МИНИСТЕРСТВО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

УРАЛЬСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

кафедра молекулярной физики

ТЕХНИЧЕСКОЕ ОПИСАНИЕ И ИНСТРУКЦИЯ ПО ЭКСПЛУАТАЦИИ

ДРАЙВЕР МАСС-СПЕКТРОМЕТРА *МИ 1201-АГМ* ДЛЯ WINDOWS 2000

Разработчик _____ Александров О.Е.
к. ф.-м. н., программист 13 разряда __.__.03

Екатеринбург 2004

РЕФЕРАТ

Александров О.Е. *ДРАЙВЕР МАСС-СПЕКТРОМЕТРА МИ 1201-АГМ ДЛЯ WINDOWS 2000*: Техническое описание и инструкция по эксплуатации. Екатеринбург: УГТУ, 2004. 34 с.

Библиогр. 0 назв. Рис. 1. Табл. 9. Прил. 0.

Издание третье, исправленное и дополненное.

ДРАЙВЕР ЯДРА; WINDOWS 2000; МАСС-СПЕКТРОМЕТР; МИ 1201-АГМ.

Описано устройство и основные функции драйвера режима ядра Windows 2000 для управления масс-спектрометром МИ-1201 АГМ, также приведено описание модуля взаимодействия с драйвером для Delphi 5.

Драйвер предназначен для обеспечения доступа к оборудованию МИ-1201 АГМ (портам ввода-вывода) в ОС Windows 2000.

Язык программы драйвера — MS Visual C.

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ УСЛОВНЫХ ОБОЗНАЧЕНИЙ СИМВОЛОВ, ЕДИНИЦ И ТЕРМИНОВ

[**значение**]	– в квадратных скобках приводится значение величины (параметра) по умолчанию.
KMD	– Kernel Mode Driver (драйвер режима ядра).
VDD	– virtual DOS device (виртуальный драйвер DOS).
Драйвер	– драйвера режима ядра ОС Windows 2000, предназначенный для управление управления масс-спектрометром МИ 1201-АГМ.
Модуль	– модуль взаимодействия с драйвером для Delphi 5.
DDK	– Microsoft® Windows 2000® Driver Development Kit.
IRQ	– Interrupt Request Query или прерывание оборудования.

ВВЕДЕНИЕ

Инструкция описывает структуру и принципы работы драйвера режима ядра ОС Windows 2000 (далее *драйвер*). Драйвер предназначен для управления масс-спектрометром МИ 1201-АГМ. Дополнительно приведено описание модуля взаимодействия с драйвером для Delphi 5 (далее *модуль*).

Управление масс-спектрометром МИ1201-АГМ требует обращения к портам ввода-вывода процессора[1)]. Операционная система Windows NT/2000 запрещает непосредственную работу прикладных программ с портами ввода-вывода процессора. Для поддержки работы прикладных программ с портами ввода-вывода необходим драйвер устройства, работающий в режиме ядра системы (kernel mode driver или KMD).

Для работы DOS-программ, обращающихся непосредственно к портам ввода-вывода процессора, также необходим драйвер предназначенный для DOS-окна Windows NT/2000 и преобразующий эти обращения в обращения к драйверу ядра (virtual DOS device, VDD), описание соответствующего драйвера см. в [1].

Настоящий драйвер является модификацией драйвера, первоначально разработанного для ОС Windows NT 4.0. Основные изменения коснулись расширения функций драйвера — добавлены функции комплексных операций ввода/вывода, снижающие нагрузку на процессор, за счет уменьшения необходимого числа вызовов драйвера; добавлена поддержка особенностей системы драйверов Windows 2000 — PnP система, динамическая загрузка и выгрузка драйверов, установка драйвера в систему стандартными средствами Windows 2000. Драйвер поддерживает полную совместимость с программами, использующими предыдущую версию драйвера для ОС Windows NT 4.0.

Драйвер написан на языке программирования Си и реализует функции доступа к портам ввода-вывода процессора и некоторые более сложных операции ввода/вывода.

Модуль взаимодействия обеспечивает набор функций, облегчающий и упрощающий обращение к драйверу из программ Delphi 5. Модуль написан на языке программирования Borland Pascal.

---

[1]) Порты ввода-вывода процессора применяются для управления внешними устройствами.

1. ОБЩЕЕ ОПИСАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.2. Принципы работы Windows NT с внешними устройствами

Для управления масс-спектрометром под Windows NT/2000 необходим доступ к портам ввода-вывода процессора. Прямой доступ к портам ввода-вывода блокируется ОС Windows NT/2000 в целях повышения безопасности и стабильности работы системы. Доступ к портам возможен только через специальный драйвер Windows NT/2000, работающий в режиме ядра (kernel mode driver).

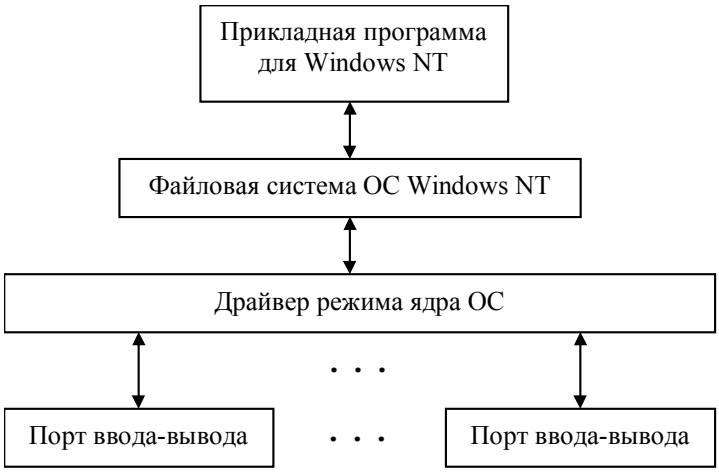
В самом общем и очень упрощенном виде схема взаимодействия прикладных программ Windows NT/2000 с портами ввода-вывода[1)] приведена на рис. 1.1.

