

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Интерфейсы и устройства вычислительных машин

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ

на тему

Конфигурационное пространство PCI

Выполнил
студент гр. 250541

В.Ю. Бобрик

Проверил
ст. преподаватель каф. ЭВМ

Д.В. Куприянова

Минск 2025

СОДЕРЖАНИЕ

1 Цель работы.....	3
2 Исходные данные к работе.....	3
3 Теоретические сведения.....	3
4 Выполнение работы.....	4
5 Вывод.....	15

1 Цель работы

Написать программу, выводящую список всех устройств, подключенных к шине PCI.

2 Исходные данные к работе

Для написания программы используется язык C/C++ и операционная система Windows 10.

Необходимо вывести список всех устройств, подключенных к шине PCI, с их характеристиками (DeviceID и VendorID, состоящие из 4-х символов) в виде таблицы.

Подключение к шине производить с помощью готовых библиотек запрещено. Подключение к шине реализовать с применением портов ввода-вывода.

3 Теоретические сведения

Шина PCI является синхронным параллельным электрическим интерфейсом с общей средой передачи данных.

Шина состоит из мультиплексированных линий передачи адреса и данных (разделение по времени) и линий различных управляющих сигналов. Шина PCI разводится внутри микросхем или на печатной плате (обычно материнской). Устройства могут быть выполнены в виде микросхем, плат расширения (например, ATX), модулей Mini PCI, Compact PCI, PXI и т.д.

Хост – источник команд и основной потребитель данных; в случае компьютера x86 это системное ядро – процессор и системная память. Хост подключен через главный мост (Host bridge), который является устройством PCI и действует от имени хоста. Хост занимается также распределением ресурсов и конфигурированием всех устройств PCI.

Мосты играют роль арбитров, обрабатывая запросы от устройств на доступ к шине и отслеживая соблюдение протокола обмена.

В общем случае шина PCI имеет топологию многоуровневая шина. К первичной шине могут подключаться устройства – мосты, управляющие вторичными шинами, и так далее. Помимо упомянутых мостов PCI-PCI, к шине подключаются мосты для связи с другими шинами, в их задачи входит трансляция транзакций, поступающих по шине PCI, к устройствам, которые подключены к другойшине.

В рамках транзакции определены два объекта: инициатор обмена (Initiator) и целевое устройство (Target). В рамках одной физической шины в конкретный момент может происходить только одна транзакция. Если физических шин несколько, то транзакции на них могут выполняться одновременно (Peer Concurrency), если пути прохождения данных не пересекаются.

Устройство, ставшее инициатором обмена и взявшее на себя временное управление шиной, называется Bus Master. Наличие этой функции не обязательно для устройств. Решение о передаче управления шиной принимает арбитр данной шины.

Механизм Bus Mastering фактически заменяет механизм с выделенным контроллером DMA: каждое устройство самостоятельно осуществляет доступ к системной памяти, выполняя все функции контроллера DMA.

Существует 4 механизма доступа к устройствам со стороны хоста или других устройств:

- обращение к области памяти или портам, выделенным устройству;
- обращение к конфигурационным регистрам (в конфигурационном адресном пространстве);
- широковещательные сообщения ко всем устройствам шины;
- механизм обмена сообщениями.

Для подачи сигналов хосту устройства применяют механизм прерываний:

- маскируемые (INTx или MSI - Message Signaled Interrupt);
- немаскируемые (NMI - NonMaskable Interrupt);
- системные (SMI - System Management Interrupt).

Когда устройства сконфигурированы, они адресуются через диапазоны пространства памяти или портов на основе анализа адреса, передаваемого в начале транзакции. В противном случае требуется механизм конфигурационного доступа.

