve ser selecionada e o nome da classe deve ser digitado na caixa de texto logo abaixo. Para este tutorial, o nome da classe será *GeradorSinal*. Ao clicar no botão “OK”, uma nova classe será criada, contendo o esboço de diversas funções e comentários para auxiliar no desenvolvimento.

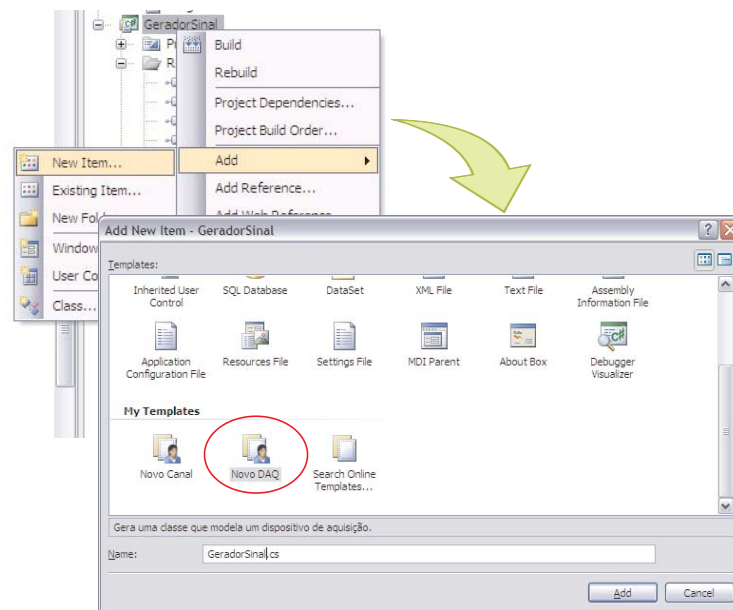


Figura A.5: Passos para criar uma subclasse de *DAQ* utilizando o gabarito fornecido com o *Framework*.

Para criar a subclasse *Canal*, os passos são os mesmos. Apenas o gabarito *Novo Canal* será utilizado no lugar do *Novo DAQ*. Para esta nova classe será dado o nome *GeradorCanal*.

Com essas duas classes, já se obtém a estrutura básica do *plug-in*. Ao observar o código gerado, é possível identificar a implementação desses requisitos. Por exemplo, a Listagem A.1 mostra a definição da classe *GeradorSinal*. Observe que já foi incluída a herança dessa classe com a classe *DAQ* (linha 2) que é o primeiro requisito para a construção do *plug-in*. Outros elementos necessários podem ser observados na mesma listagem. O

atributo *Serializable* foi adicionado à classe (linha 1) e a classe também herda da interface *ISerializable* para permitir salvar ou reaver o seu estado em arquivo.

Listagem A.1: Definição da classe *GeradorSinal*.

```
1 [ Serializable ]
2 public class GeradorSinal : DAQ, ISerializable
3 {
4     //Código da classe ...
5 }
```

Observando o código incluído na classe, percebe-se também uma série de métodos que não possuem uma implementação, apenas geram uma exceção. A construção do *plug-in* fica então reduzida à implementação destes métodos e a criação de novos que o equipamento necessite.

Para iniciar o desenvolvimento, será feito com que o *plug-in* reconheça os equipamentos conectados ao sistema. Segundo os requisitos de criação do *plug-in*, é necessário que a subclasse de *DAQ* contenha um método estático de pesquisa de equipamentos, marcado com o atributo *DAQScanMethod*. O gabarito gerou este método, entretanto não forneceu a sua implementação. O método recebeu o nome de *Scan*, e deve retornar um vetor de *strings*, uma para cada equipamento, onde, cada *string* será utilizada para identificar um dado equipamento. A Listagem A.2 mostra o método gerado pelo gabarito.

Listagem A.2: Método *Scan* gerado pelo gabarito *Novo DAQ*.

```
1 [ DAQScanMethod() ]
2 public static string[] Scan()
3 {
4
5     //TODO: remover a próxima linha após terminar a implementação
6     throw new Exception("The method or operation is not implemented.");
7
8     //TODO: Adicionar código para pesquisar os dispositivos
9     //conectados ao PC
10 }
```

O método *Scan* deve conter rotinas para encontrar equipamentos compatíveis com o *plug-in* conectados ao computador. Por exemplo, um equipamento que utilize comunicação serial, poderia enviar um comando específico do equipamento, com resposta conhecida, para todas as portas seriais disponíveis no computador. Nas portas em que a resposta esperada for recebida, pode-se deduzir que existe um equipamento compatível. Após detectar o equipamento o método deve gerar uma *string* que identifique este equipamento. Esta *string* será passada como parâmetro para o construtor do *plug-in* e pode conter informações que facilitem a conexão com o mesmo. No exemplo da porta serial, a *string* poderia ser formada pelo nome do equipamento e pela identificação da porta serial à qual o equipamento está conectado. Ao varrer todas as portas, um vetor de *strings* deve ser gerado, identificando todos os equipamentos disponíveis no sistema. Por ser um método estático, o método de pesquisa não pode utilizar variáveis membro ou outros métodos da classe e deve liberar todos os recursos utilizados no final da execução. A Listagem A.3 apresenta um exemplo de implementação para a situação descrita anteriormente.

Listagem A.3: Exemplo de implementação do método *Scan*, procurando por equipamentos conectados à portas seriais.

```
1  [DAQScanMethod()]
2  public static string[] Scan()
3  {
4      //Cria uma lista de strings que conterão os equipamentos
5      //encontrados
6      List<string> equipamentoEncontrados = new List<string>();
7
8      //Obtém uma lista de todas as portas seriais disponíveis
9      //no computador
10     string[] portasDoComputador = SerialPort.GetPortNames();
11
12     //Percorre todas as portas
13     foreach (string porta in portasDoComputador)
14     {
15         //Cria um objeto de acesso à porta serial
16         SerialPort serialPort = new SerialPort(porta);
```

```
17      //Ativa a comunicação com a porta serial
18      serialPort.Open();
19
20      //Transmite o comando para o equipamento enviar a
21      //sua versão. A resposta esperada é EQUIPAMENTO 1.00
22      serialPort.Write("Envia Versão");
23
24      //Obtém a resposta da porta serial
25      string resposta = serialPort.ReadExisting();
26
27      //Se a resposta for a esperada, adiciona o equipamento
28      //na lista dos equipamentos encontrados.
29      if (resposta.Equals("EQUIPAMENTO 1.00"))
30      {
31          //A string que identifica o equipamento contém o nome
32          //do equipamento seguido da porta que ele utiliza
33          equipamentoEncontrados.Add("Equipamento P:" + porta);
34      }
35
36      //Libera os recursos da porta serial
37      serialPort.Close();
38      serialPort.Dispose();
39
40  }
41
42      //Retorna o vetor de equipamentos encontrados
43      return equipamentoEncontrados.ToArray();
44
45  }
```

Como se está construindo um *plug-in* para um equipamento virtual, a implementação do método *Scan* é bastante simples. Ele simplesmente retorna um vetor de *strings* contendo um determinado número de equipamentos. A Listagem A.4 apresenta essa implementação, de modo que, ao recompilar o código do *plug-in* e executar o aplicativo *BioScope*, o novo equipamento é encontrado (Figura ??).