Для управления масс-спектрометром МИ1201-АГМ в операционной системе Windows NT/2000 создан драйвер ядра «MI1201.sys», описываемый ниже. Драйвер является переработанной версией стандартного драйвера ввода-вывода в порты, поставляемого Microsoft Corp. в составе «Microsoft® Windows 2000® Driver Development Kit».

Рис. 1.2

1.4. Комплект поставки драйвера

Взаимодействие прикладных программ Windows NT с портами ввода-вывода



Драйвер «MI1201.sys» — программа на языке Си. Драйвер предоставляется в комплекте с исходным кодом и настоящей документацией.

В комплект входят:

1. Готовый к установке драйвер, см. папку «\Драйвер ядра»:
   1.1. «MI1201.sys» — исполняемый файл драйвера;
   1.2. «MI1201.inf» — inf-файл операционной системы Windows 2000, необходимый для установки драйвера в систему.
2. Описание драйвера, см. папку «\Документация».
3. Исходный текст драйвера (ВНИМАНИЕ!!! для компиляции исходного кода необходим «Microsoft® Windows 2000® Driver Development Kit»), см. папку «MI1201AGM» , где находятся:
   3.1. Файлы «MI1201AGM.dsw», «MI1201AGM.ncb» и «MI1201AGM.opt» — файлы проекта MS Visual C++ 6.0 SP5, позволяющие модифицировать и компилировать исходный код драйвера средствами графической среды MS Visual C++ 6.0. Проект позволяет получить работоспособный код драйвера, но требует настройки параметров вручную. Описание необходимой настройки графической среды MS Visual C++ 6.0 не входит в данный документ. Поставляемый проект не гарантирует работоспособность исполняемого кода. Для гарантированной работоспособности рекомендуется компиляция драйвера средствами DDK.
   3.2. Папка «KernelDrv» содержит исходный текст драйвера и вспомогательные программы и файлы для его компиляции, отладки и запуска:
      3.2.1. «dirs» — файл проекта DKK.
      3.2.2. «Mi1201ioctl.H» — описание команд и структур данных для взаимодействия с драйвером из пользовательских программ.
      3.2.3. «MakeFre.cmd» — компилирует FREE версию драйвера (для работы) средствами DDK и помещает результат «MI1201.sys» в папку «KernelDrv».
      3.2.4. «MakeChk.cmd» — компилирует CHECKED версию драйвера (для отладки) средствами DDK и помещает результат «MI1201.sys» в папку KernelDrv. **ВНИМАНИЕ!!!** CHECKED версия драйвера работает только в режиме отладки Windows 2000, иначе CHECKED версия драйвера может вызвать крах системы.
      3.2.5. «RestartDriver.cmd» — копирует «MI1201.sys» из папки «KernelDrv» в системную папку драйверов и перезапускает драйвер (без перезагрузки системы). **ВНИМАНИЕ!!!** запуск модифицированного драйвера может вызвать крах системы и невозможность перезагрузки — крах при последующей загрузке. Вы должны предварительно обеспечить возможность

загрузки альтернативной ОС с возможностью вручную удалить файл «%systemroot%\system32\drivers\mi1201.sys».

3.2.6. Папка «sys» содержит исходный текст драйвера, ресурсы и файлы описания проекта для DDK:

3.2.6.1. «FuncMi1201.c», «FuncMi1201.H» — вспомогательные функции.

3.2.6.2. «MI1201Get_SetControlInfo.c», «MI1201Get_SetControlInfo.h» — функции информационных вызовов драйвера.

3.2.6.3. «Mi1201.c», «Mi1201.H» — основные функции драйвера.

3.2.6.4. «Mi1201.RC» — файл ресурсов.

3.2.6.5. «common.ver» — описание версии драйвера.

3.2.6.6. «MAKEFILE», «SOURCES» — файлы проекта DDK.

3.2.6.7. папки «obj», «objchk» «objfre» пустые, они используются DDK при компиляции проекта.

3.2.7. Папка «utils» содержит вспомогательные утилиты Windows 2000. Подробнее утилиты и их назначение настоящий документ не описывает.

3.2.8. папка «Debug» используется MS Visual C++ 6.0.

1.6. Установка драйвера в систему

ВНИМАНИЕ!!! Хотя драйвер и был протестирован на нескольких компьютерах, но запуск драйвера может вызвать крах системы и невозможность перезагрузки — крах при последующей загрузке. Вы должны обеспечить возможность загрузки альтернативной ОС, с возможностью вручную удалить файл «%systemroot%\system32\drivers\mi1201.sys».

Чтобы установить драйвер (подразумевается, что необходимое оборудование установлено в компьютер и подключено к МИ-1201 АГМ):

1) Дважды щелкните «Установка оборудования» (Add/Remove Hardware) в «Панели управления» (Control Panel).
2) Выберите «Добавить/провести диагностику устройства» (Select Add/Troubleshoot a Device).
3) Выберите «Добавление нового устройства» (Add a new device).
4) Выберите «Нет, выбрать оборудование из списка» (No, I Want to Select the Hardware from a list).
5) Выберите «Другие устройства» (Other Devices).
6) Нажмите «Установить с диска...» (Have Disk) и укажите на папку содержащую файлы «MI1201.sys» и «MI1201.inf» (папка «\Драйвер ядра»).