4 Выполнение работы

Исходный код драйвера, выполняющего поставленную задачу, приведен ниже.

```
Driver.c
001 #include <wdm.h>
002 #include <ntstrsafe.h>
003
004 #define PCI_CONFIG_ADDRESS 0xCF8
005 #define PCI_CONFIG_DATA    0xCFC
006 #define DEVICE_NAME L"\Device\PCIScanner"
007 #define SYMBOLIC_NAME L"\DosDevices\PCIScanner"
008
009 #define IOCTL_PCI_GET_DEVICES CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800,
METHOD_BUFFERED, FILE_ANY_ACCESS)
010
011 typedef struct _PCI_DEVICE_INFO {
012     UCHAR Bus;
013     UCHAR Device;
014     UCHAR Function;
015     USHORT VendorID;
016     USHORT DeviceID;
```

```

017     UCHAR BaseClass;
018     UCHAR SubClass;
019     UCHAR Revision;
020     char Description[64];
021 } PCI_DEVICE_INFO, * PPCI_DEVICE_INFO;
022
023 typedef struct _PCI_DEVICE_LIST {
024     ULONG NumberOfDevices;
025     PCI_DEVICE_INFO Devices[32];
026 } PCI_DEVICE_LIST, * PPCI_DEVICE_LIST;
027
028 DRIVER_UNLOAD UnloadDriver;
029 DRIVER_DISPATCH DispatchCreateClose;
030 DRIVER_DISPATCH DispatchDeviceControl;
031
032 // Определение типа устройства по Class/Subclass
033 const char* GetDeviceType(UCHAR base_class, UCHAR sub_class) {
034     UNREFERENCED_PARAMETER(base_class);
035     UNREFERENCED_PARAMETER(sub_class);
036
037     switch (base_class) {
038         case 0x00: return "Pre-2.0 Device";
039         case 0x01:
040             switch (sub_class) {
041                 case 0x00: return "SCSI Controller";
042                 case 0x01: return "IDE Controller";
043                 case 0x02: return "Floppy Controller";
044                 case 0x03: return "IPI Controller";
045                 case 0x04: return "RAID Controller";
046                 case 0x05: return "ATA Controller";
047                 case 0x06: return "SATA Controller";
048                 case 0x80: return "Other Mass Storage";
049                 default: return "Mass Storage Controller";
050             }
051         case 0x02:
052             switch (sub_class) {
053                 case 0x00: return "Ethernet Controller";
054                 case 0x01: return "Token Ring Controller";
055                 case 0x02: return "FDDI Controller";
056                 case 0x03: return "ATM Controller";
057                 case 0x04: return "ISDN Controller";
058                 case 0x80: return "Other Network Controller";
059                 default: return "Network Controller";
060             }
061         case 0x03:
062             switch (sub_class) {
063                 case 0x00: return "VGA Compatible Controller";
064                 case 0x01: return "XGA Controller";
065                 case 0x02: return "3D Controller";
066                 case 0x80: return "Other Display Controller";
067                 default: return "Display Controller";
068             }
069         case 0x06:
070             switch (sub_class) {
071                 case 0x00: return "Host Bridge";

```

```

072         case 0x01: return "ISA Bridge";
073         case 0x02: return "EISA Bridge";
074         case 0x03: return "MCA Bridge";
075         case 0x04: return "PCI-to-PCI Bridge";
076         case 0x05: return "PCMCIA Bridge";
077         case 0x06: return "NuBus Bridge";
078         case 0x07: return "CardBus Bridge";
079         case 0x08: return "RACEway Bridge";
080         case 0x80: return "Other Bridge";
081         default: return "Bridge Device";
082     }
083     case 0x0C:
084     switch (sub_class) {
085         case 0x00: return "Serial Controller";
086         case 0x01: return "Parallel Controller";
087         case 0x02: return "Multiport Serial Controller";
088         case 0x03: return "Modem";
089         case 0x80: return "Other Communications";
090         default: return "Communications Controller";
091     }
092     default: return "Unknown Device";
093 }
094 }
095
096 // Определение вендора по VendorID
097 const char* GetVendorName(USHORT vendor_id) {
098     UNREFERENCED_PARAMETER(vendor_id);
099
100    switch (vendor_id) {
101        case 0x8086: return "Intel";
102        case 0x10DE: return "NVIDIA";
103        case 0x1002: return "AMD";
104        case 0x1414: return "Microsoft";
105        case 0x5333: return "S3";
106        case 0x1011: return "Digital Equipment";
107        case 0x10EC: return "Realtek";
108        case 0x1969: return "Atheros";
109        default: return "Unknown Vendor";
110    }
111 }
112
113 NTSTATUS ScanPciDevices(PPCI_DEVICE_LIST deviceList) {
114     UCHAR bus, device, function;
115     ULONG devices_found = 0;
116
117     for (bus = 0; bus < 2; bus++) {
118         for (device = 0; device < 32; device++) {
119             for (function = 0; function < 8; function++) {
120                 ULONG address = (1 << 31) | (bus << 16) | (device
121 << 11) | (function << 8);
122                 WRITE_PORT_ULONG((PULONG)PCI_CONFIG_ADDRESS, address);
123                 ULONG vendor_device =
124                 READ_PORT_ULONG((PULONG)PCI_CONFIG_DATA);
125
126                 USHORT vendor_id = (USHORT)(vendor_device & 0xFFFF);

