Capítulo

7

Comunicação via portas de E/S

Autor: Constantino Seixas Filho
UFMG – Departamento de Engenharia Eletrônica

Comunicação via Porta paralela

A motivação para escrever esta aula nasceu da necessidade de alguns alunos de utilizar a porta paralela do computador para algumas atividades de interface direta com dispositivos de hardware simples. Isto tem sido muito explorado em aulas de laboratório para comunicar um PC com uma placa de *protoboard* contendo uma meia dúzia de componentes.

As instruções **_inp()** e **_outp()** são a maneira mais rápida de acessar um dispositivo de entrada e saída, mas isto é vetado a programas desenvolvidos em Windows NT. É que seria muito arriscado deixar que usuário não habilitados escrevessem diretamente no hardware. Isto impede inclusive o acesso aos endereços reservados à saída paralela.

Instruções de acesso direto a E/S do Windows 95:

```
#include <conio.h>

int _inp( unsigned short port );

unsigned short _inpw( unsigned short port );

int _outp( unsigned short port, int databyte );

unsigned short _outpw( unsigned short port, unsigned short dataword );
```

Uma maneira controlada de acessar a porta paralela, seria através da instrução *DeviceIoControl()* do repertório Win32 que chama um *device driver* em modo kernel que por sua vez realiza as operações de I/O. Este método é eficaz quando não se está preocupado com a velocidade da interface. Para algumas aplicações entretanto este método é muito lento. Por exemplo, se você deseja medir a duração de uma tarefa representada por uma thread, através da ativação de um bit externo que possa ser visualizado em um osciloscópio, este método é muito lento. Isto foi usado nas experiências descritas no capítulo de programação em tempo real do livro texto (*Programação Multihreading em ambiente Win32*). Muitos outros dispositivos estão mapeados no espaço de E/S e até mesmo para se escrever na memória de vídeo temos que acessar endereços mapeados em I/O.

DeviceIOControl

BOOL DeviceIoControl(

```
HANDLE hDevice,                 // Handle para device, arquivo ou diretório
DWORD dwIoControlCode,          // Código de controle da operação a ser
                                   realizada
LPVOID lpInBuffer,              // Apontador para buffer contendo dados
                                   para realizar a operação
DWORD nInBufferSize,            // Tamanho do Inbuffer em bytes
```

LPVOID lpOutBuffer,	// Apontador para buffer para receber dados de retorno do comando
DWORD nOutBufferSize,	// Tamanho de Outbuffer em bytes
LPDWORD lpBytesReturned,	// Apontador para variável para receber contador de bytes retornados em OutBuffer
LPOVERLAPPED lpOverlapped	// Apontador para estrutura para operações assíncronas
);	

Observe que os nomes dos buffers de entrada e de saída podem induzir a equívocos. InBuffer é o buffer de envio de dados ao comando, enquanto OutBuffer é o buffer onde os resultados da execução do comando são colocados.

Comentários sobre os parâmetros:

HANDLE hDevice,	Handle para device, arquivo ou diretório retornado pela função *CreateFile().*
DWORD dwIoControlCode	**Leitura:** IOCTL_GPD_READ_PORT_UCHAR: unsigned char IOCTL_GPD_READ_PORT_USHORT: unsigned short IOCTL_GPD_READ_PORT_ULONG: unsigned long **Escrita:** IOCTL_GPD_WRITE_PORT_UCHAR: unsigned char IOCTL_GPD_WRITE_PORT_USHORT: unsigned short IOCTL_GPD_WRITE_PORT_ULONG: unsigned long

Retorno da função:

Status	**Interpretação**
<>0	Sucesso
0	Falha. Use *GetLastError()* para obter maiores detalhes do problema ocorrido.

Exemplo: leitura de porta

```
// Leitura de um byte a partir de uma porta usando driver wkd.
// Robert B. Nelson (Microsoft) January 12, 1993

hndFile = CreateFile(
        \\\\.\\GpdDev",// Abre o arquivo Device
        GENERIC_READ,
        FILE_SHARE_READ,
        NULL,
        OPEN_EXISTING,
        0,
        NULL);
```

```
if (hndFile == INVALID_HANDLE_VALUE) {// Conseguiu abrir ?
        printf("Não consegui abrir dispositivo.\n");
        exit(1);
}

HANDLE  hndFile;  // Handle do dispositivo obtido a partir de CreateFile
union   {
  ULONG   LongData;
  USHORT  ShortData;
  UCHAR   CharData;
}   DataBuffer;   // Buffer received from driver (Data).

LONG        IoctlCode = IOCTL_GPD_READ_PORT_UCHAR;
ULONG       PortNumber = 0x300;
ULONG       DataLength = sizeof(DataBuffer.CharData);
DWORD       ReturnedLength;// Number of bytes returned
BOOL        IoctlResult;

IoctlResult = DeviceIoControl(
                hndFile,                // Handle para dispositivo
                IoctlCode,              // IO Control code para leitura
                &PortNumber,            // Endereço da porta: 0x300..
                sizeof(PortNumber),     // Comprimento do buffer em bytes.
                &DataBuffer,            // Buffer de saída.
                DataLength,             // Tamanho do buffer de saída em  bytes.
                &ReturnedLength,        // Bytes recebidos em DataBuffer.
                NULL);                  // NULL = Síncrono

if (IoctlResult)                        // Leitura efetuada com sucesso ?
{
        ULONG   Data;

        if (ReturnedLength != DataLength)  {
                printf("Ioctl transferiu %d bytes, eram esperados %d bytes\n",
                returnedLength, DataLength );
        }

        switch (ReturnedLength)  {
                case sizeof(UCHAR):
                        Data = DataBuffer.CharData;     break;
                case sizeof(USHORT):
                        Data = DataBuffer.ShortData;    break;
                case sizeof(ULONG):
                        Data = DataBuffer.LongData;     break;
        }
        printf("Read from port %x returned %x\n", PortNumber, Data);
}
else {
        printf("Ioctl falhou com código %ld\n", GetLastError() );
}
```

```
if (!CloseHandle(hndFile)) {
        printf("Falha ao fechar o dispositivo.\n");
}
```

