A programação Windows integrada com Delphi para o desenvolvimento de Recursos Computacionais voltados a Acessibilidade de Tetraplégicos com a fala comprometida

França, C. R.

***Abstract*— This paper presents the most relevant programming techniques used in the creation of Windows software components, whose use is not easily learned with a simple reading of the manuals or online help. We describe the technical details of implementing adaptive systems in the Windows environment, in particular on the use of certain APIs (Application Programming Interfaces), a mechanism called "Windows Hooks."**

***Keywords*— Windows environment, Toolkit of components , Adaptative Technology, Development motor impaired.**

I. INTRODUÇÃO

Neste artigo apresentam-se as técnicas mais relevantes da programação Windows utilizadas na criação de Componentes de software, cujo uso não é facilmente aprendido com a mera leitura dos manuais ou *help on-line*. São descritos os detalhes técnicos de implementação de sistemas adaptativos no ambiente Windows, em especial sobre a utilização de algumas APIs (Application Programming Interfaces), do mecanismo conhecido como "*Windows Hooks*".

Este trabalho é fruto da experiência do autor em desenvolvimento de componentes para criação de software para pessoas tetraplégicas, tendo sido aplicado no desenvolvimento de um Toolkit de Componentes de Software, denominado Toolkit Tupi. O referido trabalho foi executado para a obtenção do título de Mestre em informática do Programa de Pós-graduação do Núcleo de Computação e Eletrônica da Universidade Federal do Rio de Janeiro - NCE/UFRJ.

II. O DESENVOLVIMENTO DE SOFTWARE DE ACESSIBILIDADE, COM REUSO DE COMPONENTES DE UM TOOLKIT.

O desenvolvimento de software de acessibilidade tem como forte característica à interação com o sistema operacional. O TOOLKIT TUPI foi implementado na linguagem Delphi, mas toda a sua concepção envolve os conceitos que permeiam a programação Windows. A programação em Delphi, para a maior parte das aplicações, encapsula as chamadas ao sistema operacional na forma de componentes que ocultam os detalhes de programação do sistema operacional Windows. Porém, na programação do Tupi, esses componentes facilitadores não estavam disponíveis, sendo necessário o seu desenvolvimento. Para tanto necessitou-se de uma intensa programação no nível do sistema operacional Windows, bem como o uso de técnicas especiais para sincronismo com os elementos internos deste sistema.

O desenvolvimento do TUPI envolveu a execução de tarefas complexas de programação, que fazem uso das chamadas "interfaces com programas de aplicação" (*application program interfaces* ou API's) e técnicas que dão acesso não convencional ao sistema operacional Windows, em especial:

a) Execução de procedimentos que interferem nas filas de processamento do Windows.

b) Operações de entrada e saída não suportadas diretamente pelo sistema operacional.

A maior parte destas técnicas é muito pouco conhecida no Brasil, mesmo entre programadores experientes. Por este fato, é relevante mostrar aqui uma coletânea dos mecanismos mais complexos que foram utilizados na implementação do TUPI, com o objetivo de gerar uma referência para programadores que desejem ampliar ou realizar manutenção em sistemas adaptativos ou mesmo sistemas não convencionais que executem no ambiente Windows.

Carlos Roberto França - Universidade Federal da Fronteira Sul - Campus Realeza, carlos.franca@uffs.edu.br

Inicia-se com uma introdução para apresentar alguns pontos chaves e uma breve contextualização técnica sobre o mecanismo genérico de acionamento das APIs, o que envolve também o entendimento de alguns mecanismos como *callback (*chamadas ao programa de aplicação realizadas pelo sistema operacional) e *hooks* (ganchos utilizados para colocar uma tarefa na fila de execução do Windows). Em seguida são apresentadas as soluções que foram implementadas no TUPI relativas ao acoplamento com os *hooks*.