Listagem A.4: Código do método *Scan* do equipamento *GeradorSinal*.

```
1 [DAQScanMethod()]
2 public static string[] Scan()
3 {
4     //Sempre retorna dois equipamentos
5     return new string[] { "GeradorSinal #1", "GeradorSinal #2" };
6 }
```

O aplicativo *BioSocpe* já será capaz de detectar a presença do equipamento fornecido pelo *plug-in* em desenvolvimento. Entretanto, se o usuário tentar ativar um desses equipamentos, ele irá gerar um erro no aplicativo. Esse erro ocorre no construtor da classe *GeradorSinal* que recebe uma *string* como parâmetro. Este construtor é uma das exigências para a construção de um *plug-in* para o *framework DAQSystem*. O gabarito utilizado na criação da classe adicionou este construtor, mas sua implementação simplesmente gera uma exceção. A Listagem A.5 apresenta o construtor gerado pelo gabarito.

Listagem A.5: Código do método *Scan* do equipamento *GeradorSinal*.

```
1 [DAQConstructor()]
2 public GeradorSinal(string dispositivo)
3 {
4     //TODO: remover a próxima linha após terminar a implementação
5     throw new Exception("The method or operation is not implemented.");
6
7     //Armazena o nome e a versão do sistema de aquisição
8     _nome = dispositivo;
9     _versao = 1000; //0.1.0
10
11     //TODO: Criar os canais
12     //Ex: _canais = new Canal[2];
13     //for (int i = 0; i < _canais.Length; i++)
14     //{
15         //    _canais[i] = new CanalGeradorSeno();
16         //    _canais[i].Nome = "Canal " + i;
17     //}
18 }
```

```
19      //TODO: Instanciar os outros elementos da classe
20
21  }
```

A primeira linha de código adiciona o atributo *DAQConstructor* ao construtor. Este atributo é utilizado pelo *Framework* para identificar este construtor especial. O construtor recebe como parâmetro uma *string*, que será uma das *strings* que a própria classe gerou no método *Scan*. Portanto, é possível utilizar essa *string* para passar parâmetros para o construtor. Como todo construtor, este tem a função de gerar um estado inicial para os elementos da classe, incluindo alguns elementos herdados da classe *DAQ*.

Os elementos herdados da classe *DAQ* que o construtor deve-se preocupar em gerar um estado inicial são: o nome do equipamento, a versão do equipamento e o vetor que armazena os canais do equipamento. O nome deve ser definido através da variável *_nome* que recebe uma *string*. A versão do equipamento é definida na variável *_versao* que recebe um valor inteiro. Cada três algarismos desse inteiro correspondem a uma parte da versão. Por exemplo: A versão 1.20.32 seria representada pelo inteiro 1.020.032. O terceiro elemento herdado da classe *DAQ* que deve receber um estado inicial é o vetor de canais. Os comentários incluídos no código gerado pelo gabarito já indicam como esse estado inicial é gerado. Primeiro deve-se criar um novo vetor de objetos do tipo *Canal* com dimensão igual ao número de canais do equipamento. Em seguida deve-se associar cada elemento desse vetor a um novo objeto de uma das subclasses de *Canal*. A Listagem A.6 apresenta o código que realiza essas três operações para o *plug-in GeradorSinal*.

Listagem A.6: Definindo os valores iniciais para os membros herdados da classe *DAQ* no construtor da classe *GeradorSinal*.

```
1  [DAQConstructor()]
2  public GeradorSinal(string dispositivo)
3  {
4      //Armazena o nome e a versão do sistema de aquisição
5      _nome = dispositivo;
6      _versao = 1000; //0.1.0
7  }
```

```
8      //Como o equipamento possui quatro canais, a dimensão
9      //do vetor é 4
10     _canais = new Canal[4];
11     for (int i = 0; i < _canais.Length; i++)
12     {
13         //Associando cada elemento do vetor com uma das
14         //subclasses de canal
15         _canais[i] = new GeradorCanal();
16         //Definindo o nome do canal
17         _canais[i].Nome = "Canal " + i;
18     }
19
20     //TODO: Instanciar os outros elementos da classe
21
22 }
```

Após gerar o estado inicial dos elementos herdados da classe *DAQ*, o construtor deve definir o valor inicial dos elementos presentes na própria classe. Um destes elementos, provavelmente, será um objeto que é utilizado para a comunicação com o equipamento. Voltando ao exemplo de um equipamento que utiliza a porta serial, um dos membros da classe seria um objeto do tipo *SerialPort*. O construtor deve gerar uma instância desse objeto e, em seguida, abrir o canal de comunicação com o equipamento. Com a comunicação com o equipamento estabelecida deve-se enviar uma série de comandos para este equipamento, definindo o seu estado inicial. Durante o desenvolvimento do *plug-in*, deve-se ter o cuidado de sempre manter o estado do equipamento em sincronia com o estado do objeto que o modela.

Mesmo implementando este construtor para a classe *GeradorSinal*, o aplicativo *BioScope* não é capaz de ativar o equipamento. Isso ocorre por que ainda não temos alguns elementos implementados na classe *GeradorCanal*. Estes elementos são, o construtor padrão e o método *Clone*.

O construtor padrão da classe *GeradorCanal*, criado pelo gabarito, inclui um código para gerar uma exceção (Linha 4 da Listagem A.7). Este erro foi incluído apenas para

lembrar o usuário do gabarito de gerar uma implementação para este construtor. Se o código for removido, já é possível gerar uma instância desse objeto. Continuando o desenvolvimento do *plug-in*, este construtor deve gerar o valor inicial dos elementos da classe. Esse código será adicionado posteriormente.

Listagem A.7: Código do construtor padrão da classe *GeradorCanal* criado pelo gabarito.

```
1 public GeradorCanal()  
2 {  
3     //TODO: remover a próxima linha após terminar a implementação  
4     throw new Exception("The method or operation is not implemented.");  
5  
6 }
```

O método *Clone* é utilizado para gerar uma cópia do objeto, ou seja, uma nova instância do objeto, contendo os mesmos valores do objeto original. Como pode ser observado na Listagem A.8, o gabarito criou alguns comentários exemplificando como deve ser feita a implementação deste método.