Exemplo: escrita em porta

```
BOOL        IoctlResult;
HANDLE      hndFile;            // Handle para porta gerado por CreateFile
typedef struct  _GENPORT_WRITE_INPUT {
        ULONG   PortNumber;         // Endereço do port
        Union       {       // Dado a ser enviado ao port
        ULONG   LongData;
        USHORT  ShortData;
        UCHAR   CharData;
        };
}   GENPORT_WRITE_INPUT;

GENPORT_WRITE_INPUT InputBuffer; // buffer de entrada p/ DeviceIoControl
LONG        IoctlCode;
ULONG       DataValue;          // Contém valor a ser escrito na porta
ULONG       DataLength;
ULONG       ReturnedLength; // Number of bytes returned in output buffer

hndFile = CreateFile(
                "\\\\.\\GpdDev", // Abre o arquivo Device
                GENERIC_WRITE,
                FILE_SHARE_WRITE,
                NULL,
                OPEN_EXISTING,
                0,
                NULL);

if (hndFile == INVALID_HANDLE_VALUE)  {       // O dispositivo foi aberto ?
        printf("Não consegui abrir dispositivo\n");
        exit(1);
}

IoctlCode = IOCTL_GPD_WRITE_PORT_UCHAR;
InputBuffer.PorNumber = 0x0300;
InputBuffer.CharData = (UCHAR)DataValue;
DataLength = offsetof(GENPORT_WRITE_INPUT, CharData) + sizeof(
InputBuffer.CharData);

IoctlResult = DeviceIoControl(
        hndFile,            // Handle para dispositivo
        IoctlCode,          // IO Control code para escrita
        &InputBuffer,       // Buffer para driver.  Contém porta & dados.
        DataLength,         // Comprimento do buffer em bytes
        NULL,               // Buffer de saída.   Não usado.
        0,                  // Comprimento do buffer em bytes.
```

```
        &ReturnedLength,   // Bytes colocados em outbuf.  Deve ser 0.
        NULL               // NULL = Síncrono
);

if (IoctlResult)   {// Verifica se sucesso
  printf(  "Escreveu no port  %x o dado %x\n",  InputBuffer.PortNumber,
  DataValue);
}
else  {
  printf( "Ioctl falhou com código %ld\n", GetLastError() );
}

if  (!CloseHandle(hndFile)) {
  printf("Falha ao fechar dispositivo \n");
}
```

Acesso direto ao espaço de I/O

Para conseguir acessar diretamente endereços mapeados no espaço de I/O, temos que entender como funciona o mecanismo de proteção do WNT.

A arquitetura 80x86 define quatro níveis ou castas de privilégios: 0 a 3. A CPU trabalha no nível de privilégio mais alto: 0 e as demais aplicações são distribuídas nos demais níveis dependendo de sua importância e dos recursos que utiliza do sistema. No caso do WNT apenas dois níveis de privilégio são utilizados. Um aplicação pode ser executada em nível 0 (kernel mode) ou em nível 3 (user mode). Aplicações do usuário são geralmente definidas para execução em modo usuário.