III. ENTENDENDO O MECANISMO UTILIZADO NAS DLLS DO WINDOWS

a) mensagens

O Windows é um sistema operacional orientado a mensagens. Muitos pedidos ao sistema operacional, bem como muitas das respostas enviadas por ele, especialmente os que são realizados de forma assíncrona, são encapsulados em "mensagens",(que são estruturas de dados padronizadas que definem os "detalhes" destes pedidos ou respostas). As mensagens são colocadas em filas por rotinas como PostMessage (envio assíncrono), SendMessage (envio síncrono) , PeekMessage (recepção assíncrona) e GetMessage (recepção síncrona).

b) DLLs

Segundo Hetch [Hetch, 2001] [1], todos sistemas Windows são construídos na forma de "bibliotecas dinâmicas" (DLL – dynamic link libraries), que podem ser compartilhadas por múltiplas aplicações. O código e os recursos contidos nas DLLs não são geralmente carregados na memória até a hora em que são necessários e podem ser descartados quando termina seu uso. A utilização de DLLs, em comparação com a técnica antiga de *linkage-edition* apresenta diversa vantagens, como:

- economia de espaço em disco;

- a possibilidade de fazer uma atualização e trocar partes de um programa sem ter que recompilar tudo;

- a possibilidade de carregar diferentes recursos e código baseado em algum critério específico, disponibilizado apenas em *runtime*.

O pequeno trecho a seguir, baseado em [Hetch,2001] [1], exemplifica o esquema básico de programação de uma rotina simples de uma das principais APIs do Windows: a USER32, responsável por grande parte da interface com o usuário, além do processamento do teclado e de mouse. Nele é mostrado a carga da DLL na memória, o uso seguro de um de seus métodos, e a descarga da DLL da memória.

```
Procedure ExecuteBeep;
Var
   DllHandle:Thandle;
   BeepFn:function(BeepType:integer): Bool sdtcall;
Begin
  // responsável por carregar a biblioteca USER32 na memória
  DllHandle:=LoadLibrary('USER32');
  // assegura-se que conseguiu carregar
  If DllHandle<>0 then
   begin
        // obtém o endereço da rotina de bip, permitindo a sua manipulação.
        @BeepFn:= GetProcAddress(DllHandle,'MessageBeep');
        // testa se a função de bip pode ser utilizada.
        if @BeepFn<> nil then
             // executa um tipo específico de bip
             BeepFn($FFFFFFFF);
        // libera a biblioteca dinâmica da memória
        FreeLibrary(DllHandle);
   End;
End;
```

Como é fácil perceber pela leitura deste trecho, é muito complicada uma interação direta com o sistema operacional, mesmo para realizar uma tarefa tão trivial como essa, dar um bip, dada a quantidade de detalhes específicos envolvidos. Assim é fundamental a criação de mecanismos de encapsulamento de detalhes que atendam às situações mais comuns.

c) Funções de chamada de retorno (callback)

Numa situação simples, o sistema operacional é chamado para realizar alguma tarefa e na maior parte das vezes o programa espera que esta tarefa seja concluída para continuar. Esse mecanismo, entretanto, não atende às necessidades de um processamento interativo assíncrono,

pela própria natureza deste processamento, onde a execução de uma tarefa não pode ser descrita por uma função de tempo linear.

As funções de chamada de retorno permitem especificar o endereço de uma função que será chamada automaticamente pelo sistema operacional ou por uma DLL na ocorrência de um evento qualquer (por exemplo, ao término da ação pedida). Desta forma é criada uma situação em que o sistema operacional não é apenas chamado pelo programa do usuário, mas ele próprio pode chamar o programa de aplicação quando necessário.

O uso mais trivial deste mecanismo é a execução de operações de Entrada e Saída, em que um erro ou término de processamento, ativa uma rotina do programa de aplicação permitindo assim que o tratamento seja realizado de forma completamente assíncrona.

d) Recursos (Resources)

Um caso particular de utilização de bibliotecas pode ser dado pelas utilizações de interfaces gráficas e textos de mensagens. Um compilador especial (*Resource Compiler)* permite a criação de uma estrutura de dados que especifica o posicionamento de botões, rótulos, *listbox*, etc, que são na verdade rotinas implementadas dentro de DLLs específicas, bem como os textos que são usados na formatação de telas e relatórios. Em tempo de execução, essa estrutura de dados é acessada pelo programa e entregue ao sistema operacional, que realiza, em *background*, as seqüências de chamadas referentes à manipulação destes componentes, inclusive as funções de *callback*.