```

```

125     USHORT device_id = (USHORT)((vendor_device >> 16) & 0xFFFF);
126
127         if (vendor_id != 0xFFFF) {
128             // Читаем Class Code и Revision
129             WRITE_PORT ULONG((PULONG)PCI_CONFIG_ADDRESS, address | 0x08);
130             ULONG class_rev = READ_PORT ULONG((PULONG)PCI_CONFIG_DATA);
131             UCHAR revision = (UCHAR)(class_rev & 0xFF);
132             UCHAR sub_class = (UCHAR)((class_rev >> 16) & 0xFF);
133             UCHAR base_class = (UCHAR)((class_rev >> 24) & 0xFF);
134
135             // Сохраняем информацию об устройстве
136             PPCI_DEVICE_INFO devInfo = &deviceList->Devices[devices_found];
137             devInfo->Bus = bus;
138             devInfo->Device = device;
139             devInfo->Function = function;
140             devInfo->VendorID = vendor_id;
141             devInfo->DeviceID = device_id;
142             devInfo->BaseClass = base_class;
143             devInfo->SubClass = sub_class;
144             devInfo->Revision = revision;
145
146             // Формируем описание
147             const char* vendor_name = GetVendorName(vendor_id);
148             const char* device_type = GetDeviceType(base_class, sub_class);
149
150             // Копируем информацию
151             NTSTATUS status;
152             status = RtlStringCbPrintfA(devInfo-
>Description, sizeof(devInfo->Description),
153                                         "%s %s", vendor_name, device_type);
154
155             // Запасной вариант копирования
156             if (!NT_SUCCESS(status)) {
157                 RtlStringCbCopyA(devInfo->Description,
158                                 sizeof(devInfo->Description), "Unknown Device");
159             }
160             devices_found++;
161
162             if (devices_found >= 32) {
163                 deviceList->NumberOfDevices = devices_found;
164                 return STATUS_SUCCESS;
165             }
166         }
167     }
168 }
169 }
170
171     deviceList->NumberOfDevices = devices_found;
172     return STATUS_SUCCESS;
173 }
174
175 NTSTATUS DispatchCreateClose(PDEVICE_OBJECT DeviceObject, PIRP
Irp) {
176     UNREFERENCED_PARAMETER(DeviceObject);

```

```

177     Irp->IoStatus.Status = STATUS_SUCCESS;
178     Irp->IoStatus.Information = 0;
179     IoCompleteRequest(Irp, IO_NO_INCREMENT);
180     return STATUS_SUCCESS;
181 }
183
184 NTSTATUS DispatchDeviceControl(PDEVICE_OBJECT DeviceObject, PIRP
Irp) {
185     UNREFERENCED_PARAMETER(DeviceObject);
186
187 PIO_STACK_LOCATION irpStack = IoGetCurrentIrpStackLocation(Irp);
188     NTSTATUS status = STATUS_SUCCESS;
189     ULONG infoLength = 0;
190
191     switch (irpStack->Parameters.DeviceIoControl.IoControlCode) {
192         case IOCTL_PCI_GET_DEVICES: {
193             if (irpStack-
>Parameters.DeviceIoControl.OutputBufferLength >=
sizeof(PCI_DEVICE_LIST)) {
194                 PPCI_DEVICE_LIST deviceList = (PPCI_DEVICE_LIST)Irp-
>AssociatedIrp.SystemBuffer;
195                 status = ScanPciDevices(deviceList);
196                 infoLength = sizeof(PCI_DEVICE_LIST);
197             }
198             else {
199                 status = STATUS_BUFFER_TOO_SMALL;
200             }
201             break;
202         }
203         default:
204             status = STATUS_INVALID_DEVICE_REQUEST;
205             break;
206     }
207
208     Irp->IoStatus.Status = status;
209     Irp->IoStatus.Information = infoLength;
210     IoCompleteRequest(Irp, IO_NO_INCREMENT);
211     return status;
212 }
213
214 void UnloadDriver(PDRIVER_OBJECT DriverObject) {
215     UNICODE_STRING symbolicName;
216     RtlInitUnicodeString(&symbolicName, SYMBOLIC_NAME);
217     IoDeleteSymbolicLink(&symbolicName);
218
219     if (DriverObject->DeviceObject) {
220         IoDeleteDevice(DriverObject->DeviceObject);
221     }
222
223     DbgPrint("PCISCAN: Driver Unloaded\n");
224 }
225
226 NTSTATUS DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING
RegistryPath) {