Listagem A.8: Código do método *Clone* gerado pelo gabarito *Novo Canal*.

```
1 public override object Clone()  
2 {  
3     //TODO: remover a próxima linha após terminar a implementação  
4     throw new Exception("The method or operation is not implemented.");  
5  
6     //Cria um novo objeto  
7     GeradorCanal canal = new GeradorCanal();  
8     //Copia os membros da classe base  
9     CloneCanalMembers(canal);  
10  
11     //TODO: Copiar os membros desta classe  
12     //Ex: canal.Nome = this.Nome;  
13  
14     return canal;  
15 }
```

Para implementar esse método, primeiro é preciso remover a exceção gerada pelo gabarito na linha quatro. O método clone deve criar um novo objeto da mesma classe, e copiar todos os campos deste objeto para o novo, incluindo aqueles que foram herdados da classe base *Canal*. Uma parte desse processo já foi criado pelo gabarito. Na linha sete, o novo objeto é criado. Na linha nove, a função auxiliar *CloneCanalMembers*, herdada da classe *Canal*, copia todos os membros da classe canal para o novo objeto. Os demais elementos da classe devem ser copiados conforme mostra o exemplo comentado. Ao final do processo, o método retorna o objeto recém criado.

Como não foram adicionados campos à classe *GeradorCanal*, basta remover a exceção para que o método possa ser executado. À medida que novos campos forem adicionados à classe, o método *Clone* deverá ser modificado para realizar a cópia destes. Neste momento já é possível ativar o equipamento referente ao *plug-in* no aplicativo *BioScope*, facilitando a realização de testes com as classes em desenvolvimento. Ao executar o projeto, a IDE já copia a *Dll* do *plug-in* para o diretório do executável do *BioScope* e executa-o. A Figura A.6 mostra os painéis de configuração exibindo as propriedades do novo *Plug-in*.

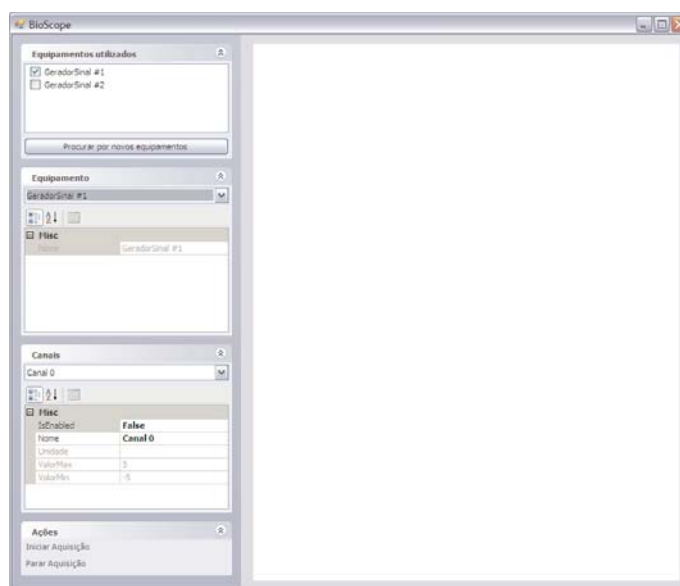


Figura A.6: Aplicativo *BioScope* exibindo as propriedades do equipamento *GeradorSinal*, ainda incompleto.

Com o *plug-in* sendo capaz de ser ativado, pois este já apresenta os requisitos básicos do *framework*, pode-se passar para a implementação dos recursos de configuração do equipamento. O *framework DAQSystem* exige que esses recursos sejam implementados na forma de propriedades. Um dos recursos do gerador de sinal é poder definir qual a frequência de aquisição. Para acrescentar este recurso no *plug-in*, primeiro será criado um campo denominado *_freqAquisicao* para armazenar esse valor. Em seguida deve-se criar uma nova propriedade, que será utilizada para obter o valor atual do membro ou alterá-lo. O código da Listagem A.9 demonstra as duas etapas descritas anteriormente.

Listagem A.9: Código da propriedade *FreqAquisicao*, adicionado à classe *GeradorSinal*, para fornecer acesso aos recursos do equipamento.

```
1  //Campo que irá armazenar a frequência de aquisição
2  private float _freqAquisicao = 100;
3
4  //Propriedade utilizada para alterar a frequência de
5  //aquisição
6  public float FreqAquisicao
7  {
8      //Obtendo o valor da propriedade
9      get
10     {
11         return _freqAquisicao;
12     }
13     //Código para definir o novo valor
14     set
15     {
16         //Só altera o valor se a frequência for
17         //maior do que zero, mantendo a consistência
18         //da classe
19         if(value > 0)
20             _freqAquisicao = value;
21     }
22 }
```

Na segunda linha do código, é criado o campo *_freqAquisicao*, que irá armazenar o valor da frequência de aquisição utilizado pelo equipamento. As linhas 6 até 22 definem a propriedade *FreqAquisicao*. Essa propriedade define dois métodos: o método *Get* que retorna o valor da propriedade, e o método *Set* que altera o valor da propriedade. O método *Get* simplesmente retorna o valor contido no campo *_freqAquisicao*. Já o método *Set*, utilizado para alterar o valor da variável, deve garantir a consistência do objeto. Por isso, foi adicionado um teste, verificando se o valor passado é maior do que zero. Em um equipamento real, este método também deverá enviar comandos para o equipamento, configurando o novo valor no *hardware*.

Caso o recurso do equipamento só possa assumir alguns valores específicos, é interessante criar uma enumeração e utilizá-la como tipo de dado no campo e na propriedade. Isso facilitaria a edição destes valores, já que o editor de propriedades é capaz de identificar os itens que fazem parte dessa enumeração, e listá-los em uma caixa de combinação. Dessa forma, a edição da propriedade passa ocorrer da mesma forma como se seleciona um tipo de fonte em um editor de texto.

Além da propriedade que controla a frequência de aquisição, foi adicionada uma propriedade que define a frequência da senóide gerada. A codificação dessa propriedade segue as mesmas regras e pode ser encontrada no CD que acompanha este documento.

Para adicionar acesso à configuração de um recurso do canal, o processo é análogo. Cria-se um campo que irá armazenar o valor da configuração e uma propriedade para fornecer acesso a este campo. Entretanto, para os canais, existem alguns detalhes extras que devem ser observados. Estes detalhes serão apresentados um pouco mais à frente no texto.

No exemplo do *GeradorSinal*, um de seus requisitos especificam que cada canal pode possuir um ganho diferente. Este requisito deve ser implementado com um recurso de configuração do canal. A Listagem A.10 apresenta um exemplo dessa implementação. Nela, é criado o campo *_ganho* na classe *GeradorCanal*, e a propriedade *Ganho* associada a este campo.

Listagem A.10: Código da propriedade *Ganho* incluído na classe *GeradorCanal*.

```
1 //Campo que irá armazenar o ganho do canal
2 private float _ganho = 1.0f;
3
4 //Propriedade para configurar o ganho
5 public float Ganho
6 {
7     get
8     {
9         return _ganho;
10    }
11    set
12    {
13        if (value > 0)
14            _ganho = value;
15    }
16 }
```

Na implementação do método *Set*, referente à propriedade *Ganho*, novamente, foi incluído um código para garantir a consistência do objeto. Neste ponto, acabam as semelhanças de desenvolvimento de um recurso presente no equipamento para um canal. A primeira diferença, é que o objeto canal não possui acesso direto ao canal digital de comunicação com o equipamento. Dessa forma, este objeto não irá alterar o valor da propriedade em *hardware*, essa operação será feita posteriormente na classe *GeradorSinal*, através do método *AtualizaCanais*. Um segundo detalhe, é adicionar o código que realiza a cópia do novo membro no método *clone*. A Listagem A.11 apresenta o código do método *Clone* após adicionar a cópia do membro *_ganho* (Linha 11).