e) componentes

A maior dificuldade de programação do Windows está no número imenso de mensagens, APIs e DLLs que existem (muitas centenas) cada uma das quais disponibilizando inúmeras funções, somado a uma fenomenal quantidade de convenções de que essas funções fazem uso (tais como códigos de ativação e interpretação de retornos, por exemplo). Segundo Piccard [2], por definição, cada APIs deve incluir uma descrição de funções realizadas pelo módulo e os mecanismos usados para passar informação para dentro e para fora do módulo. Tipicamente isso vai incluir a especificação do:

- Número e ordem dos argumentos;
- Para cada argumento, o tipo de variável e se a informação é passada por valor ou por referência.
- Informações de localidade em relação a determinados módulos.

Essa complexidade fez com que hoje em dia o uso de componentes que encapsulam ou reorganizam este imenso número de detalhes se torne cada vez mais comum na programação de Windows. Dependendo do uso pretendido, os componentes podem ser implementados na forma de bibliotecas específicas ou então num padrão conhecido como "Component Object Model", cuja interface mais usada recebe o nome genérico de Active X, [3] que é um mecanismo genérico criado pela Microsoft para permitir o uso destes componentes em diversas linguagens ou em aplicativos específicos (como editores de textos e planilhas, entre outros).

IV. ACESSO A DISPOSITIVOS

No Windows, todos os dados que são provenientes ou que se destinam a um dispositivo externo, são controlados por uma rotina específica (device driver), geralmente construída pelo fabricante do *hardware*, e que se encarrega de tratar de todos os detalhes específicos da entrada ou saída. O Windows mantém para cada dispositivo, filas de dados que serão escritos ou que foram lidos, e que provêm dos *driver*, pois a E/S quase sempre ocorre de forma assíncrona. Entre os *driver* e os *buffer* que receberão ou que contém os dados, existe um complexo mecanismo de troca de mensagens, que visa estabelecer proteção num sistema multiprogramado como o Windows.

Todo este processo é apresentado para o usuário através de uma interface de programa de aplicação (API), no qual os detalhes internos são escondidos. A API é basicamente um conjunto de rotinas ou macros que fazem acessos às funções da DLL.

O acesso direto aos *device driver* quase nunca é realizado, sendo normalmente mediado por chamadas específicas a APIs particulares do sistema operacional. Essas APIs mantém para cada dispositivo, filas de dados que serão escritos ou que foram lidos, e que provêm dos driver, pois a E/S quase sempre ocorre de forma assíncrona. Entre os *driver* e os *buffer* que receberão ou que contém os dados, existe um complexo mecanismo de troca de mensagens, que visa estabelecer proteção num sistema multiprogramado como o Windows.

V. TRATAMENTO DE INTERRUPÇÕES NO WINDOWS

Como descrito no artigo "Programming Interrupts for DOS-Based Data Acquisition on 80x86-Based Computers" [National Instruments] [4] desde a época do sistema operacional DOS uma prática para acessar o fluxo destinado aos *driver* era dado a partir de uma tabela de tratamento de interrupções de acesso, causando um desvio para uma rotina especifica e vice e versa. (fig .1)

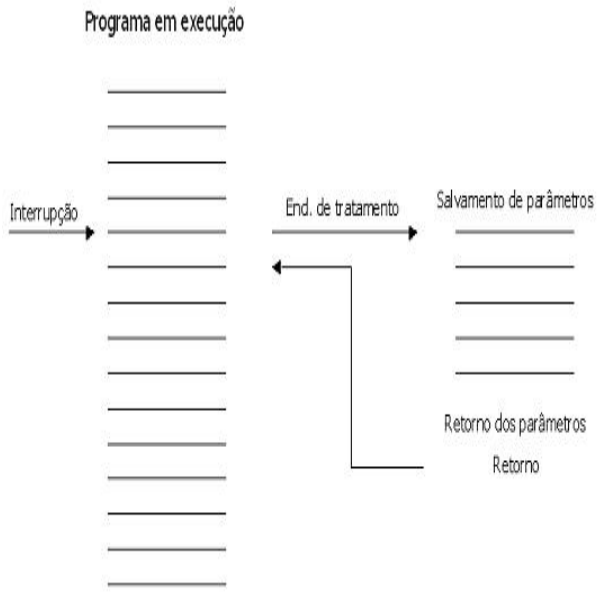

Figura 1. tratamento de interrupções

Como mecanismo de proteção, o Windows inibe, em geral, o uso dessas técnicas, permitindo que o acesso a interrupções seja feita unicamente através de rotinas certificadas (driver) e que obedecem a convenções muito específicas de sistema operacional.