```

```

227     UNREFERENCED_PARAMETER(RegistryPath);
228
229     NTSTATUS status;
230     PDEVICE_OBJECT deviceObject = NULL;
231     UNICODE_STRING deviceName, symbolicName;
232
233     // Инициализируем строки
234     RtlInitUnicodeString(&deviceName, DEVICE_NAME);
235     RtlInitUnicodeString(&symbolicName, SYMBOLIC_NAME);
236
237     // Создаем устройство
238     status = IoCreateDevice(DriverObject, 0, &deviceName,
239                             FILE_DEVICE_UNKNOWN,
240                             FILE_DEVICE_SECURE_OPEN, FALSE, &deviceObject);
241
242     if (!NT_SUCCESS(status)) {
243         return status;
244     }
245
246     // Создаем символьическую ссылку
247     status = IoCreateSymbolicLink(&symbolicName, &deviceName);
248     if (!NT_SUCCESS(status)) {
249         IoDeleteDevice(deviceObject);
250         return status;
251     }
252
253     // Настраиваем функции драйвера
254     DriverObject->DriverUnload = UnloadDriver;
255     DriverObject->MajorFunction[IRP_MJ_CREATE] = DispatchCreateClose;
256     DriverObject->MajorFunction[IRP_MJ_CLOSE] = DispatchCreateClose;
257     DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
258         DispatchDeviceControl;
259
260     DbgPrint("PCISCAN: Driver loaded successfully\n");
261 }
```

Исходный код консольной программы приведен ниже.

```

main.cpp
000 #define _CRT_SECURE_NO_WARNINGS
001 #include "app.h"
002
003 int main() {
004     Application app;
005     return app.Run();
006 }

app.h
000 #pragma once
001 #include <windows.h>
002 #include <iostream>
003
```

```

004 class Application {
005 public:
006     int Run();
007
008 private:
009     void SetupConsole();
010    void ShowError(DWORD errorCode);
011    void WaitForExit();
012};

app.cpp
000 #include "app.h"
001 #include "pci_scanner.h"
002 #include "console_formatter.h"
003
004 int Application::Run() {
005     SetupConsole();
006
007     Console_Formatter::PrintHeader();
008
009     try {
010         PCI_Scanner_App scanner;
011
012         std::cout << "Initializing PCI scanner... ";
013         if (!scanner.Initialize()) {
014             DWORD error = GetLastError();
015             std::cout << "FAILED\n\n";
016             ShowError(error);
017             WaitForExit();
018             return 1;
019         }
020         std::cout << "SUCCESS\n\n";
021
022         // Сканирование
023         std::cout << "Scanning PCI bus... ";
024         auto devices = scanner.Scan();
025         std::cout << "COMPLETED\n\n";
026
027         // Вывод результатов
028         Console_Formatter::PrintDevices(devices);
029         Console_Formatter::PrintStatistics(devices);
030
031         std::cout << "\nOperation completed successfully!\n";
032
033     }
034     catch (const std::exception& ex) {
035         std::cerr << "\nError: " << ex.what() << "\n";
036         WaitForExit();
037         return 1;
038     }
039
040     WaitForExit();
041     return 0;
042 }
043