Listagem A.11: Código do método *Clone*, com o código para a cópia do campo *_ganho* na linha 11.

```
1 public override object Clone()
2 {
3     //Cria um novo objeto
4     GeradorCanal canal = new GeradorCanal();
```

```
5      //Copia os membros da classe base
6      CloneCanalMembers( canal );
7
8      //TODO: Copiar os membros desta classe
9      //Ex: canal.Nome = this.Nome;
10     //Copiando o campo _ganho
11     canal._ganho = this._ganho;
12
13     return canal;
14 }
```

Após realizar as operações descritas anteriormente na classe *GeradorCanal*, é preciso retornar para a classe *GeradorSinal* que modela o equipamento para que esta possa enviar as novas configurações do canal para o equipamento. Isto é feito através da sobrecarga do método *AtualizaCanais*. O gabarito já gerou esta sobrecarga (Listagem A.12), entretanto é necessário fornecer a sua implementação.

Listagem A.12: Código do método *AtualizaCanais* da classe *GeradorSinal* gerado pelo gabarito.

```
1 protected override void AtualizaCanais()
2 {
3     //TODO: remover a próxima linha após terminar a implementação
4     throw new Exception("The method or operation is not implemented.");
5
6     //TODO: Gerar código para atualizar os canais.
7 }
```

Como o exemplo apresentado não comunica-se com nenhum equipamento físico, basta remover a exceção gerada pelo gabarito para que já seja possível acessar a propriedade *Ganho* através do aplicativo *BioScope*. Após recompilar o projeto e executar o aplicativo *BioScope*, ativando um dos equipamentos virtuais, obtém-se a os painéis apresentados na Figura A.7.

Em um equipamento real o método *AtualizaCanais* deve enviar comandos para realizar a configuração do equipamento de acordo com as novas configurações do canal. A Listagem

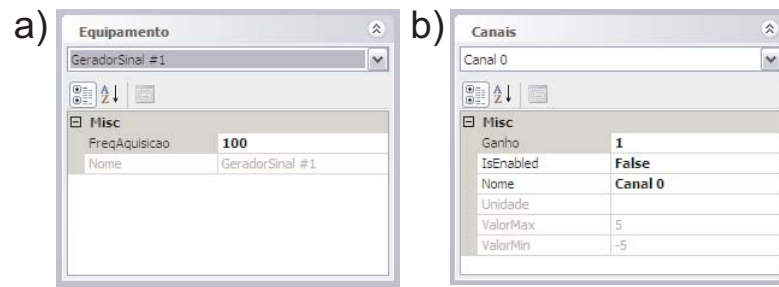


Figura A.7: Aplicativo *BioScope* exibindo as propriedades do equipamento *GeradorSinal*. O painel exibe: (a) A propriedade *FreqAquisicao*, adicionada à classe *GeradorSinal*. (b) A propriedade *Ganho*, adicionada à classe *GeradorCanal*.

A.13 exemplifica como isso poderia ser feito, se o *GeradorSinal* utilizasse uma das portas seriais do computador para comunicar-se com o equipamento. O método percorre todos os canais através de um laço, e, para cada canal, envia o comando de configuração do ganho. A linha 9 faz uma conversão de tipo, isso se faz necessário, pois o vetor *canais* armazena objetos do tipo *Canal*.

Listagem A.13: Código do método *AtualizaCanais* da classe *GeradorSinal* gerado pelo gabarito.

```

1  protected override void AtualizaCanais()
2  {
3      //Varre os canais do objeto
4      for(int i = 0; i < _canais.Length; i++)
5      {
6          //Para cada canal envia as configurações
7
8          //Realiza uma conversão de tipo (TypeCast)
9          GeradorCanal canalDoGerador = (GeradorCanal) _canais[i];
10         //Transmite o novo ganho para o equipamento
11         _portaSerial.Write("Configurar Ganho: Canal " + i.ToString()
12             + " Ganho " + canalDoGerador.Ganho.ToString());
13     }
14 }
```

A.3 Iniciando e parando o processo de aquisição

Com o código construído até este momento, já é possível ativar o equipamento e configurar seus parâmetros de aquisição. Nesta próxima etapa, será codificado o processo de aquisição. O *framework DAQSystem* exige que a classe que modela o equipamento implemente dois métodos relacionados a este processo: O método *IniciarAquisicao* e *PararAquisicao*.

O método *IniciarAquisicao* deve criar uma nova tarefa concorrente que irá, continuamente, obter amostras do equipamento, convertê-las para a sua unidade final e adicioná-las aos *buffers* dos canais. O método *PararAquisicao* deve, basicamente, terminar a tarefa gerada pelo método *IniciarAquisicao*, encerrando o processo.

O gabarito já gerou uma implementação parcial dos dois métodos, com alguns comentários. A Listagem A.14 apresenta o código gerado pelo gabarito para o método *IniciarAquisicao*. A linha 13 contém apenas uma exceção gerada para lembrar o usuário de realizar a implementação do método, sendo que esta deve ser removida. As linhas 16 a 19 limpam os *buffers* dos canais, eliminando qualquer amostra que o aplicativo não tenha coletado. Na linha 25, o gabarito cria uma nova tarefa concorrente, associando-a a variável membro da classe denominada *_aqThread*. A tarefa a ser executada deve ser implementada no método *ThreadAquisicao*, que também foi gerado pelo gabarito. Por fim, na linha 26, a nova tarefa concorrente é colocada em modo de execução, passando a disputar o processador com as outras tarefas do sistema.

Listagem A.14: Código do método *IniciarAquisicao* da classe *GeradorSinal* gerado pelo gabarito.

```
1 //Variável membro utilizada para controlar a tarefa
2 //concorrenete
3 private Thread _aqThread;
4
5 //Código da classe...
6
7 /// <summary>
8 /// Inicia a aquisição de dados
```

```

9  /// </summary>
10 public override void IniciarAquisicao()
11 {
12     //TODO: remover a próxima linha após terminar a implementação
13     throw new Exception("The method or operation is not implemented.");
14
15     //Limpa os buffers
16     foreach (Canal c in _canaIs)
17     {
18         c.Buffer.Limpa();
19     }
20
21     //TODO: Gerar código para preparar o sistema de aquisição
22
23
24     //Dispara a thread de aquisição de dados
25     _aqThread = new Thread(new ThreadStart(this.ThreadAquisicao));
26     _aqThread.Start();
27 }

```

Em um sistema real, antes de iniciar a tarefa de aquisição, o método *IniciarAquisicao* deve enviar comandos para o equipamento, ativando o processo de digitalização. Também é de responsabilidade do *plug-in* informar o tempo de cada amostra. Para fazer isso, é possível adicionar uma nova variável membro (*_tempoAtual*) à classe, e, no código do método *IniciarAquisicao*, definir seu valor inicial como zero. O código Listagem A.15 exemplifica esse processo, onde a comunicação com o equipamento é feita através da porta serial.