1- O código precisa estar dentro de uma DLL;

2- O código precisa exibir uma execução muito rápida;

3- O programador terá que prover o encadeamento de interceptações de modo a coordenar o fluxo das mesmas.

Para acessarmos, adicionarmos ou interceptarmos alguma informação proveniente de periférico (na verdade não apenas a fila de E/S, mas até mesmo a fila de mensagens do sistema) uma alternativa viável seria a reescrita de *driver* para incorporar as tarefas especificas, de acordo com as necessidades. Esta alternativa deve, entretanto ser descartada pelo o fato das funcionalidades mais especificas serem de acesso exclusivo do fabricante dos *driver*.

O mecanismo usado para atender a estes requisitos são os *Windows Hook* que, através do qual uma sub-rotina é colocada no caminho do mecanismo normal de tratamento de mensagens do Windows.

VI . WINDOWS HOOKS

Como Rutledge descreve em seu artigo "Advanced Delphi – Windows Hooks"[5], uma "hook procedure" captura do sistema certas mensagens do Windows antes delas serem enviadas para as devidas rotinas de tratamento. No sistema operacional Windows existem vários tipos diferentes de "procedures hook", onde cada um deste tipo fornece um aspecto diferente do mecanismo de tratamento de mensagens do Windows. É, portanto uma função que pode ser introduzida no sistema de mensagem do Windows, assim uma aplicação pode acessar a mensagem corrente antes de outro processo tomar o lugar no processamento.

O quadro 1 - abaixo descreve os diversos tipos de *hooks* que são disponíveis no Windows.

WH_CALLWNDPROC	é chamada sempre que a função *SendMessage()* for chamada..
WH_CBT	usada especialmente para implementação de treinamentos, interceptando o sistema antes da chamada computacional de: ativação; criação; destruição; minimização; maximização; movimentação ou dimensionamento de uma janela; antes de completarem um comando de sistema ; antes de um novo movimento do mouse ou um evento do teclado; antes de ajustar o foco de entrada ; e antes da sincronização do sistema da fila de mensagens.

WH_DEBUG	Permite a criação de uma rotina de debug que atuará antes da chamada de um filtro instalado.
WH_GETMESSAGE	faz uma chamada de mensagem sempre que a função *GetMessage()* tenha recuperado uma mensagem na fila de aplicação
WH_HARDWARE	usada sempre que ocorre uma chamada de mensagem hook sem ser padrão de hardware, a aplicação chama as funções *GetMessage()* ou *PeekMessage()* e um evento de hardware (como um evento de mouse, teclado ou outro para processar).
WH_JOURNALRECORD	tem por função instalar uma **procedure hook** especialmente para gravar as mensagens de entrada afixadas na fila de mensagem do sistema pela *WH_JOURNALPLAYBACK*
WH_JOURNALPLAYBACK	tem por função enviar mensagens previamente gravadas pela a hook. WH_JOURNALRECORD. O tipo da função "callback", chamada de retorno, é jounalRecordProc.
WH_KEYBOARD	é chamada sempre que uma aplicação chamar as funções *GetMessage()* ou *PeekMessage()* e existir uma mensagem de teclado para processar, *WM_KEYUP* ou *WM_KEYDOWN* .
WH_MOUSE	é chamada sempre que uma aplicação chamar as funções *GetMessage()* ou *PeekMessage()* e existir uma mensagem de mouse para ser processada.
WH_MSGFILTER-	é uma chamada de mensagem hook para uma aplicação depois de uma caixa de diálogo, uma mensagem de caixa ou um menu de mensagem recuperada , porém antes a mensagem é processada.
WH_SHELL	É uma procedure de chamada da aplicação Shell , quando uma mensagem de notificação do sistema tiver sido feita.
WH_SYSMSGFILTER	Essa procedure faz uma chamada de sistemas após uma caixa de diálogo, uma mensagem de caixa, ou menu de recuperação de uma mensagem, antes que a mensagem esteja processada.