Система скопирует файл MI1201.sys в папку «%systemroot%\system32\drivers» и запустит драйвер. Файл MI1201.inf определяет логическую конфигурацию драйвера.

1.8. Остановка/запуск драйвера и удаление драйвера

Чтобы приостановить работу драйвера или удалить драйвер (подразумевается, что драйвер был установлен):

1) Щелкните правой кнопкой мыши на «Мой компьютер».
2) Выберите «Оборудование» и нажмите «Диспетчер устройств».
3) Выберите в меню «Вид» пункт «Устройства по типу».
4) Щелкните дважды в списке на разделе «МИ-1201 АГМ».
5) Щелкните правой кнопкой мыши на «МИ-1201 АГМ Драйвер ввода/вывода в порты».
6) Выберите «Отключить...» — для отключения (приостановки) драйвера или «Удалить...» — для удаления драйвера из системы.

Отключенный драйвер может быть впоследствии запущен в любой момент. Для запуска отключенного драйвера выполните пункты 1)÷5) и выберите «Задействовать».

1.10. Краткое описание параметров INF-файла

В ОС Windows 2000 драйверы устанавливаются в систему посредством стандартной процедуры. Параметры установки требуется указать в специальном файле с расширением INF. Система Windows 2000, в соответствии с данными INF-файла копирует необходимые исполняемые файлы и файлы данных, создает необходимые разделы реестра и, главное, резервирует за драйвером необходимые системные ресурсы.

Настоящий драйвер не обладает никакой специфической для МИ-1201 АГМ функциональностью. По сути, это универсальный драйвер ввода/вывода и, следовательно, драйвер может использоваться для управления другими приборами, в том числе и приборами использующими физическую память, вместо портов, для обмена данными и для приборов, использцющие другие диапазоны и другое количество диапазонов (драйвер поддерживает до 10 диапазонов).

Для использования других диапазонов или типов портов следует отредактировать раздел:

```
[MI1201.NT.FactDef]
ConfigPriority=HARDRECONFIG
IOConfig=120-127(3ff::)  ; 10 bit decode ranging from 120-127
IOConfig=ED00-EDFF(3ff::) ; 10 bit decode ranging from ED00-EDFF
```

здесь приведено содержание раздела из «MI1201.INF», поставляемого с драйвером.

Указав соответствующее количество параметров *IOConfig*= и исправив их значения (см. документацию к DDK), можно изменить число диапазонов и диапазоны используемых портов. Указав параметр *MemConfig* = можно изменить тип используемых портов на физическую память.

Для установки новых параметров, драйвер следует удалить (см. пункт 1.8) и установить заново (см. пункт 1.6), используя измененный INF-файл.

1.12. Замечания по безопасности и контролю допуска к управлению

Настоящая версия драйвера не может контролировать права пользователя на допуск к функциям драйвера. Будучи установленным и запущеным (см. пункты 1.6 и 1.8) драйвер позволяет любому локальному пользователю обращаться к своим функциям в полном объеме.

Это приводит к небезопасности драйвера для оборудования масс-спектрометра. Например, любой пользователь может изменить настройки масс-спектрометра, что может привести к его неработоспособности или даже поломке.

Драйвер, также, небезопасен для работоспособности и устойчивости самой ОС. Например, злонамеренный или ошибочный вызов длительной операции ожидания, может существенно ухудшить отклик системы в других приложениях и даже блокировать выполнение других приложений.

Решением этой проблемы может стать дополнение драйвера средствами контроля допуска пользователя, имеющимися в составе ОС Windows 2000. Дополнение драйвера средствами контроля допуска требует доработки драйвера.

2. ОПИСАНИЕ ФУНКЦИЙ И ФОРМАТА ДАННЫХ ДРАЙВЕРА

2.2. Обращение к драйверу

Взаимодействие прикладной программы с драйвером осуществляется через вызовы системных функций, приведенные в табл. 2.1.

Таблица 2.2

Функции Windows API для работы с драйверами ядра

Функция	Назначение
1. *CreateFile*()	Создает файловый дескриптор (HANDLE) по символи-

Функция	Назначение
	ческому имени драйвера.
2. *CloseHandle*()	Уничтожает дескриптор, созданный функцией *CreateFile* и освобождает ресурсы.
3. *DeviceIOControl*()	Обмен данными с драйвером.

1.6.1. *CreateFile*()

Для работы с драйвером необходимо создать файловый дескриптор (HANDLE) посредством вызова *CreateFile*(). В качестве параметров передается имя файлового устройства, созданного драйвером при загрузке. При вызове *CreateFile*(), задается режим доступа (запись/чтение), режим совместного доступа, параметры защищенности, разрешение на создание файла, атрибуты и шаблон. Большинство этих параметров для работы с драйвером несущественны, типичные значения параметров приведены в табл. 2.2.

Пример подключения к драйверу (на Паскале):

```
HDevice := CreateFile(
    '\\.\MI1201_Dev',    // имя драйвера
    GENERIC_READ or GENERIC_WRITE,     // тип доступа
    FILE_SHARE_READ,     // разрешение совместного доступа
    nil,                 // адрес описания прав доступа
    OPEN_EXISTING,       // как создавать файл
    0,       // атрибуты файла
    0        // идентификатор файла для копирования атрибутов файла
);
```

В переменной *HDevice* будет возвращен дескриптор файла (handle) для доступа к драйверу.

Драйвер поддерживает символическое имя, использовавшееся в предыдущей версии драйвера для Windows NT: '\\.\PortIODev'. Это, вместе с поддержкой соответствующих функций ввода/вывода, позволяет исполнять программы, написанные под предыдущую версию драйвера, без изменений.