```

```

044 void Application::SetupConsole() {
045     SetConsoleOutputCP(CP_UTF8);
046     SetConsoleCP(CP_UTF8);
047     SetConsoleTitleW(L"PCI Device Scanner - C++");
048 }
049
050 void Application::ShowError(DWORD errorCode) {
051     std::cerr << "Cannot access PCI scanner driver.\n";
052     std::cerr << "Error code: " << errorCode << "\n\n";
053     std::cerr << "Some help:\n";
054     std::cerr << "1. sc create PCIScanner binPath=
\"C:\\\\path\\\\pci_scanner.sys\" type= kernel\n";
055     std::cerr << "2. sc start PCIScanner\n";
056     std::cerr << "3. You have administrator privileges\n";
057 }
058
059 void Application::WaitForExit() {
060     std::cout << "\nPress Enter to exit...";
061
062     char buffer[100];
063     if (std::cin.getline(buffer, sizeof(buffer))) {
064     }
065
066     std::cin.clear();
067     std::cin.ignore(10000, '\n');
068     std::cin.get();
069 }

console_formatter.h
000 #pragma once
001 #include <iostream>
002 #include <iomanip>
003 #include <vector>
004 #include <map>
005 #include "pci_device_info.h"
006
007 class Console_Formatter {
008 public:
009     static void PrintHeader();
010     static void PrintDevices(const std::vector<PCI_DEVICE_INFO>&
devices);
011     static void PrintStatistics(const
std::vector<PCI_DEVICE_INFO>& devices);
012
013 private:
014     static void PrintTableRow(const std::vector<std::string>&
columns, const int widths[]);
015     static void PrintSeparator(int length);
016 };

console_formatter.cpp
000 #include "console_formatter.h"
001
002 void Console_Formatter::PrintHeader() {
003     std::cout << "PCI Device Scanner\n";

```

```

004     std::cout << "-----\n\n";
005 }
006
007 void Console_Formatter::PrintDevices(const
008 std::vector<PCI_DEVICE_INFO>& devices) {
009     if (devices.empty()) {
010         std::cout << "No PCI devices found.\n";
011         return;
012     }
013     // Таблица с выравниванием
014     constexpr int col_widths[] = { 8, 12, 8, 10, 40 };
015
016     PrintTableRow({ "Addr", "Vendor:Device", "Class", "Rev",
017 "Description" }, col_widths);
018     PrintSeparator(80);
019
020     for (const auto& device : devices) {
021         PrintTableRow({
022             device.GetLocation(),
023             device.GetVendorDeviceID(),
024             device.GetClassCodes(),
025             std::format("{:02X}", device.Revision),
026             device.Description
027         }, col_widths);
028     }
029
030 void Console_Formatter::PrintStatistics(const
031 std::vector<PCI_DEVICE_INFO>& devices) {
032     std::cout << "\nScan Statistics:\n";
033     std::cout << "-----\n";
034     std::cout << "Total devices: " << devices.size() << "\n\n";
035
036     // Группировка по классам
037     std::map<UCHAR, int> classCount;
038     std::map<USHORT, int> vendorCount;
039
040     for (const auto& device : devices) {
041         vendorCount[device.VendorID]++;
042     }
043
044     std::cout << "Devices by class:\n";
045     for (const auto& [classCode, count] : classCount) {
046         std::cout << " Class " << std::hex <<
047             static_cast<int>(classCode)
048             << std::dec << ":" << count << " devices\n";
049
050     std::cout << "\nDevices by vendor:\n";
051     for (const auto& [vendorID, count] : vendorCount) {
052         std::cout << " Vendor " << std::hex << vendorID
053             << std::dec << ":" << count << " devices\n";
054 }