O Windows mantém internamente um "*hook chain*", que funciona como uma lista de ponteiros para as *"procedure hook*" com finalidades idênticas que os diversos programas instalaram. Este procedimento do sistema operacional Windows se faz necessário pelo fato de muitos programas instalarem uma *"procedure hook"*. Quando acontece uma chamada de mensagem no sistema, o Windows primeiro passa por cada uma das procedures no *"hook chain"* uma depois da outra. Então, caso a mensagem não tenha sido bloqueada por qualquer uma das *"procedure hook*", o Windows encaminha a mensagem para a janela adequada.

VII. IMPLEMENTANDO UMA *WINDOWS HOOK* PARA *JOURNAL PLAYBACK*: A DLL DVKBM32

Cada *Windows hook* possui um certo objetivo específico, mas existe uma grande intersecção entre as funções que elas executam. Na implementação do Tupi os requisitos para definição desses objetivos eram simples: deveria ser possível introduzir elementos que permitissem simular eventos de mouse e teclado, introduzindo novos itens nas filas respectivas.

A escolha da Windows Hook específica deveria atender também a alguns critérios importantes

a) Aplicabilidade a um grande número de situações: dada a complexidade de implementação, seria importante pensar em uma solução que pudesse ser aproveitada em situações similares, sem modificação.

b) Sincronismo de execução: O funcionamento do Hook deveria permitir um uso completamente síncrono, o que permitiria o seu uso em algumas situações ou obriga a criação de "threads de execução" para evitar o bloqueio de processamento. Apesar disso, em todo desenvolvimento do TOOLKIT TUPI, não houve nenhum momento em que esse sincronismo trouxesse maiores dificuldades à programação.

c) Encapsulamento das dificuldades: Os procedimentos e funções de ativação deveriam ser o mais invisível possível.

A solução usada foi baseada em uma implementação já existente, criada originalmente para uso nos sistemas DOSVOX e MOTRIX, mas cuja documentação inexistia até a criação do TUPI: a DLL DVKBM32. Segundo uma entrevista realizada com o desenvolvedor, professor Antonio Borges, do NCE/UFRJ "o *principal interesse era sintetizar as ações e eventos que teríamos que lidar nos nossos desenvolvimentos e principalmente evitar a necessidade do tratamento de conflitos e interrupções alheias à nossa aplicação. A escolha recaiu sobre a Windows Hook JournalPlayback por ela ser de uso amigável e poder comunicar-se sem conflitos com a maioria das APIs do Windows.* "

A DLL DVKBM32 basicamente mantém uma fila de ações desejadas e seu tempo de duração pretendido. O usuário pode inserir nesta fila pedido de ação de mouse e teclado. Uma rotina chamada PlayStart dá início ao processamento, estabelecendo o *hook*. Quando a fila está vazia, a própria rotina desestabelece o hook.

Neste ponto, é oportuno destacar que a reusabilidade e a usabilidade de componentes são algumas das principais aplicabilidades de um toolkit. A reusabilidade esta caracterizada pelo reuso de componentes sem que haja mudanças nas suas linhas de código, mas com aplicabilidades diferentes das que o originaram. Ou seja: O componente permanece com as suas características, mas a sua empregabilidade foi diversificada. Quando o componente é utilizado na resolução do problema alvo que o originou, diz-se se tratar de uma usabilidade. Na verdade, todas e quaisquer implementações que envolvem componentes de um toolkit, estarão fortemente ligadas à interface dos mesmos e as configurações que permitiram os ajustes e padronizações de acordo com os objetivos do desenvolvedor.

Por sua vez, o que garante a utilização das aplicabilidades, configurações e padronizações dos componentes de um toolkit, são os motivos que levaram a utilização do mesmo na resolução de determinados problemas. As facilidades técnicas do conjunto de ferramentas podem ser mais ou menos aproveitadas, de acordo com a habilidade do desenvolvedor. A construção de Software baseada e/ou orientada a componentes, utilizam padrões de projetos que implementam métodos como o *Catalysis*, e uma linguagem de programação de alto nível orientada a objetos, como por exemplo, o Delphi.