1.6.2. *CloseHandle*()

Для завершения работы с драйвером необходимо освободить файловый дескриптор посредством вызова *CloseHandle*(). В качестве параметра передается дескриптор, созданный ранее вызовом *CreateFile*(). Например:

CloseHandle(*HDevice*);

данный вызов освобождает системные ресурсы и прекращает доступ к драйверу.

Таблица 2.4

1.6.3. *DeviceIOControl*()

Обмен данными с драйвером происходит посредством вызовов функции *DeviceIOControl*(). В качестве параметров передается дескриптор устройства, управляющий код, буферы данных для входной и выходной информации и их размеры, а также длина возвращенной информации и указатель на структуру для асинхронной работы с устройством. Управляющий код формируется из битовых полей, говорящих системе о вызываемой функции. Например, запись в порт:

```
DeviceIOControl(
    HDevice,      // идентификатор файла
    IOCTL_GPD_WRITE_PORT_UCHAR,      // код операции
    @buf,  // указатель на буфер с данными вывода
    SizeOf(buf),   // размер буфера с данными вывода
    nil,      // указатель на буфер для получения данных ввода
    0,        // размер буфера для получения данных
    ret,      // переменная для размера возвращаемых данных
    nil       // указатель на структуру асинхронного ввода-вывода
);
```

аналогично, чтение порта:

```
DeviceIOControl(
    HDevice,      // идентификатор файла
```

Параметры функции *CreateFile*()[2]

Тип	Значение
1. строка	\\.\MI1201_Dev или \\.\PortIODev
2. двойное слово	GENERIC_READ or GENERIC_WRITE
3. двойное слово	FILE_SHARE_READ
4. указатель	nil
5. двойное слово	OPEN_EXISTING
6. двойное слово	0
7. дескриптор	0

```
    IOCTL_GPD_READ_PORT_UCHAR,      // код операции
    @TmpPort,   // указатель на буфер с данными вывода
    SizeOf(TmpPort),    // размер буфера с данными вывода
```

[2]) Все параметры здесь и далее приведены в синтаксисе Delphi.

```
    @Result,      // указатель на буфер для получения данных ввода
    1,            // размер буфера для получения данных
    ret,          // переменная для размера возвращаемых данных
    nil           // указатель на структуру асинхронного ввода-вывода
);
```

Размеры буферов должны соответствовать вызываемой функции. Настоящая версия драйвера требует точного соответствия размера буфера и функции. Передача буфера как меньшего, так и большего размеров приводит к ошибке — невыполнению затребованной операции.

2.4. Основные функции драйвера

Основные функции драйвера перечислены в табл. 2.3. В следующих разделах приведено подробное описание каждой функции и формата используемых данных. Все форматы данных описаны в файле-заголовке MI1201ioctl.h.

Таблица 2.6

Основные функции драйвера

Символьный идентификатор кода функции[3]	Выполняемые действия
IOCTL_GPD_READ_PORT_UCHAR IOCTL_GPD_READ_PORT_USHORT IOCTL_GPD_READ_PORT_ULONG	Чтение данных из порта длиной байт, слово и двойное слово, соответственно[4].
IOCTL_GPD_WRITE_PORT_UCHAR IOCTL_GPD_WRITE_PORT_USHORT IOCTL_GPD_WRITE_PORT_ULONG	Запись данных в порт длиной байт, слово и двойное слово, соответственно.
Новые функции драйвера[5]	
IOCTL_GPD_READ_PORT_BUFFER	Чтение массива данных из порта.
IOCTL_GPD_WRITE_PORT_BUFFER	Запись массива данных из порта.
IOCTL_GPD_WAIT_ON_PORT	Ожидание в течении заданного интервала времени события на порту.
IOCTL_GPD_WRITE_AND_WAIT_ON_PORT	Запись в порт 1 + Ожидание в течении заданного интервала времени события на порту 2.
IOCTL_GPD_MULTIFUNCTION_IO_OPE-	Проверка корректности входного буфера для

[3]) Коды см. в файле MI1201ioctl.h или MI1201ioctl.pas.
[4]) МИ-1201 АГМ использует только порты размером байт, но драйвер обладает возможностью записи/чтения и портов других размеров.
[5]) Эти функции отсутствуют в предыдущей версии.

RATION_CHECK_INPUT_BUFFER	MULTIFUNCTION_IO_OPERATION
IOCTL_GPD_MULTIFUNCTION_IO_OPERATION	Выполнение многофункциональной операции MULTIFUNCTION_IO_OPERATION
IOCTL_MI1201_GET_CONTROL_INFO	Запрос информации о драйвере
IOCTL_MI1201_SET_CONTROL_INFO	Управление драйвером
IOCTL_MI1201_ABORT	Запрос на немедленное прекращение всех операций драйвера, немедленно прерывает все ожидания и все многофункциональные операции.