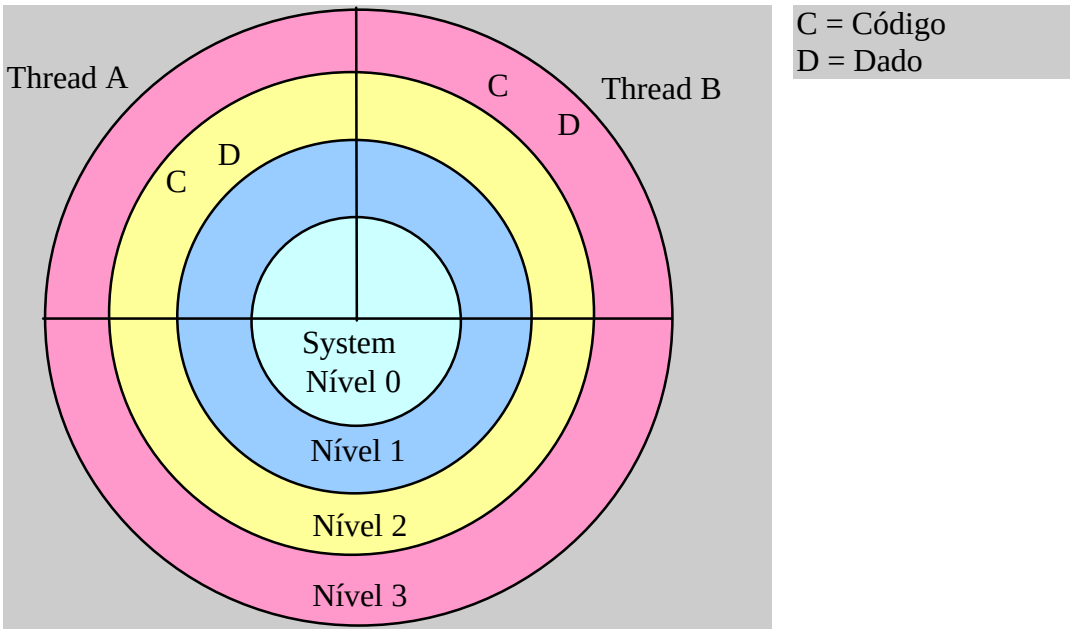


Figura 1 Níveis de privilégios na arquitetura Intel 80x86

O nível de privilégio corrente do processador (CPL) é armazenado nos dois bits menos significativos do registro CS (*code segment*). A arquitetura poderia definir que processos de nível de privilégio 0 teriam livre acesso ao sistema de I/O, mas ao invés de um mecanismo estático optou-se por um mecanismo dinâmico. A CPU define um *I/O privilege level* (IOPL) que é comparado com o valor do CPL para determinar se o processo pode acessar entradas e saídas. O IOPL é armazenado em dois bits do registro EFLAGS do processador. Se um processo possui um CPL maior que o IOPL, então não pode acessar portas diretamente e deve utilizar os mecanismos de proteção do sistema operacional para requisitar os serviços desejados. Logicamente programas em kernel mode (CPL=0) podem acessar I/O diretamente. No WNT o valor de IOPL é 0.

O segundo mecanismo de proteção determina para cada tarefa que portas de I/O ela pode acessar. Cada I/O port está mapeada em um bit de um array de bits. Se o bit é 0 a tarefa tem permissão, se é 1, ela não pode acessar o port. Na arquitetura 80x86 existem 65536 ports de 8 bits. A tabela de mapeamento denominada *I/O Permission BitMap* (IOPM) tem 8192(0x2000) posições.

A IOPM é armazenada no *Task State Segment* (TSS) na memória principal. O registro *Segment Selector,* que faz parte do *Task Register* (TR) do processador, contém o endereço de um descritor localizado na GDT (*Global Descriptor Table*) que por sua vez contém o endereço e o tamanho da tabela TSS. O conteúdo da posição 0x63 da TSS é o offset para a tabela IOPM.

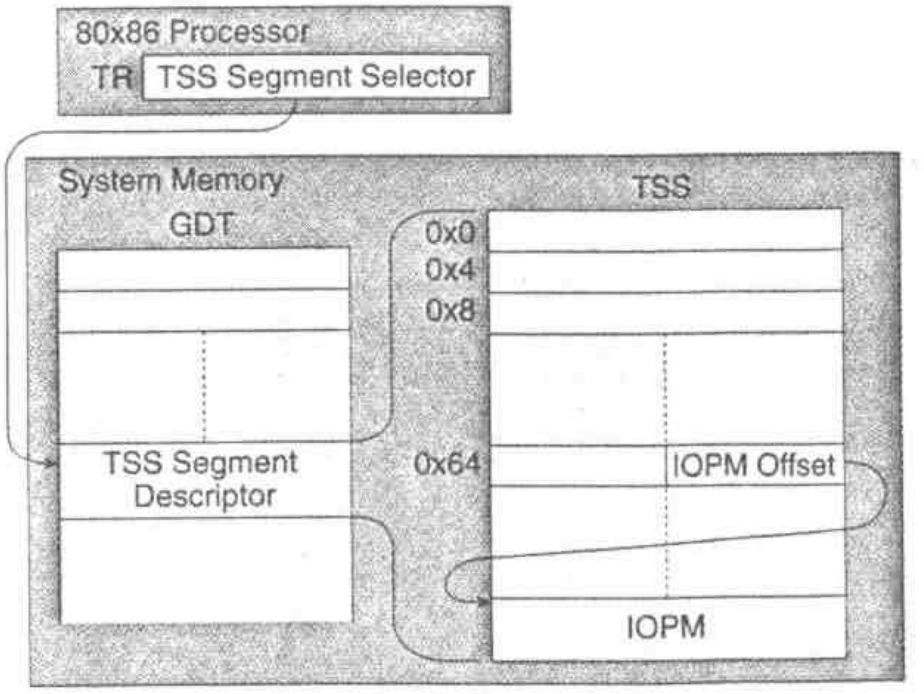


Figura 2: O TSS Segment Selector aponta para um descritor na GDT que define o tamanho e aposição da tabela TSS na memória.