O quadro 2 lista as funções disponibiliza na DLL DVKBM32

PlayInit	cria um array que conterá a lista de ações desejadas. Uma das informações que será armazenada é o tempo que será gasto em cada ação.
PlayMouse	insere no array as informações sobre uma ação de mouse (mover, clicar, e as coordenadas).
PlayKey	insere no array as informações sobre a tecla a ser "pressionada"
PlayVirtKey	semelhante a PlayKey informando código virtual e não o ASCII
PlayAltKey	idem para tecla ALTs
PlayInsertKeyEvent	insere um evento de baixo nível de teclado (como a informação de apertar e desapertar)
PlayStart	ativa o hook
PlayIsActive	Função que vê se a fila ficou vazia nesta DLL

A JournalPlayback é ativada como um callback do Windows, em que são especificados os seguintes tipos de mensagem:

HC_SKIP – o windows indica que está pronto para atender a um pedido

HC_NEXT – o windows indica que acabou de atender ao último pedido

Outros tipos de mensagem podem ser enviados pelo Windows a uma JournalPlayback, mas são pouco importantes para nossa implementação.

A listagem a seguir apresenta o algoritmo utilizado no "núcleo" desta DLL

```
function PlaybackProc(Code: Integer;
        wParam: TwParam;
        lParam: TlParam): Longint; stdcall;
var
   TimeToFire: Longint;

begin
   PlaybackProc := 0;
   case Code of

      HC_SKIP:
      begin
         CurrentMsg := CurrentMsg + 1;
         if CurrentMsg >= MsgCount then
           if TheHook <> 0 then
             if UnHookWindowsHookEx(TheHook) then
               begin
                 TheHook := 0;
                 FreeMem(PMsgBuff, Sizeof(TMsgBuff));
                 PMsgBuff := nil;
               end;
          exit;
      end;

      HC_GETNEXT:
      begin
        PEventMsg(lParam)^ := PMsgBuff^[CurrentMsg];
        PEventMsg(lParam)^.Time :=
             StartTime + PMsgBuff^[CurrentMsg].Time;
        TimeToFire :=
             PEventMsg(lParam)^.Time - GetTickCount;
        if TimeToFire > 0 then
             PlaybackProc := TimeToFire;
        exit;
      end;

.......

   If code < 0  then
     PlaybackProc :=
       CallNextHookEx(TheHook, Code, wParam, lParam);
end;
```

IX. UMA INTERFACE SIMPLIFICADA PARA A DLL: DVMACRO

Para uso prático da DLL foi criada uma interface simples, que encapsulasse as funções mais utilizadas, denominada DVMACRO. A funcionalidade desta interface foi talhada para as necessidades dos software de acessibilidades para deficientes visuais (DOSVOX), e para deficientes motores (MOTRIX), e recentemente, com as necessidades do Tupi. Os procedimentos e funções da DVMACRO são responsáveis pela a ativação dos *hooks procedures* utilizadas em todos os componentes do TOOLKIT TUPI.

Além de encapsular diretamente as rotinas básicas da DVKBM32, esta interface ainda implementa diversas funções de conveniência que realizam a execução de forma síncrona de diversas ações (em outras palavras, uma seqüência de PlayInit – Ação específica – PlayStart – e espera terminar), para inserção de uma ou várias teclas, diversos tipos de cliques no mouse e de dragging de cursor.

VII. ACESSO À PORTA PARALELA DO PC EM WINDOWS NT E SUCEDÂNEOS

A porta paralela do PC, utilizada habitualmente para conexão de impressoras, é muito adequada para conexão de dispositivos não convencionais. Os sinais que são produzidos nos pinos de saída desta *interface* ou que são lidos através de seus pinos de entrada têm um sistema de proteção e amplificação de sinal que simplifica muito a conexão de chaves, leds, circuitos acionadores, e outros dispositivos cuja conexão tenha características *ON/OFF*, ou seja, baseada em interfaces digitais.