2.6. Ввод одиночного значения

Драйвер при вызове *DeviceIOControl* с кодами:

IOCTL_GPD_READ_PORT_UCHAR
IOCTL_GPD_READ_PORT_USHORT
IOCTL_GPD_READ_PORT_ULONG

выполняет чтение данных из порта длиной байт, слово и двойное слово, соответственно. Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct _GENPORT_READ_INPUT {
    ULONG   PortNumber;
}   GENPORT_READ_INPUT, *PGENPORT_READ_INPUT;
```

где *PortNumber* - адрес порта.

Драйвер возвращает данные в буфере вывода (OUTPUT) следующего формата:

```
typedef struct _GENPORT_READ_OUTPUT {
union   {
        ULONG   LongData;
        USHORT  ShortData;
        UCHAR   CharData;
    };
}   GENPORT_READ_OUTPUT, *PGENPORT_READ_OUTPUT;
```

Размер буфера вывода зависит от кода операции: 1 байт, 2 байта и 4 байта, соответственно.

2.8. Вывод одиночного значения

Драйвер при вызове *DeviceIOControl* с кодами:

IOCTL_GPD_READ_WRITE_UCHAR
IOCTL_GPD_READ_WRITE_USHORT
IOCTL_GPD_READ_WRITE_ULONG

выполняет запись данных в порт длиной байт, слово и двойное слово, соответственно. Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct _GENPORT_WRITE_INPUT {
    ULONG   PortNumber;
    union   {
```

```
        ULONG   LongData;
        USHORT  ShortData;
        UCHAR   CharData;
    };
}   GENPORT_WRITE_INPUT, *PGENPORT_WRITE_INPUT;
```

где *PortNumber* - адрес порта; *Long|Short|CharData* - данные для записи в порт. Размер буфера ввода зависит от кода операции: 4+1 байт, 4+2 байта и 4+4 байта, соответственно.

Драйвер не возвращает данные в буфере вывода (OUTPUT), в качестве размера буфера вывода следует указывать ноль.

2.10. Ввод массива из одиночного порта

Драйвер при вызове *DeviceIOControl* с кодом:

IOCTL_GPD_READ_PORT_BUFFER

выполняет чтение массива данных из порта. Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _GENPORT_READ_MULTI_INPUT_BUFFER {
       ULONG   Port;
       UCHAR   PortType;
       ULONG   Count;
}   GENPORT_READ_MULTI_INPUT_BUFFER, *PGENPORT_READ_MULTI_INPUT_BUFFER;
```

где *Port* - адрес порта; *PortType* - тип порта допустимые значения см. табл. 2.4; *Count* - число считываемых значений.

Таблица 2.8

Поддерживаемые тивы портов

Тип порта (имя символьной константы)	Значение	Размер порта, байт
myPORT_UCHAR	0	1
myPORT_USHORT	1	2
myPORT_ULONG	2	4

Драйвер возвращает данные в буфере вывода (OUTPUT) следующего формата:

```
typedef struct  _GENPORT_READ_MULTI_OUTPUT_BUFFER {
       GENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER      Header;
       tLSHArrayUnion                               Data;
} GENPORT_READ_MULTI_OUTPUT_BUFFER, *PGENPORT_READ_MULTI_OUTPUT_BUFFER;
```

где тип GENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER определен

```
typedef struct  _GENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER {
       ULONG   Count;
} GENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER, *PGENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER;
```

а тип *tLSHArrayUnion*

```
typedef struct  _LSHArrayUnion {
        union {
                ULONG   ULong[1];
                USHORT  UShort[1];
                UCHAR   UChar[1];
                LONG    Long[1];
                SHORT   Short[1];
                CHAR    Char[1];
        };
} tLSHArrayUnion, *PLSHArrayUnion;
```

Размер буфера вывода зависит от *Count* - число считываемых значений и *PortType* - типа порта. И вычисляется так:

```
BufferSize = PortSize(PortType)*Count+sizeof(GENPORT_READ_MULTI_OUTPUT_BUFFER_HEADER),
```

где *PortSize*(*PortType*) значение размера порта в соответствии с табл. 2.8.

2.12. Вывод массива в одиночный порт

Драйвер при вызове *DeviceIOControl* с кодом:

IOCTL_GPD_WRITE_PORT_BUFFER

выполняет запись массива данных в порт. Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _GENPORT_WRITE_MULTI_INPUT_BUFFER {
        GENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER      Header;
        tLSHArrayUnion                               Data;
} GENPORT_WRITE_MULTI_INPUT_BUFFER, *PGENPORT_WRITE_MULTI_INPUT_BUFFER;
```

где тип *GENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER* определен так

```
typedef struct  _GENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER {
        ULONG   Port;
        UCHAR   PortType;
        ULONG   Count;
}   GENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER, *PGENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER;
```

а тип *tLSHArrayUnion*

```
typedef struct  _LSHArrayUnion {
        union {
                ULONG   ULong[1];
                USHORT  UShort[1];
                UCHAR   UChar[1];
                LONG    Long[1];
                SHORT   Short[1];
                CHAR    Char[1];
        };
} tLSHArrayUnion, *PLSHArrayUnion;
```

Port - адрес порта; *PortType* - тип порта допустимые значения см. табл. 2.8; *Count* - число записываемых в порт значений; *Data* - массив данных, программа должна разместить область достаточного размера, размер области вычисляется:

```
BufferSize = PortSize(PortType)*Count+sizeof(GENPORT_WRITE_MULTI_INPUT_BUFFER_HEADER),
```

где *PortSize*(*PortType*) значение размера порта в соответствии с табл. 2.8.

Драйвер не возвращает данные в буфере вывода (OUTPUT), в качестве размера буфера вывода следует указывать ноль.

2.14. Ожидание события на порту

Ожидание в течении заданного интервала времени события на порту.