Na arquitetura 80x86 cada task do sistema pode ter a sua própria TSS. No WNT a TR nunca é modificada e todos os processos usam a mesma cópia da IOPM.

No WNT a tabela IOPM está localizada após o fim da TSS. Isto proíbe os uso da IOPM por qualquer processo em user mode. Para usar esta tabela deveríamos ou modificar o offset da tabela IOPM para colocá-la dentro da TSS ou estender a TSS para que IOPM seja incluída.

Simplesmente estender a TSS para que uma IOPM toda zeros seja incluída não é uma grande idéia. Isto daria acesso ao espaço de I/O que fosse habilitado a qualquer aplicação, incluindo aplicações mal comportadas ou maliciosas e colocaria todo o sistema em risco. Nós devemos garantir o acesso livre a apenas um processo.

Usando um driver em modo kernel para garantir acesso a um único processo

Usando um debugger pode-se notar que existe um bloco de 0xFF que se estende da posição 0x88 até o final da TSS. Este deveria ser o local reservado para a tabela IOPM. Poder-se-ia mudar o offset para a IOPM para apontar para esta posição como indicado na figura 3.

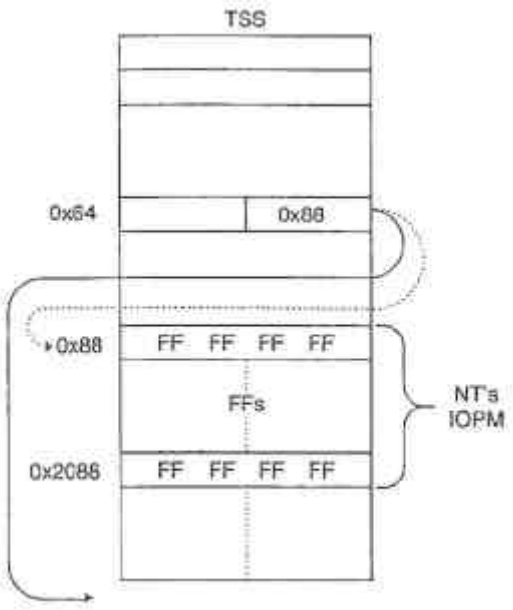


Figura 3: NT coloca IOPM no offset 0x88 na TSS

Modificar o offfset na posição 0x63 da TSS entretanto não resolve o problema porque o verdadeiro offset é carregado de uma posição contida na memória do processo toda vez que este ganha o controle.

Algumas rotinas em modo kernel são necessárias para nos ajudar a resolver este problema.

K e 3 8 6 S e t I o A c c e s s M a p

Copia uma tabela de bits de tamanho 0x200 para o offset 0x88 da TSS.

BOOL Ke386SetIoAccessMap(

int nFunction,	// Inteiro que deve ser feito igual a 1 0: enche IOPM com 0xFF.
LPVOID lpBuffer);	// Apontador para buffer contendo dados para realizar a operação

Ke386QueryIoAccessMap

Copia o IOPM corrente da TSS para o buffer passado como argumento.

BOOL Ke386QueryIoAccessMap(

int nFunction,	// Inteiro que deve ser feito igual a 1 0: retorna tabela cheia de 0xFF
LPVOID lpBuffer);	// Apontador para buffer que receberá os dados da operação

Ke386IoSetAccessProcess

Esta função habilita ou desabilita um processo de utilizar os ports. Se o parâmetro nAccessRights for 0, o offset para o IOPM será definido para além do final da TSS, desabilitando o uso da tabela. Se for feito igual a 1 o offset será definido para o início da tabela IOPM e os direitos de acesos de cada byte serão obedecidos.

BOOL Ke386IoSetAccessProcess(

PEPROCESS pCurrentProcess,	// Ponteiro para o processo corrente retornado por PEPROCESS PsGetCurrentProcess(void);
int nAccessRights);	// 1: garante acesso ao port 0: nega direito de acesso

Assim estas três rotinas de suporte de device drivers, não documentadas, oferecem um mecanismo que permite definir o direito de acesso de cada port e habilitar o uso do IO Permission Bitmap.