A referência [Axelson, 1998] [6] é um bom guia para programação de portas paralelas em diversas situações. Um exemplo tirado do livro de Gabriel Torres [Torres, 2001] [7], demonstra que a programação de acesso à porta paralela é uma tarefa bem simples a partir dos endereçamentos da porta paralela. Os códigos de acesso podem ser gerados em Assembly, utilizando as instruções *IN e OUT* para leitura e escrita, respectivamente, ou em Pascal como o exemplo abaixo.

Em Pascal para MS-DOS, utiliza-se o comando PORT, conforme o exemplo abaixo.

```
program portaParalela;
uses crt;
begin
   // coloca o valor 255 (ffh) na porta paralela
   port [$378]:= 255;
end.
```

Escrever programas para se comunicar via porta paralela, nos sistemas operacionais Win95/98, é tarefa relativamente fácil a partir dos comandos do MS-DOS ou utilizando o endereçamento da porta paralela, como mostrado acima. Mas quando a programação é feita

nos sistemas operacionais Win NT, Win2000 e WinXP, toda a simplicidade desaparece, pois nestes sistemas operacionais, o acesso a dispositivos é privilégio exclusivo dos *driver* do sistema operacional, e um acesso direto a portas paralelas através dessas mesmas rotinas provoca uma exceção de instrução privilegiada, geralmente associada a uma mensagem como na fig. 2

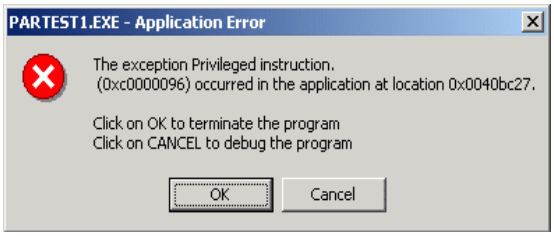

Figura 2 . Mensagem de erro

O Windows NT atribui alguns privilégios e limitações aos tipos diferentes de programas que funcionam nele a partir da classificação dos programas de acordo com o seu modo de execução. Os programas podem executar em modo usuário ou pelo *kernel mode*, e os programas serão alocados para rodarem em camadas ou anéis do núcleo *(Kernel)*.

No modo usuário, os programas rodam na camada três ou *ring3Mode*, como é mais conhecido na programação Windows, e são restritas as instruções do tipo entrada e saída (*in/out) e outras similares* no modo *kernel*, os programas rodam na camada zero ou *ring0Mode*, são programas do próprio sistema operacional. Este modo de execução não possui as restrições do modo usuário.

Segundo [Porttalk, 2000] [8] existem duas formas de solucionar essas restrições sob Windows NT. A primeira solução é um criar/utilizar um Device Driver que rode em Ring0Mode, o que lhe permite ter acesso irrestrito às portas. A segunda é alterar o "I/O permission bitmap" do sistema, estrutura de dados que é usada pelo Windows para controlar o acesso a certas portas. Essa mesma referência assinala que, mesmo sendo mais simples, essa segunda alternativa deve ser evitada, pois abre a possibilidade de acessos indevidos que removeriam a segurança do sistema.

É interessante mencionar que mesmo que se tenha conseguido encontrar um *driver* pronto para executar a tarefa desejada, instalá-lo e configurá-lo são tarefas temidas pela maior parte do staff técnico e de suporte, que tendem a olhar com desconfiança a instalação de itens de sistema operacional que não são "certificadas pela Microsoft". Por outro lado, escrever um excitador *(Driver*) não é um trabalho fácil, nem para programadores experientes, dado o enorme número de rotinas, estruturas de dados, convenções e padrões usuais de implementação que são descritas no Windows DDK (Device Driver Kit) [9] .

X. IMPLEMENTAÇÃO DO ACESSO À PORTA PARALELA NO TUPI

Assim, é interessante poder utilizar soluções prontas ou comerciais. Nossa busca na Internet demonstrou uma preponderância dois *drivers* para porta paralela específicos para determinados equipamentos: o PortTalk e o InpOut32, ambas com funcionalidade muito semelhante.