Listagem A.15: Exemplo de implementação do código do método *IniciarAquisicao* utilizando a comunicação serial.

```

1 public override void IniciarAquisicao()
2 {
3     //Limpa os buffers
4     foreach (Canal c in _canaIs)

```

```
5      {
6          c . Buffer . Limpa ( ) ;
7      }
8
9      //Reseta uma variável membro utilizada
10     //para controlar o tempo
11     _tempoAtual = 0.0f;
12
13     //Envia o comando para iniciar a aquisição
14     //antes de iniciar a nova tarefa concorrente
15     _portaSerial . Write ( "Inicia a Aquisicao" );
16
17     //Dispara a thread de aquisição de dados
18     _aqThread = new Thread ( new ThreadStart ( this . ThreadAquisicao ) );
19     _aqThread . Start ( ) ;
20 }
```

Neste momento, é preciso criar o código da tarefa concorrente que foi iniciada no método *IniciarAquisicao*. Esta tarefa deverá, constantemente, receber amostras do equipamento, convertê-las para a sua unidade final, e adicioná-las aos *buffers* de cada canal. Para fazer isso, iremos criar um laço, controlado por uma variável membro da classe chamada *_estaAquisicionando*. É muito comum, ao criar tarefas concorrentes, utilizar um laço infinito. Entretanto, a opção por utilizar uma variável de controle é mais interessante, pois é possível terminar a tarefa concorrente simplesmente terminando a execução do método. Isso evita uma parada abrupta do código, se fosse utilizado, por exemplo, o método *Abort* no objeto *_aqThread*, forçando o término da tarefa. A Listagem A.16 demonstra o laço criado para realizar a tarefa concorrente. Este laço foi incluído no método *ThreadAquisicao*, criado pelo gabarito. A Listagem A.16 também mostra a variável de controle sendo definida como verdadeira no método *IniciarAquisicao*.

Listagem A.16: Laço criado no método *ThreadAquisicao* para executar o processo de aquisição.

```
1  //Variável de controle da tarefa concorrente
2  private bool _estaAquisicionando = false;
3
4  private void ThreadAquisicao() {
5      //Laço que realiza o processo de aquisição
6      while (_estaAquisicionando)
7      {
8          //Lê as amostras do equipamento
9
10         //Converte para a unidade final
11
12         //Envia as amostras para o buffer do canal
13     }
14 }
15
16 public override void IniciarAquisicao()
17 {
18     //Limpa os buffers
19     foreach (Canal c in _canaIs)
20     {
21         c.Buffer.Limpa();
22     }
23
24     //Reseta o tempo
25     _tempoAtual = 0.0f;
26
27     //Seta a variável de controle
28     _estaAquisicionando = true;
29
30     //Dispara a thread de aquisição de dados
31     _aqThread = new Thread(new ThreadStart(this.ThreadAquisicao));
32     _aqThread.Start();
33 }
```

A Listagem A.17 apresenta a implementação do método *ThreadAquisicao* para o *plug-in GeradorSinal*, seguindo os passos descritos anteriormente. O código começa definindo o valor inicial para a variável membro da classe *_tickAnterior*. Esta variável será utilizada para sincronizar o tempo de aquisição com o relógio interno do computador. Isso é feito utilizando a propriedade *TickCount* da classe *Environment*. Esta propriedade retorna um inteiro com o número de milisegundos passados desde que o computador foi ligado.

Em seguida, o método cria um vetor para armazenar os ganhos de cada canal (linhas 7 a 11). Os ganhos são obtidos através do objeto *_canais*, sendo necessário realizar uma mudança de tipo para a classe *GeradorCanal* (linha 10). A opção de armazenar os ganhos no início do método foi feita para melhorar o desempenho, já que uma operação de mudança de tipo é um pouco dispendiosa em termos de processamento para ser colocada dentro de um laço.

Na linha 13, é iniciado o laço da tarefa concorrente. Dentro do laço, novamente, utiliza-se a propriedade *TickCount* para descobrir o tempo decorrido desde a interação anterior, que na primeira iteração corresponde ao início do processo de aquisição. O valor calculado é armazenado na variável *intervalo*. A partir do intervalo e da frequência de aquisição, é calculado o número de amostras que serão geradas pelo sistema (linha 21). Estes passos são necessários apenas porque estamos criando um sistema virtual, sendo preciso criar um mecanismo de controle de tempo. Em um sistema real, bastaria utilizar o canal de comunicação para obter as amostras disponíveis no equipamento, já que este controle de tempo estaria embutido no equipamento.

Em seguida o *plug-in* gera as amostras e as envia para os *buffers* dos canais. Cada amostra é gerada na linha 26, utilizando as rotinas matemáticas da plataforma *.net*. Em seguida um laço percorre os diversos canais, adicionando a nova amostra aos *buffers* dos canais. Para isso, é utilizado o método *Adiciona* da classe *BufferCircular*. Este método possui duas sobrecargas: uma recebe um único valor e a outra recebe um vetor. No caso, foi utilizada a versão que recebe um único valor. O *buffer* do canal é composto de objetos do tipo *Sinal*. Este objeto é composto por um par, amplitude e tempo. Um dos construtores da classe *Sinal* recebe dois valores em ponto flutuante, para gerar o

valor inicial do par. O primeiro valor é a amplitude do sinal, e o segundo o tempo. Este construtor foi utilizado na criação do objeto *Sinal* do exemplo, onde é passado como amplitude o valor gerado anteriormente, multiplicado pelo ganho do canal e, para o tempo, é utilizado o valor da variável *_tempoAtual*.

O código para adicionar os valores ao *buffer* do canal foi incluído dentro de um bloco de tratamento de exceção. Neste caso, a exceção que pode ocorrer é tentar adicionar uma amostra ao *buffer* sendo que este não possui espaço disponível. Neste caso, a exceção é capturada, gerando uma nova exceção para o *Framework* (linha 41).

Após gerar as amostras de todos os canais habilitados, a variável *_tempoAtual*, que controla o tempo de aquisição, é incrementada (linha 48) e o laço continua gerando as demais amostras. Após gerar o número de amostras calculado no início do laço principal (linha 21), a variável *_tickAnterior* é atualizada de acordo com o número de amostras geradas. Por fim, a tarefa é suspensa por 50 ms liberando o processador para que outras tarefas, como, por exemplo, o processamento do sinal, possam ter mais tempo para executar.