Exemplo: GiveIO.SYS: *Kernel mode device driver*

Este programa habilita o acesso deireto de todos os ports.

```
/*********************************************************************
 * Author:     Dale Roberts
 * Date:       8/30/95
 * Program:    GIVEIO.SYS
 * Compile:    Use DDK BUILD facility
 * Purpose:    Give direct port I/O access to a user mode process.
```

```
**************************************************************/

#include <ntddk.h>

/*  Nome do device driver */
#define DEVICE_NAME_STRING        L"giveio"

/*
 * Estrutura IOPM: um simples array de 0x2000 bytes
 * Cada byte mapeia 8 portas do espaço de endereçamanto de I/O de 8K
 * Posições. 0 habilita o uso do port por um processo em user mode. 1 veta o uso
 * do port.
*/
#define      IOPM_SIZE        0x2000
typedef UCHAR IOPM[IOPM_SIZE];

/*
 *  Esta variável irá manter uma array de zeros que será copiado
 * no IOPM real no TSS através da instrução Ke386SetIoAccessMap().
 * A memória será alocada no momento da carga do driver.
 */
IOPM *IOPM_local = 0;

/* Rotinas não documentadas do kernel */
void Ke386SetIoAccessMap(int, IOPM *);
void Ke386QueryIoAccessMap(int, IOPM *);
void Ke386IoSetAccessProcess(PEPROCESS, int);

/*************************************************************
  Libera quaisquer objetos já alocados
*************************************************************/
VOID GiveioUnload(IN PDRIVER_OBJECT DriverObject)
{
        WCHAR DOSNameBuffer[] =
        L"\\DosDevices\\" DEVICE_NAME_STRING;
        UNICODE_STRING uniDOSString;

        if (IOPM_local)
                MmFreeNonCachedMemory(IOPM_local, sizeof(IOPM));

        RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);
        IoDeleteSymbolicLink (&uniDOSString);
        IoDeleteDevice(DriverObject->DeviceObject);
}  /* GiveioUnload */

/*************************************************************
 * Define o IOPM (I/O permission map) do processo que chama a rotina para dar
 * a ele acesso completo ao espaço de I/O.  O IOPM_local[] só contém zeros e
 * portanto a IOPM conterá apenas zeros. Se OnFlag=1 o processo terá acesso
 * livre, se for 0 o direito de acesso será negado.
```

```
**************************************************************************/
VOID SetIOPermissionMap(int OnFlag)
{
        Ke386IoSetAccessProcess(PsGetCurrentProcess(), OnFlag); /* habilita */
        Ke386SetIoAccessMap(1, IOPM_local);  /* define tabela toda = 1 ou 0*/
}  /* SetIOPermissionMap */

void GiveIO(void)
{
        SetIOPermissionMap(1);
}  /* GiveIO */

/**************************************************************************
 * Handler de serviço para chamada CreateFile() em modo usuário.
 * Esta rotina é introduzida na tabela de chamada de funções do objeto driver
 * pela rotina DriveEntry(). Quando a rotina de modo usuário chamr CreateFile(),
 * esta rotina será chamada dentro do contexto da aplicação em modo usuário, mas
 * com o CPL =0  (Current Privilege Level do processador). Isto nos permite
 * realizar operações em modo kernel. GiveIO() é chamada para dar ao processo
 * qua a invoca direitos de acesso a I/O. Tudo que a aplicação em modo usuário
 * precisa fazer para obter acesso a I/O é abrir o device com CreateFile()
 * Nenhuma outra operação é requerida.
**************************************************************************/

NTSTATUS GiveioCreateDispatch(
        IN      PDEVICE_OBJECT    DeviceObject,
        IN      PIRP          Irp)
{
        GiveIO();           // de acesso livre a I/O ao processo que chama a função

        Irp->IoStatus.Information = 0;
        Irp->IoStatus.Status = STATUS_SUCCESS;
        IoCompleteRequest(Irp, IO_NO_INCREMENT);
        return STATUS_SUCCESS;
}  /* GiveioCreateDispatch */

/**************************************************************************
 * Rotina Driver Entry
 * Esta rotina é chamada apenas uma vez quando o driver é carregado na memória
 * Ela aloca tudo que é necessário à operação do driver.
 * No nosso caso ela aloca memória para o array IOPM e cria um device
 * Que pode ser aberto por aplicações em modo usuário.
 * Ela também cria um link simbólico com o device driver.
 * Isto permite a uma aplicação em modo usuário acessar nosso drive
 * Usando a notação \\.\giveio.
**************************************************************************/
NTSTATUS DriverEntry(
        IN PDRIVER_OBJECT       DriverObject,
        IN PUNICODE_STRING      RegistryPath)
{
```

```
    PDEVICE_OBJECT deviceObject;
    NTSTATUS status;
    WCHAR NameBuffer[] = L"\\Device\\" DEVICE_NAME_STRING;
    WCHAR DOSNameBuffer[] = L\\DosDevices\\
    DEVICE_NAME_STRING;
    UNICODE_STRING uniNameString, uniDOSString;

    //  Alloca um buffer para o IOPM local e o zera.
    IOPM_local = MmAllocateNonCachedMemory(sizeof(IOPM));
    if (IOPM_local == 0)
        return STATUS_INSUFFICIENT_RESOURCES;
    RtlZeroMemory(IOPM_local, sizeof(IOPM));

    //  Define nome do device driver e do objeto device
    RtlInitUnicodeString(&uniNameString, NameBuffer);
    RtlInitUnicodeString(&uniDOSString, DOSNameBuffer);

    status = IoCreateDevice(DriverObject, 0,
                            &uniNameString,
                            FILE_DEVICE_UNKNOWN,
                            0, FALSE, &deviceObject);

    if (!NT_SUCCESS(status))
        return status;

    status = IoCreateSymbolicLink (&uniDOSString, &uniNameString);

    if (!NT_SUCCESS(status))
        return status;

    // Inicializa o objeto Driver com os pontos de entrada do driver
    // tudo de que precisamos são as operações Create e Unload.
    DriverObject->MajorFunction[IRP_MJ_CREATE] =
    GiveioCreateDispatch;
    DriverObject->DriverUnload = GiveioUnload;
    return STATUS_SUCCESS;
} // DriverEntry
```