IOCTL_GPD_WAIT_ON_PORT

Данная функция выполняет чтение порта в течение интервала времени не превосходящего заданный и сравнивает результат чтения с заданным образцом по следующему алгоритму:

$$Порт \text{ and } Маска = Образец,$$

при обнаружении желаемого результата (выполнении равенства) или истечении времени — ожидание прекращается и возвращаются результаты ожидания. Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _GENPORT_WAIT_INPUT_BUFFER {
        ULONG           Port;
        UCHAR           PortType;
        USHORT          TimeOut;
        USHORT          TimeResolution;
        ULONG           Mask;
        ULONG           Patt;
        ULONG           ControlFlags;
}  GENPORT_WAIT_INPUT_BUFFER, *PGENPORT_WAIT_INPUT_BUFFER;
```

где *Port* - адрес порта; *PortType* - тип порта допустимые значения см. табл. 2.8; *TimeOut* -время ожидания в миллисекундах; *TimeResolution* - периодичность опроса порта ожидания в миллисекундах; *Mask* - маска, с которой драйвер выполняет побитовый *and* для значения прочитанного из порта, прежде чем сравнить с образцом; *Patt* - образец, при получении которого драйвер считает, что событие наступило; *ControlFlags* - битовые флаги дополнительного управления ожиданием, см. табл. 2.5.

Периодичность опроса порта *TimeResolution* используется так: драйвер считывает порт через интервалы времени *TimeResolution*, в промежутке между опросами драйвер ничего не делает, передав управление системе — это уменьшает загрузку процессора на некритических ожиданиях. Значение *TimeResolution* может изменяться от 1 до *TimeOut*, в этом случае драйвер использует указанный интервал. При *TimeResolution* = 0, драйвер использует минимальный интервал системного таймера, это интервал зависит от ОС и компьютера и обычно составляет 5÷10 *мс*. При *TimeResolution* > *TimeOut* драйвер устанавливает

$$TimeResolution = TimeOut/4.$$

Флаги управления ожиданием (номера битов) определены так:

```
typedef enum _tWaitControlFlag {
  wcfUserRequest,
  wcfUserMode,
  wcfAlertable,
  wcfWaitOnAPC_Level,
  wcfWaitOnDISPATCH_Level,
```

```
  wcfAllowZeroResolution,
  wcfUseQueryPerformanceCounter,
  wcfReturnWaitIntervalIn100nanosecUnit
} tWaitControlFlag;
```

Таблица 2.10

Флаги управления ожиданием[6] (номера бита)	Примечания
0. wcfUserRequest	запрос пользователя, иначе [*Executive*], см. описание функции *KeWaitForSingle-Object* в DDK.
1. wcfUserMode	ожидать в режиме *UserMode*, иначе [*KernelMode*], см. описание функции *KeWaitForSingleObject* в DDK.
2. wcfAlertable	ожидание можно прервать, иначе [*FALSE*], см. описание функции *KeWaitForSingleObject* в DDK.
3. wcfWaitOnAPC_Level[7]	ожидание на IRQL=APC_LEVEL
4. wcfWaitOnDISPATCH_Level[7]	ожидание на IRQL=DISPATCH_LEVEL
5. wcfAllowZeroResolution	разрешить ожидание с *TimeResolution* =0 разрешением — непрерывный опрос порта.
6. wcfUseQueryPerformanceCounter[7]	использовать *KeQueryPerformance-Counter* для определения времени ожидания, вместо обычного системного таймера, см. описание функции в DDK.
7. wcfReturnWaitIntervalIn100nanosecUnit[7]	возвратить данные ожидания в единицах 100 наносекунд.

Драйвер возвращает данные в буфере вывода (OUTPUT) следующего формата:

```
typedef struct  _GENPORT_WAIT_OUTPUT_BUFFER {
        BOOLEAN         Event;
        ULONG           WaitResult;
        ULONG           TimeOut;
        ULONG           Data;
}   GENPORT_WAIT_OUTPUT_BUFFER, *PGENPORT_WAIT_OUTPUT_BUFFER;
```

[6]) Для удобства в MI1201ioctl.h определены битовые константы для флагов bit_wcfUserRequest, и т.д.

[7]) Флаг действует только для IOCTL_GPD_WRITE_AND_WAIT_ON_PORT и IOCTL_GPD_MULTIFUNCTION_IO_OPERATION.

где *Event* - результат ожидания: *Event* = TRUE - дождались; *WaitResult* - детальное описание результатов ожидания, см. ниже тип *tPortWaitResult* и табл. 2.6; *TimeOut* - реальное время, затраченное на ожидание в *мс*, точность измерения зависит от *TimeResolution* и интервала системного таймера; *Data* - последние данные, считанные с порта.

Результаты ожидания определены типом *tPortWaitResult* и описаны в табл. 2.6:

```
typedef enum _tPortWaitResult {
        pwrSUCCESS,
        pwrTIMEOUT,
        pwrABANDONED
} tPortWaitResult;
```

Таблица 2.12

Результаты ожидания	Примечания
0. pwrSUCCESS	дождались
1. pwrTIMEOUT	не дождались — таймаут
2. pwrABANDONED	ожидание было прервано, см. IOCTL_MI1201_ABORT

2.16. Запись в порт 1 и ожидание события на порту 2

Запись в порт 1 и ожидание события на порту 2:
IOCTL_GPD_WRITE_AND_WAIT_ON_PORT.

Данная функция выполняет запись в указанный порт и ожидание события на другом указанном порту (см. пункт 2.14). Функция введена для возможности отслеживать «быстрые» события, связанные с запуском счетчиков или работой контроллера эл.магнита, когда между стартом исполнения функции оборудования и результатом интервал времени менее 100 *мс*. Дело в том, что в «нормальном» режиме работы программы невозможно гарантировать интервал времени между двумя обращениями к драйверу, поэтому невозможно точно реагировать на «быстрые» ответы оборудования масс-спектрометра. Например, запуск счета на интервал 100 *мс* или менее не позволяет определить, посредством последующего чтения порта состояния: запустился ли счет? Пока второе обращение к драйверу будет обработано системой может пройти более 100 *мс* и счетчик уже закончит работу.

Данная функция позволяет провести запись в порт и ожидать на уровне обработки прерываний IRQL = DISPATCH_LEVEL (см. табл. 2.10), на котором блокируется переключение задач, т.е. драйвер практически монопольно использует процессор (блокируется не все, обработка некоторых прерывания не пре-

кращается, например, таймера). В этом случае можно практически гарантированно отслеживать быстрые реакции оборудования.

Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _GENPORT_WRITE_AND_WAIT_INPUT_BUFFER {
        GENPORT_WRITE_AND_WAIT_WRITE_DATA           WriteData;
        GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA    WaitData;
} GENPORT_WRITE_AND_WAIT_INPUT_BUFFER, *PGENPORT_WRITE_AND_WAIT_INPUT_BUFFER;
```