```

```

055
056 void Console_Formatter::PrintTableRow(const
057     std::vector<std::string>& columns, const int widths[]) {
058     for (size_t i = 0; i < columns.size(); ++i) {
059         std::cout << std::left << std::setw(widths[i]) <<
060             columns[i];
061     }
062
063 void Console_Formatter::PrintSeparator(int length) {
064     std::cout << std::string(length, '-') << "\n";
065 }

pci_device.info.h
000 #pragma once
001 #include <windows.h>
002 #include <string>
003 #include <format>
004
005 #define IOCTL_PCI_GET_DEVICES CTL_CODE(FILE_DEVICE_UNKNOWN, 0x800,
METHOD_BUFFERED, FILE_ANY_ACCESS)
006
007 struct PCI_DEVICE_INFO {
008     UCHAR Bus;
009     UCHAR Device;
010    UCHAR Function;
011    USHORT VendorID;
012    USHORT DeviceID;
013    UCHAR BaseClass;
014    UCHAR SubClass;
015    UCHAR Revision;
016    char Description[64];
017
018    std::string GetLocation() const {
019        return std::format("{:02X}:{:02X}.{:X}", Bus, Device, Function);
020    }
021
022    std::string GetVendorDeviceID() const {
023        return std::format("{:04X}:{:04X}", VendorID, DeviceID);
024    }
025
026    std::string GetClassCodes() const {
027        return std::format("{:02X}:{:02X}", BaseClass, SubClass);
028    }
029 };
030
031 struct PCI_DEVICE_LIST {
032     ULONG NumberOfDevices;
033     PCI_DEVICE_INFO Devices[32];
034 };

pci_scanner.h
000 #pragma once
001 #include <windows.h>

```

```

002 #include <vector>
003 #include <stdexcept>
004 #include "pci_device_info.h"
005
006 class PCI_Scanner_App {
007 private:
008     HANDLE m_hDevice{ nullptr };
009
010 public:
011     PCI_Scanner_App() = default;
012     ~PCI_Scanner_App();
013
014     PCI_Scanner_App(const PCI_Scanner_App&) = delete;
015     PCI_Scanner_App& operator=(const PCI_Scanner_App&) = delete;
016
017     bool Initialize();
018     void Shutdown();
019     std::vector<PCI_DEVICE_INFO> Scan();
020     bool IsOpen() const;
021
022 private:
023     bool Open();
024     void Close();
025 };

```

```

pci_scanner.cpp
000 #include "pci_scanner.h"
001
002 PCI_Scanner_App::~PCI_Scanner_App() {
003     Close();
004 }
005
006 bool PCI_Scanner_App::Initialize() {
007     return Open();
008 }
009
010 void PCI_Scanner_App::Shutdown() {
011     Close();
012 }
013
014 bool PCI_Scanner_App::Open() {
015     m_hDevice = CreateFileW(
016         L"\\\\\\\\.\\\\PCIScanner",
017         GENERIC_READ | GENERIC_WRITE,
018         0,
019         nullptr,
020         OPEN_EXISTING,
021         FILE_ATTRIBUTE_NORMAL,
022         nullptr
023     );
024
025     return (m_hDevice != INVALID_HANDLE_VALUE);
026 }
027
028 void PCI_Scanner_App::Close() {

```

```

029     if (m_hDevice && m_hDevice != INVALID_HANDLE_VALUE) {
030         CloseHandle(m_hDevice);
031         m_hDevice = nullptr;
032     }
033 }
034
035 std::vector<PCI_DEVICE_INFO> PCI_Scanner_App::Scan( ) {
036     std::vector<PCI_DEVICE_INFO> devices;
037
038     if (!IsOpen( )) {
039         throw std::runtime_error("Device not opened");
040     }
041
042     PCI_DEVICE_LIST deviceList{};
043     DWORD bytesReturned = 0;
044
045     BOOL result = DeviceIoControl(
046         m_hDevice,
047         IOCTL_PCI_GET_DEVICES,
048         nullptr, 0,
049         &deviceList, sizeof(deviceList),
050         &bytesReturned,
051         nullptr
052     );
053
054     if (!result) {
055         DWORD error = GetLastError();
056         throw std::runtime_error(std::format("DeviceIoControl
failed with error: {}", error));
057     }
058
059     devices.assign(deviceList.Devices, deviceList.Devices +
deviceList.NumberOfDevices);
060     return devices;
061 }
062
063 bool PCI_Scanner_App::IsOpen( ) const {
064     return m_hDevice && m_hDevice != INVALID_HANDLE_VALUE;
065 }

```

5 Вывод

В ходе выполнения лабораторной работы написана программа, выводящая список всех устройств, подключенных к шине PCI. Дополнительно к программе написан драйвер для шины PCI, осуществляющий низкоуровневое взаимодействие с консольной программой.