Exemplo Aplicação de teste TESTIO

Este programa usa acesso direto aos endereços de I/O para utilizar o speaker do PC.

```
/*********************************************************************
Author:     Dale Roberts
Date:       8/30/95
Program:    TSTIO.EXE
Purpose:    Testa o device driver GIVEIO realizando I/O direto para acessar o
            auto falante do PC
*********************************************************************/
```

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#include <conio.h>

typedef struct {
        short int pitch;
        short int duration;
} NOTE;

// Tabela de notas musicais: 5 notas de contatos imediatos do 3o grau
// O primeiro número corresponde ao offset em relação ao lá médio
NOTE notes[] = {{14, 500}, {16, 500}, {12, 500}, {0, 500}, {7, 1000}};
//                      si4             ré4     lá4         lá3         mi3
/*****************************************************************
Toca uma nota dada a sua freqüência em Hertz.
O alto falante é controlado por um chip temporizador Intel 8253/8254 no endereço
de I/O 0x40-0x43.
*****************************************************************/
void setfreq(int hz)
{
        hz = 1193180 / hz;              // clocked at 1.19MHz
        _outp(0x43, 0xb6);              // timer 2, square wave
        _outp(0x42, hz);                // escreve high byte
        _outp(0x42, hz >> 8);           //  escreve low byte
}  // setfreq

/*****************************************************************
Passa uma nota em passos referente ao lá médio: 440 Hz.
A escala dodecafônica é exponencial: f(n+1)=f(n)*2^(1/12)
```

DO1	261
DO1_S	277
RE1	293
RE1_S	311
MI1	329
FA1	349
FA 1_S	369
SOL1	391
SOL1_S	415
LA1	440
LA1_S	466
SI1	493

```
O registro de controle do alto falante está na porta 0x61.
Ativando os dois bits menos significativos habilita o temporizador 2 do
timer 8253/8254 e liga o alto falante.
*****************************************************************/
void playnote(NOTE note)
{
        _outp(0x61, _inp(0x61) | 0x03);                 // inicia som
        setfreq((int)(400 * pow(2, note.pitch / 12.0)));  // calcula freq da nota
        Sleep(note.duration);                           // espera duração
        _outp(0x61, _inp(0x61) & ~0x03);                // interrompe som
```

```
}

/*****************************************************************
Abre e fecha o device GIVEIO. Isto pode nos dar acesso direto ao espaço de IO.
Teste-o e ouça o resultado.
*****************************************************************/
int main()
{
	int i;
#ifdef WNT
	HANDLE h;

	h = CreateFile("\\\\.\\giveio", GENERIC_READ, 0, NULL,
	OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

	if(h == INVALID_HANDLE_VALUE) {
		printf("Não consegui acessar o dispositivo giveio\n");
		return -1;
	}
	CloseHandle(h);
#endif

	for (i=0; i < sizeof(notes)/sizeof(int); ++i)
		playnote(notes[i]);

	return 0;
}	// main
```

Uma vez que a aplicação *user mode* ganhou acesso aos ports o *device drive* pode ser descarregado.

Velocidade de execução

Na verdade, a velocidade de execução das instruções *_inp* e *_oup*, usando o artificio descrito, é muito menor do que a velocidade usando o acesso direto proporcionado pelo DOS ou Windows 95 (modo real, não protegido). O mecanismo de proteção do WNT irá checar a cada instrução de entrada e saída em que CPL>IOPL, os bits de habilitação no IOPM.
O número de ciclos de clock é cerca de três vezes maior (30 ciclos) que o de uma aplicação executando em *kernel mode* (10 ciclos). Entretanto este valor é cerca de 200 vezes menor do que o de uma aplicação que chama as rotinas de acesso do Win32 para realizar I/O (6000 a 12000 ciclos).

O I/O direto é fundamental para aplicações que necessitam acessar diretamente a memória de vídeo como por exemplo vídeo games.

Outros programas:

UserPort

O site http://www.lvr.com/files/userport.zip permite realizar o download de um drive de porta paralela denominado UserPort, baseado no mesmo artigo que serviu de base a esta aula. Além dos programas fontes são fornecidos dois arquivos: um device drive denominado UserPort.SYS e um programa executável: UserPort.EXE.

A grande vantagem é facilidade de instalação e uso do driver.
Após fazer o download dos programas, copie o device driver UserPort.SYS para o diretório Windows\System32\Drivers.
Use agora o programa UserPort.exe para habilitar somente os ports desejados. A partir de agora todos os programas no micro poderão acessar os ports livres.

Rode por exemplo o programa TestIo recompilado para retirar as instruções de instalação do drive. Habilite os endereços no range 40-70. Agora você poderá executar livremente o programa.

Os fontes do programa UserPort são dados no CD que acompanha o curso.

Delphi

A Scientific Software disponibiliza dum componente com a mesma funcionalidade para uso no Delphi, no endereço: http://www.sstnet.com/DownLoad. O DriverLink Port I/O possui um programa de set up para facilitar sua instalação.

Porta Paralela

No PC as portas paralelas ocupam uma das seguintes faixas de endereços:

```
3BCh, 3BDh, 3BEh
378h, 379h, 37Ah
278h, 279h, 27Ah
```

A segunda faixa é hoje a mais comum.

Para verificar a faixa de enderços de sua porta no Windows95 use o ControlPanel e selecione: System->DeviceManager.

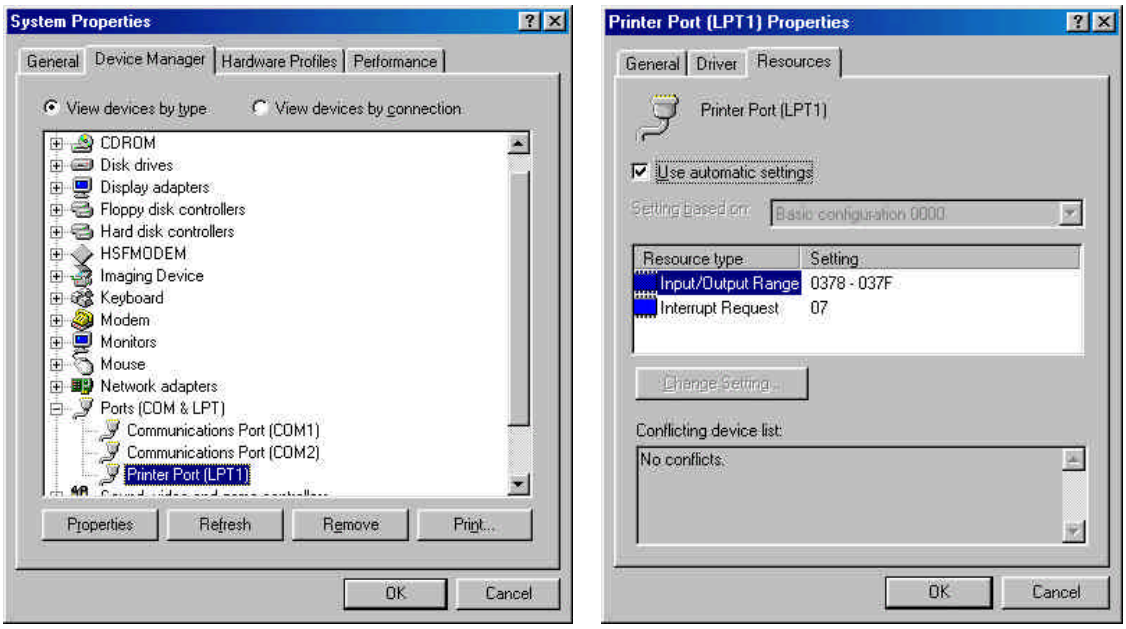


Figura 4: Determinando o endereços da porta paralela

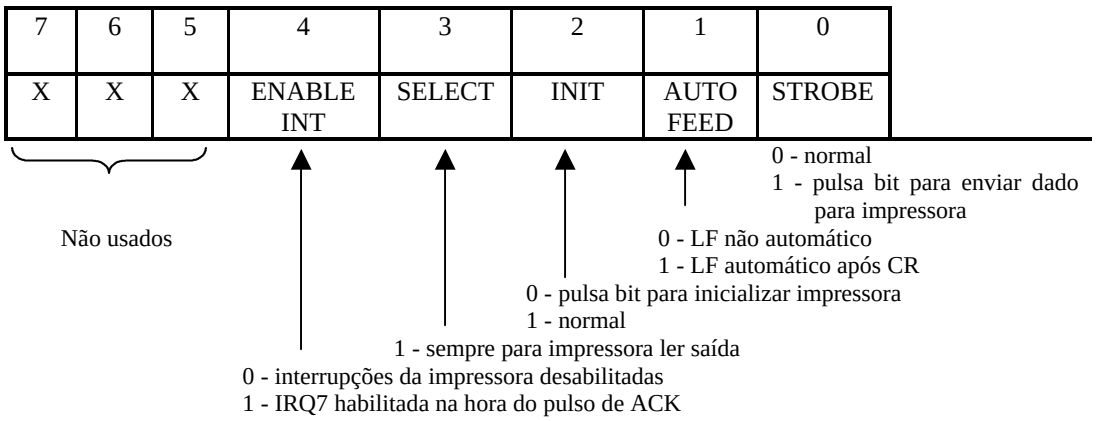
O primeiro endereço corresponde ao endereço da porta de dados (8 bits), o segundo é o endereço da porta de status e o terceiro corresponde à porta de comandos.

Data register	0x378
Status register	0x379
Command register	0x37A

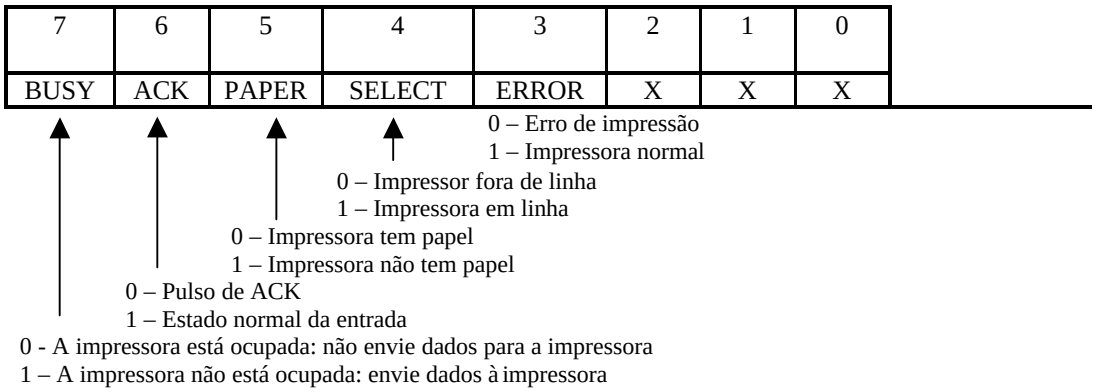
Mapa do conector:

Número do pino	Nome do sinal	Direção
2	Bit de dado 0	Para impressora
3	Bit de dado 1	Para impressora
4	Bit de dado 2	Para impressora
5	Bit de dado 3	Para impressora
6	Bit de dado 4	Para impressora
7	Bit de dado 5	Para impressora
8	Bit de dado 6	Para impressora
9	Bit de dado 7	Para impressora
1	Strobe	Para impressora
13	Select Input	Para impressora
14	Auto Feed	Para impressora
16	Initialize printer	Para impressora
10	Acknowledge	Para computador
11	Busy	Para computador
12	Out of Paper	Para computador
13	Select	Para computador
15	Error	Para computador

Mapa de bits de controle de saída:



Mapa de bits de controle de entrada:



O funcionamento da interface básica de impressão é simples:

A impressora não irá aceitar nenhum dados até a linha *select* estar no estado correto. A linha *inicializa impressora* é usada para inicializar a impressora quando o sistema é alimentado pela primeira vez. A linha deve ser ativada durante 50μs.