где данные записи в порт *GENPORT_WRITE_AND_WAIT_WRITE_DATA*

```
typedef struct  _GENPORT_WRITE_AND_WAIT_WRITE_DATA {
        PORT_DATA       Port;
        tLSHUnion       Data;
} GENPORT_WRITE_AND_WAIT_WRITE_DATA, *PGENPORT_WRITE_AND_WAIT_WRITE_DATA;
```

описание порта *PORT_DATA*

```
typedef struct  _PORT_DATA {
        ULONG           Number;
        ULONG           Type;
        ULONG           MemoryType;
}   PORT_DATA, *PPORT_DATA;
```

где *Number* - номер порта; *Type* - тип порта, см. табл. 2.8; значение *MemoryType* - не используется, это поле инициализируется самим драйвером при трансляции порта. Описание данных для записи в порт *tLSHUnion*, см. пункт 2.8.

Данные ожидания на порту *GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA*

```
typedef struct  _GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA {
        PORT_DATA       Port;
        ULONG           TimeOut;
        ULONG           TimeResolution;
        ULONG           Mask;
        ULONG           Patt;
        ULONG           ControlFlags;
} GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA, *PGENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA;
```

Смысл и значения полей см. пункт 2.14 описание *GENPORT_WAIT_OUTPUT_BUFFER*. Отличия состоят в размерах полей и в том, что данный вызов драйвера реагирует на флаги в *ControlFlags*:

```
        wcfWaitOnAPC_Level,
        wcfWaitOnDISPATCH_Level,
        wcfUseQueryPerformanceCounter,
        wcfReturnWaitIntervalIn100nanosecUnit,
```

см. табл. 2.10.

При ожидании с флагом *wcfWaitOnDISPATCH_Level*, опрос порта ведется непрерывно, значение периодичности опроса порта *TimeResolution* игнорируется, флаг *wcfAlertable* — игнорируется. Не рекомендуется ожидание с *wcfWaitOnDISPATCH_Level* на длительных (более 100 *мс*) интервалах, поскольку это практически останавливает исполнение пользовательских программ.

Драйвер возвращает данные в буфере вывода (OUTPUT) следующего формата:

```
typedef struct  _GENPORT_WAIT_OUTPUT_BUFFER {
        BOOLEAN         Event;
```

```
        ULONG           WaitResult;
        ULONG           TimeOut;
        ULONG           Data;
}   GENPORT_WAIT_OUTPUT_BUFFER, *PGENPORT_WAIT_OUTPUT_BUFFER;
```

где *Event* - результат ожидания, если *TRUE*, то дождались; *WaitResult* - детальное описание результатов ожидания, см. ниже тип *tPortWaitResult* и табл. 2.12; *TimeOut* - реальное время, затраченное на ожидание в *мс* или в 100 *нс* интервалах, если был установлен упр. флаг wcfReturnWaitIntervalIn100nanosecUnit, точность измерения зависит от *TimeResolution* и интервала системного таймера; *Data* - последние данные, считанные с порта. Результаты ожидания определены типом *tPortWaitResult* и описаны в табл. 2.12.

2.18. Многофункциональная операция ввода/вывода

Многофункциональная операция ввода/вывода состоит из двух вызовов:

IOCTL_GPD_MULTIFUNCTION_IO_OPERATION_CHECK_INPUT_BUFFER

IOCTL_GPD_MULTIFUNCTION_IO_OPERATION

Первый выполняет только проверку входного (INPUT) буфера команды на допустимость (правильность) и возвращает результаты проверки, без выполнения команд. Второй вызов проверяет весь буфер обращением к IOCTL_GPD_-MULTIFUNCTION_IO_OPERATION_CHECK_INPUT_BUFFER и, если нет ошибок, то выполняет команды входного (INPUT) буфера и возвращает результат выполнения команд.

Операция проверки буфера предназначена для целей отладки, готовая программа, по-возможности, должна избегать ее использования.

Многофункциональная операция ввода/вывода позволяет уменьшить число вызовов драйвера, и тем самым снизить загрузку ОС.

Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_INPUT_BUFFER {
        ULONG                                   Count;
        GENPORT_MULTIFUNCTION_IO_INPUT_HEADER   Data;
}   GENPORT_MULTIFUNCTION_IO_INPUT_BUFFER, *PGENPORT_MULTIFUNCTION_IO_INPUT_BUFFER;
```

где *Count* - число операций в буфере; *Data* - заголовок записи первой операции. Все записи операций выровнены на 32 бита. Заголовок операции определен так

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_INPUT_HEADER {
        tMultifunctionOpCode   OpCode;
}   GENPORT_MULTIFUNCTION_IO_INPUT_HEADER, *PGENPORT_MULTIFUNCTION_IO_INPUT_HEADER;
```

где *OpCode* - код операции, который определяет размер записи операции, допустимые коды операций определены так:

```
typedef enum _tMultifunctionOpCode {
        mfocRead,
        mfocWrite,
        mfocReadBuffer,
        mfocWriteBuffer,
        mfocWait,
        mfocWriteAndWait,
```

```
        mfocInvalidOperation
} tMultifunctionOpCode, *ptMultifunctionOpCode;
```

описание операций приведено в табл. 2.7.