Listagem A.17: Implementação do método *ThreadAquisicao* no *plug-in GeradorSinal*.

```
1 private void ThreadAquisicao()
2 {
3     //Obtém o tempo atual em milisegundos
4     _tickAnterior = Environment.TickCount;
5
6     //Gera um vetor com os valores dos ganhos
7     float [] ganhos = new float [_canais.Length];
8     for(int i = 0; i < _canais.Length; i++)
9     {
10         ganhos[i] = ((GeradorCanal)_canais[i]).Ganho;
11     }
12
13     while (_estaAquisicionando)
14     {
15         int novoTick = Environment.TickCount;
16         //Obtém o intervalo desde a última iteração do laço em ms
```

```
17         int intervalo = novoTick - _tickAnterior;
18
19         //Calcula o número de amostras necessárias para a próxima
20         //iteração
21         int nroAmostras = (int)(((float)intervalo) * _freqAquisicao /
22                               1000.0f);
23
24         for (int amostra = 0; amostra < nroAmostras; amostra++)
25         {
26             //Gera a amostra
27             float amplitude = (float)Math.Sin(2 * Math.PI * _freqSenoide *
28                                     _tempoAtual);
29
30             //Envia a amostra para os buffers dos canais
31             for(int nroCanal = 0; nroCanal < _canais.Length; nroCanal++)
32             {
33                 //Só envia amostra para os canais habilitados
34                 if( _canais[nroCanal].IsEnabled)
35                 {
36                     try
37                     {
38                         float ganho = ganhos[nroCanal];
39                         _canais[nroCanal].Buffer.Adiciona(new Sinal(
40                             amplitude * ganho, _tempoAtual));
41                     }
42                     catch (Biolab.Collections.BufferFullException)
43                     {
44                         throw new DAQException("Overflow do Buffer do canal
45                             " + nroCanal);
46                     }
47                 }
48             }
49
50             //Incrementa o tempo
```

```
48         _tempoAtual += (1.0f / _freqAquisicao);
49     }
50
51     //Recalcula o valor do tick de acordo com o número de
52     //amostras geradas
53     _tickAnterior += (int)((float)nroAmostras / _freqAquisicao * 1000.0
54         f);
55
56     //”Dorme” por um tempo, liberando o processador
57     Thread.Sleep(50);
58 }
59 }
```

Ao executar o aplicativo *BioScope* neste momento, já é possível iniciar a aquisição de dados e observar o sinal gerado. Entretanto, o *plug-in* ainda não é capaz parar o processo de aquisição. Para fazer isso, é preciso gerar uma implementação para o método *ParaAquisicao*. Este método deve enviar comandos para o equipamento, parando o processo, além de terminar a tarefa concorrente criada pelo método *IniciarAquisicao*. A Listagem A.18 apresenta a implementação deste método para o *plug-in GeradorSinal*. Ele simplesmente coloca o valor da variável *_estaAquisicionando* em falso. Como a tarefa concorrente, incluída no método *ThreadAquisicao*, encontra-se em um laço controlado por esta variável, ao modificar o valor dela, o fluxo do programa sai do laço encerrando a sua execução. Com isso já é possível iniciar e parar o processo de aquisição, sendo possível observar os gráficos gerados no aplicativo *GeradorSinal* (Figura A.8).

Listagem A.18: Código do método *PararAquisicao*

```
1 public override void PararAquisicao()
2 {
3
4     _estaAquisicionando = false;
5
6 }
```

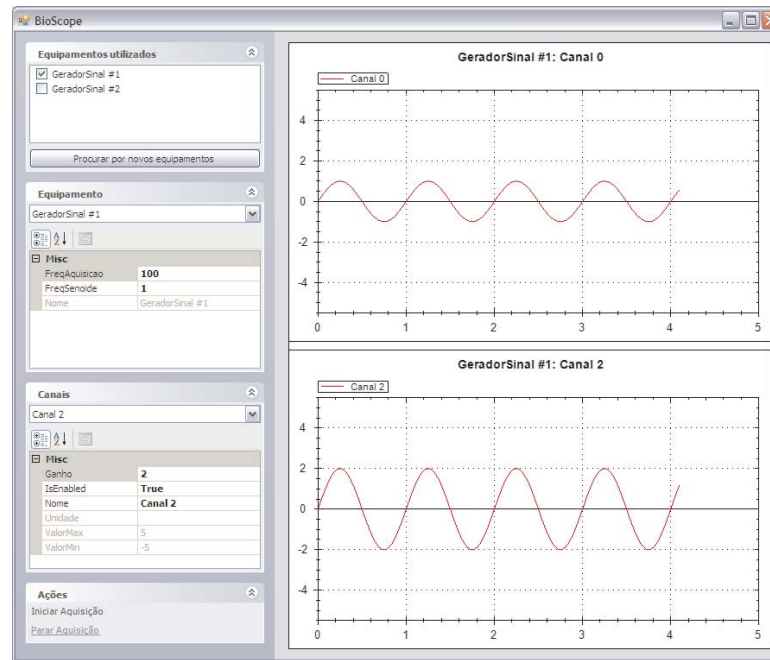


Figura A.8: Aplicativo *BioScope* exibindo os sinais coletados através do *plug-in GeradorSinal*.

A.4 Armazenando o estado do equipamento

Apesar de funcional, o *plug-in* ainda não está completo. O *plug-in* ainda deve ser capaz de salvar o seu estado em disco, para que os aplicativos que o utilizem possam armazenar o estado do equipamento. Isso pode ser útil para repetir experimentos, onde as configurações do equipamento são armazenadas para uma utilização futura. O *plug-in* deve então implementar um mecanismo de serialização. O gabarito já forneceu a estrutura básica para realizar a serialização, entretanto, cabe ao desenvolvedor do *plug-in*, concluir essa implementação.

A serialização é realizada através da implementação da interface *ISerializable*. Esta interface exige a implementação do método *GetObjectData* e de um construtor especial. O método *GetObjectData* realiza a operação de serializar o objeto, ou seja, transformá-lo em um vetor de bytes. Já o construtor especial realiza a operação inversa, gerando uma nova instância do objeto a partir do vetor de bytes. Tanto o método *GetObjectData*,

quanto o construtor especial recebem dois argumentos para realizar essa função: um objeto *SerializationInfo*, através do qual são enviados os campos que serão serializados; e um objeto *StreamingContext* que contém informações sobre para qual tipo de dispositivo o objeto está sendo enviado.

A Listagem A.19 apresenta o código do método *GetObjectData* da classe *GeradorSinal* gerado pelo gabarito. Novamente, temos uma exceção incluída para lembrar ao desenvolvedor de implementar este método (linha 5). Em seguida, são apresentados alguns comentários que indicam como deve ser feita essa implementação. Basicamente, se utiliza o método *AddValue* do objeto *info* para adicionar cada campo do objeto ao vetor de bytes que será criado pelo método.

Listagem A.19: Código do método *GetObjectData* da classe *GeradorSinal* gerado pelo gabarito.

```
1 public void GetObjectData(SerializationInfo info, StreamingContext context)
2 {
3
4     //TODO: remover a próxima linha após terminar a implementação
5     throw new Exception("The method or operation is not implemented.");
6
7     //TODO: Salvar os canais
8     //Exemplo
9     //info.AddValue("NroCanais", Canal.Length);
10    //for(int i = 0; i < _canais.Length; i++)
11    //{
12        //    info.AddValue("Canal" + i, _canais[i]);
13    //}
14
15    //Salvar os membros
16    //Exemplo:
17    //info.AddValue("FrequenciaAquisicao", _freqAquisicao);
18
19 }
```

A Listagem A.20 apresenta a implementação do método para o *plug-in GeradorSinal*.