Na implementação do TUPI escolhemos a biblioteca de domínio público INPOUT32.DLL disponível em [http: //www. logix4u.net/inpout32.htm], que foi considerada uma das que melhor dá conta destas restrições, criando um acesso funcional em todas as versões do Windows (98, NT, 2000 e XP), através de um *driver* em "*kernel mode*" (*hwinterface.sys*), embutido na DLL.

A característica proeminente de Inpout32.dll, é poder trabalhar com todas as versões do Windows sem nenhuma modificação no código do usuário ou na própria DLL. A DLL verificará a versão do sistema operacional quando as funções são chamadas, e se o sistema se operacional for WIN9X, a DLL usará o *inp ()* e funções do *outp* para leitura/escrita a partir da porta paralela.

As duas funções principais que são exportadas pela inpout32.dll são:

1) ' **Inp32** ', lê dados de um registro especifico da porta paralela.

2) ' **Out32** ', escreve dados em um registro especifico da porta paralela.

As outras funções executadas pela Inpout32.dll estão expostas no quadro 3 abaixo:

a) Funções	**Atribuições**
DllMain	É uma função chamada quando a DLL é carregada ou descarregada. Quando a DLL é

	carregada, verifica a versão do sistema operacional e carrega *hwinterface.sys* se necessário for.
Closedriver	Esta função fecha o *driver* que realizou a chamada, antes de baixar o *drvier* que foi chamado.
Opendriver	É a função que abre um driver de chamada para a função *hwinterface*.
Inst	Tem por função extrair a função ' hwinterface.sys' do *driver* da raiz do sistema de diretórios binários e criam um serviço. Esta função é chamada quando a função ' Opendriver ' não abre um *driver* de chamada válido ao serviço ' do hwinterface '.
Start	Tem por função iniciar o serviço da função hwinterface utilizando o gerenciador de APIs(Application services) do controle de serviço.
SystemVersion	Esta função verifica a versão do sistema operacional e retorna o código apropriado.

Quadro 3 – Funções executadas pela Inpout32.dll

XI. CONSIDERAÇÕES FINAIS

Na implementação do TUPI foram encontrados muitos casos que envolvem interfaces pouco conhecidas com o Windows. Na maior parte dos casos, as soluções são pouco documentadas, mas conceitualmente simples ou de implementação rápida. Apresentou-se neste artigo, os dois casos mais característicos, e que foram mais difíceis de encontrar solução e implementá-la.

Julga-se, outrossim, importante que o toolkit seja mantido em "código aberto", para que, através do estudo aprofundado de seu código, outros programadores possam obter subsídios para solucionar grande número de situações simples que ocorrem no desenvolvimento de software adaptativos. Esse código servirá de documentação complementar ao material disponível nos manuais e livros textos, provendo exemplos práticos de implementações de um grande número de situações em especial relacionadas à programação no nível de software básico ou de sistema operacional no ambiente Windows.

XII. REFERÊNCIAS

[1] Joe C. Hecht, Borland Technical Support Group – Manual técnico da Borland

[2] http://oak.cats.ohiou.edu/~piccard/mis300/osapis.htm

[3]http://www.et.utt.ro/public/Docs/ActiveX%20Programming%20Unleashed/

[4] http://zone.ni.com/devzone/conceptd.nsf/webmain/

[5]http://www.prestwood.com/community/delphi/usergroup/newsletter/1998oct.html

[6] Axelson, J. - Parallel Port Complete - Lakeview Research – 1998

[7] http://geocities.yahoo.com.br/conexaopcpc/artigos/portas_paralelas.htm

[8] www.beyondlogic.org/**porttalk**/**porttalk**.htm

[9] Microsoft, www.microsoft.com/whdc/**ddk**/win**ddk**.mspx**.**

Carlos Roberto França, Mestre em Informática pelo IM/NCE/UFRJ, tendo defendido a sua dissertação intitulada: " Especificação e Desenvolvimento de um Toolkit para Construção de Ferramentas de Acessibilidade para Deficientes Motores ", em 2005, tem 17 anos de experiência em desenvolvimento de Tecnologia Adaptativa. Atualmente é Professor Assistente do quadro da Universidade Federal da Fronteira Sul - UFFS, Campus Realeza - PR