Os dados a serem enviados são colocados nas oito linhas de dados. Então a linha de *strobe* é pulsada. A impressora processa o dado e envia um pulso confirmação na linha de *acknowledge*. Quando este sinal é recebido, o computador pode enviar um outro caracter. Ao invés de esperar o pulso de ACK, o computador pode verificar a linha de BUSY. O computador pode enviar caracteres para a impressora, enquanto esta linha não estiver ativada. O bit *out of paper* diz ao computador que impressora está sem papel.

A velocidade máxima alcança com a interface padrão é de cerca de 150kbytes/s.
Um novo padrão de comunicação bidirecional usando o canal paralelo foi aprovado em 1994 que permite velocidades até 1Mbytes/s. Este padrão é o IEEE 1284 ("*Standard Signaling Method for a Bi-directional Parallel Peripheral Interface for Personal Computers*") adotado desde então.

O canal paralelo pode ser usado par muitas outras finalidades além de propiciar a comunicação com impressoras como já mencionado nesta aula.

Exercícios:

1) Escreva um programa para o Windows95 para enviar um array de dados de oito bits pela porta paralela do PC.

2) Rescreva o programa anterior para o Windows NT usando o device drive Giveio.sys.

Bibliografia

[Hunt 94] Dale Roberts, Direct Port I/O and Windows NT, Dr. Dobbs Journal, May 1996, pp 14..24, 76..78.

[Microsoft 93] PortIO: um exemplo de WNT Device Driver DDK

[Axelson 96] Jan Axelson, Parallel Port Complete, Programming, Interfacing & using the PC's parallel printer port, chapiter 1, Lakeview Research, 1996

[Willen 83] David C. Willen and Jeffrey I. Krantz, 8088 Assembler Language Programming, The IBM PC, Howard W. Sams & Co., Inc., 1st, edition,1983

Sites a serem visitados

www.lvr.com/parport.htm	O melhor site sobre portas paralelas
http://www.sstnet.com/DownLoad/dnload.htm#item3	Scientific Software Tools Inc.
http://www.beyondlogic.org/	Artigos sobre portas seriais e paralelas
http://www.fapo.com/ieee1284.htm	IEEE1284