Inicialmente é gravado o estado dos diversos canais do equipamento. A linha 4 envia inicialmente o número de canais presentes no equipamento associado à chave “*NroCanais*”. Em seguida um laço percorre todos os canais, realizando a serialização destes. Em seguida os demais membros da classe são transmitidos, sendo cada um deles associados a uma chave única.

Listagem A.20: Implementação do método *GetObjectData* da classe *GeradorSinal*.

```
1 public void GetObjectData(SerializationInfo info, StreamingContext context)
2 {
3     //Enviando os canais
4     info.AddValue("NroCanais", Canal.Length);
5     for(int i = 0; i < _canais.Length; i++)
6     {
7         info.AddValue("Canal" + i, _canais[i]);
8     }
9
10    //Enviando os demais membros
11    info.AddValue("FrequenciaAquisicao", _freqAquisicao);
12    info.AddValue("FrequenciaSenoide", _freqSenoide);
13    info.AddValue("Nome", _nome);
14    info.AddValue("Versao", _versao);
15
16 }
```

Observe que nem todos os campos da classe *GeradorSinal* foram utilizados na serialização. Campos como *_tempoAtual* e *_aqThread* não foram serializados, pois não representam necessariamente o estado do objeto, sendo estes utilizados apenas no controle do processo de aquisição. Geralmente serão enviados apenas os campos referentes à configuração do equipamento, os quais estão associados às propriedades.

O construtor especial recebe os mesmos argumentos do método *GetObjectData*. Este construtor também foi gerado pelo gabarito, sendo necessário fornecer a sua implementação. A Listagem A.21 mostra o construtor gerado pelo gabarito. Os comentários incluídos já indicam como deve ser feita a implementação deste construtor (linha 7). O método *Get-*

Value do objeto *info* é utilizado para obter os valores dos campos. Este método recebe dois argumentos: o primeiro é a chave do campo, sendo esta a mesma que foi passada no método *GetObjectData*; o segundo argumento é o tipo do objeto, sendo utilizado um objeto da classe *Type*. No capítulo 2 mostrou-se a utilização da palavra chave *typeof* para gerar um objeto da classe *Type*, referente a um determinado tipo de dado. Essa palavra chave é utilizada no código para fornecer o segundo argumento do método. Por fim, o método retorna uma referência do tipo *Object*. Portanto, é preciso fazer uma conversão de tipo antes que este possa ser associado a variável em questão.

Listagem A.21: Construtor utilizado na serialização da classe *GeradorSinal* gerado pelo gabarito.

```
1 protected GeradorSinal(SerializationInfo info , StreamingContext ctxt)
2 {
3     //TODO: remover a próxima linha após terminar a implementação
4     throw new Exception("The method or operation is not implemented.");
5
6     //TODO: Adicionar código para deserializar os membros do seu objeto.
7     //Ex: _nome = (string)info.GetValue("Nome", typeof(string));
8 }
```

A listagem A.22 apresenta a implementação do construtor para a classe *GeradorSinal*. Ele deve realizar a operação inversa do método *GetObjectData*, inclusive seguindo a mesma ordem na obtenção dos parâmetros. Como, na implementação do método *GetObjectData*, o primeiro elemento a ser serializado foi o número de canais, o código começa obtendo o número de canais (linha 5). Observe que a chave “NroCanais” é a mesma utilizada no método *GetObjectData*. Como o número é representado por um valor do tipo *int*, é passado o seu tipo através utilização da palavra chave *typeof*. Uma conversão de tipo é realizada, colocando o nome do tipo entre parênteses antes de associá-lo à variável *nroCanais*. Em seguida é criado o vetor de canais e cada elemento é deste vetor é obtido do vetor de bytes utilizando procedimentos análogos. Por fim, os demais elementos da classe são deserializados. Como este é um sistema virtual, isso conclui a implementação do construtor. Em um sistema real, seria necessário ainda abrir o canal de comunicação com

o equipamento e configurá-lo com este estado, fazendo com que o modelo computacional fique em sincronia com o estado real do equipamento.

Listagem A.22: Implementação do construtor utilizado na serialização da classe *GeradorSinal*.

```

1  protected GeradorSinal(SerializationInfo info , StreamingContext ctxt)
2  {
3
4      //Obtém o número de canais
5      int nroCanais = (int)info.GetValue("NroCanais", typeof(int));
6      //Cria o vetor
7      _canais = new Canal[nroCanais];
8
9      //Obtém o estado de cada canal
10     for (int i = 0; i < nroCanais; i++)
11     {
12         _canais[i] = (GeradorCanal)info.GetValue("Canal" + i, typeof(
13             GeradorCanal));
14     }
15
16     //Obtém os demais parâmetros
17     _freqAquisicao = (float)info.GetValue("FrequenciaAquisicao", typeof(
18         float));
19     _freqSenoide = (float)info.GetValue("FrequenciaSenoide", typeof(float))
20         ;
21     _nome = (string)info.GetValue("Nome", typeof(string));
22     _versao = (int)info.GetValue("Versao", typeof(int));
23 }

```

Os processos de serialização descritos até o momento, também devem ser realizados na classe que modela o canal. Ela também deve possuir o método *GetObjectData* e o construtor especial. Estes métodos também foram gerados pelo gabarito, cabendo ao desenvolvedor do *plug-in* apenas fornecer a sua implementação. A Listagem A.23 apresenta a implementação do método e do construtor especial para a classe *GeradorCanal*. Ao

observar a listagem, percebe-se que o procedimento é análogo ao da classe *GeradorCanal*. Uma pequena diferença pode ser notada na linha 11, onde é realizada uma chamada do método *GetObjectData* da classe base. Isso é necessário para que sejam serializados os membros presentes na classe *Canal* que não podem ser acessados pelas suas classes derivadas.

Listagem A.23: Implementação da serialização na classe *GeradorCanal*.

```
1 protected GeradorCanal(SerializationInfo info , StreamingContext ctxt)
2     : base(info , ctxt)
3 {
4     //Deserializa os membros da classe
5     _ganho = (float)info.GetValue("Ganho" , typeof(float));
6 }
7
8 public override void GetObjectData(SerializationInfo info , StreamingContext
    context)
9 {
10    //Chamada da classe base
11    base.GetObjectData(info , context);
12
13    //Serializa os membros da classe
14    info.AddValue("Ganho" , _ganho);
15 }
```

A.5 Considerações finais

Para aumentar a compatibilidade com o maior número possível de equipamentos, o *framework* não possui propriedades ou métodos para definir a frequência de aquisição. Esta frequência geralmente é definida através de uma das propriedades que são acessadas por reflexão. Além disso, os dados são enviados para um *buffer* circular de tamanho limitado. Portanto, ao utilizar uma frequência de aquisição muito alta, é possível que ocorra um “estouro” do *buffer*. Para contornar essa limitação, a propriedade *BufferSize* do *Canal*

pode ser utilizada para alterar a capacidade deste *buffer*. Definir esta capacidade é de responsabilidade do *plug-in*, sendo que a mesma é diretamente proporcional à frequência de aquisição. Os tamanhos desses *buffers* podem ser definidos no construtor da classe que modela o equipamento, assumindo um valor que comporte a frequência máxima de aquisição, ou alterados quando o usuário define a frequência de aquisição.

Outro fato que deve ser observado na criação de um *plug-in*, é o gerenciamento de recursos do sistema. Um dos recursos que todo *plug-in* irá utilizar é o canal de comunicação com o equipamento. Este recurso deve ser retornado ao sistema assim que o *plug-in* for destruído, por isso, é recomendável que o *plug-in* possua um destrutor que libere este recurso.

Referências

ABBATE, J. Getting small: A short history of the personal computer. *Proceedings of the IEEE*, v. 87, n. 9, p. 1695–1698, 1999.

AITKEN, P. G. *Get Started with Multithreading in .NET*. 2002. Disponível em: <http://www.devx.com/dotnet/article/7003>. Acesso em: 30 jan. 2007.

ANDRADE, A. de O. *Metodologia para Classificação de Sinais EMG no Controle de Membros Artificiais*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Janeiro 2000.

BROOKS, J. F. P. No silver bullet: Essence and accidents of software engineering. *Computer*, v. 20, n. 4, p. 10–19, 1987.

CHARTIER, R. *Serialization in the .NET Framework*. 2002. Disponível em: <http://www.15seconds.com/issue/020903.htm>. Acessado em 30 jan. 2007.

DEITEL, H. M. et al. *C# How to Program*. [S.l.]: Prentice Hall, 2001.

DEMERS, F.-N.; MALENFANT, J. Reflection in logic, functional and object-oriented programming: a short comparative study. In: *IJCAI'95, Workshop on Reflection and Metalevel Architectures and their Applications in AI*. Montreal, Quebec, Canadá: [s.n.], 1995. p. 29–38.

DILASCIA, P. Wrappers: Use our manwrap library to get the best of .net in native c++ code. *MSDN Magazine*, v. 20, n. 4, 2005.

- GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1. ed. [S.l.]: Bookman, 1995.
- GESELOWITZ, D. B.; GESELOWITZ, M. N. The bioelectrical century: bioelectrical engineering and the “inside story” of the electrical century. *Proceedings of the IEEE*, v. 87, n. 10, p. 1842–1846, 1999.
- GRAY, J. A java-based approach for teaching principles of adaptive and evolvable software. *Science of Computer Programming*, v. 53, n. 1, p. 21–69, 2004.
- HAUX, R. Aims and tasks of medical informatics. *International Journal of Medical Informatics*, v. 44, n. 1, p. 9–20, 1997.
- HOLY, T. et al. Data acquisition and processing software package for medipix2. *Nuclear Instruments & Methods in Physics Research*, v. 563, p. 254–258, 2006.
- JOHNSON, R. E. Frameworks = (components + patterns). *Communications of the ACM*, v. 40, n. 10, p. 39–42, 1997.
- KAISER, D. A. Basic principles of quantitative eeg. *Journal of Adult Development*, v. 12, n. 2-3, p. 99–104, 2005.
- KOLDITZ, O.; BAUER, S. A process-oriented approach to compute multi-field problems in porous media. *Journal of Hydroinformatics*, v. 6, n. 3, p. 225–244, 2004.
- KURIAN, S.; PONT, M. J. The maintenance and evolution of resource-constrained embedded systems created using design patterns. *Journal of Systems and Software*, v. 80, n. 1, p. 32–41, 2007.
- MAES, P. Concepts and experiments in computational reflection. In: *Object-oriented programming systems, languages and applications, OOPSLA*. Orlando, FL: [s.n.], 1987. p. 147–155.
- MICROSOFT. *Dev Center - .NET Framework*. 2007. Disponível em: <http://www.microsoft.com/brasil/msdn/framework/Default.aspx>. Acesso em: 19 mar. 2007.

MICROSOFT. *ICloneable - .NET Framework Class Library Reference*. 2007. Disponível em: [msdn2.microsoft.com/pt-br/system.icloneable\(VS.80\).aspx](http://msdn2.microsoft.com/pt-br/system.icloneable(VS.80).aspx). Acesso em: 22 jan. 2007.

MICROSOFT. *Microsoft Visual Studio Express Editions*. 2007. Disponível em: <http://msdn.microsoft.com/vstudio/express/default.aspx>. Acesso em: 19 mar. 2007.

MICROSOFT. *Reflection - .NET Framework Class Library Reference*. 2007. Disponível em: [http://msdn2.microsoft.com/pt-br/library/system.reflection.emit\(VS.80\).aspx](http://msdn2.microsoft.com/pt-br/library/system.reflection.emit(VS.80).aspx). Acesso em: 22 jan. 2007.

MORTON, D. What difference did semiconductors and microelectronics make? *Proceedings of the IEEE*, v. 87, n. 6, p. 1049 – 1052, 1999.

MULLICK, R. et al. Plug-ins: A software model for biomedical imaging & visualization research. In: *Biomedical Imaging Symposium: Visualizing the Future of Biology and Medicine, National Institutes of Health (NIH)*. [S.l.: s.n.], 1999.

NAVES, E. L. M. *Desenvolvimento de uma plataforma de força para análise da performance biomecânica*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Setembro 2001.

NESS, S. V. *Copying, Cloning, and Marshalling in .NET*. 2002. Disponível em: <http://www.ondotnet.com/pub/a/dotnet/2002/11/25/copying.html>. Acessado em 30 jan. 2007.

PAPAJORGJI, P. A plug and play approach for developing environmental models. *Environmental Modelling & Software*, v. 20, n. 10, p. 1353–1357, 2005.

RIVIÈRES, J. d.; WIEGAND, J. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, v. 43, n. 2, p. 371–383, 2004.

SÁ, Â. A. R. de. *Uma Proposta de Sistema Computacional em Tempo Real para Biofeedback*. Dissertação (Mestrado) — Universidade Federal de Uberlândia, Maio 2004.

SUTHERLAND, D. H. The evolution of clinical gait analysis part I: kinesiological emg. *Gait and Posture*, v. 14, n. 1, p. 61–70, 2001.

TOUB, S. Write faster code with the modern language features of visual c++ 2005. *MSDN Magazine*, v. 19, n. 5, 2004.

VINCENZI, A. M. R. *Orientação a objeto: definição, implementação e análise de recursos de teste e validação*. Tese (Doutorado) — Universidade de São Paulo, 2004.

WILLIAMS, M. *.NET Serialization*. 2001. Disponível em:
<http://www.codeguru.com/columns/DotNet/article.php/c6595/>. Acessado em 30 jan. 2007.