Таблица 2.14

СПИСОК ОПЕРАЦИЙ МНОГОФУНКЦИОНАЛЬНОГО ВВОДА/ВЫВОДА

Операция	Примечания
0. mfocRead	чтение одного значения
1. mfocWrite	запись одного значения
2. mfocReadBuffer	чтение нескольких значений
3. mfocWriteBuffer	запись нескольких значений
4. mfocWait	ожидание
5. mfocWriteAndWait	запись одного значения+ожидание
6. mfocInvalidOperation	неверная операция - зарезервировано

Операция проверки входного буфера возвращает данные в буфере вывода (OUTPUT) в следующем формате:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_ERROR_DESCRIPTION {
        tMultifunctionError    ErrorCode;
        union{
                ULONG   RecordNumber;   // номер записи, в которой ошибка
                ULONG   OutputDataSize; // полный размер буфера для возврата данных
        };
}   GENPORT_MULTIFUNCTION_IO_ERROR_DESCRIPTION, *PGENPORT_MULTIFUNCTION_IO_ERROR_DESCRIPTION;
```

где *ErrorCode* - код ошибки, определенный так:

```
typedef enum _tMultifunctionError {
        mferrOK,
        mferrBufferTooSmall, // буфер слишком мал даже для заголовка буфера
        mferrRecordsCountEqZero,
        mferrBufferTooSmall_2, // буфер слишком мал для заголовка операции
        mferrInvalidOpCode,
        mferrBufferTooSmall_3, // буфер слишком мал для записи операции
        mferrBufferTooSmall_4, // буфер слишком мал массива данных операции
        mferrInvalidPort,
        mferrInvalidPort_2,  // неверный порт ожидания
        mferrWriteBuffer_CountZero,
        mferrReadBuffer_CountZero,
        mferrInvalidPortType,
        mferrOutputBufferTooLarge // требуется буфер OUTPUT >0xffffffff байт
} tMultifunctionError, *ptMultifunctionError;
```

смысл ошибок следует из их названий; если $ErrorCode \neq mferrOK$, то *RecordNumber* = номеру первой ошибочной записи, иначе $ErrorCode = mferrOK$ и *OutputDataSize* = размеру буфера вывода (OUTPUT), необходимого для для выполнения IOCTL_GPD_MULTIFUNCTION_IO_OPERATION с данным набором команд.

2.20. Форматы записей команд для многофункциональной операции ввода/вывода

Все записи команд многофункциональной операции ввода/вывода содержат заголовок записи и данные записи, заголовок определен так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_INPUT_HEADER {
        tMultifunctionOpCode   OpCode;
}   GENPORT_MULTIFUNCTION_IO_INPUT_HEADER, *PGENPORT_MULTIFUNCTION_IO_INPUT_HEADER;
```

где *OpCode* - код операции, который определяет размер записи операции, допустимые коды операций, см. пункт 2.18 и табл. 2.14.

Данные операции *mfocRead* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_READ_INPUT {
        GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
        PORT_DATA                                    Port;
}   GENPORT_MULTIFUNCTION_IO_READ_INPUT, *PGENPORT_MULTIFUNCTION_IO_READ_INPUT;
```

где определение *PORT_DATA*, см. пункт 2.16.

Данные операции *mfocWrite* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_WRITE_INPUT {
        GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
        PORT_DATA                                    Port;
        tLSHUnion                                    Data;
}   GENPORT_MULTIFUNCTION_IO_WRITE_INPUT, *PGENPORT_MULTIFUNCTION_IO_WRITE_INPUT;
```

где определение *PORT_DATA*, см. пункт 2.16; определение *tLSHUnion*:
и тип *tLSHUnion* определен так

```
typedef struct  _LSHUnion {
    union   {
               ULONG   ULong;
               USHORT  UShort;
               UCHAR   UChar;
               LONG    Long;
               SHORT   Short;
               CHAR    Char;
        };
}   tLSHUnion, *PLSHUnion;
```

Данные операции *mfocReadBuffer* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_READ_BUFFER_INPUT {
        GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
        PORT_DATA                                    Port;
        ULONG                                        Count;
}   GENPORT_MULTIFUNCTION_IO_READ_BUFFER_INPUT, *PGENPORT_MULTIFUNCTION_IO_READ_BUFFER_INPUT;
```

где *Count* - число считываемых значений; определение *PORT_DATA*, см. пункт 2.16.

Данные операции *mfocWriteBuffer* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_WRITE_BUFFER_INPUT {
        GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
        PORT_DATA                                    Port;
        ULONG                                        Count;
        tLSHArrayUnion                               Data;
}   GENPORT_MULTIFUNCTION_IO_WRITE_BUFFER_INPUT, *PGENPORT_MULTIFUNCTION_IO_WRITE_BUFFER_INPUT;
```

где *Count* - число записываемых значений; *Data* - данные для записи; определение *PORT_DATA*, см. пункт 2.16; определение *tLSHArrayUnion*, см. пункт 2.10. Размер данных *Data* в байтах вычисляется так: [*Count*·*PortSize*(*Port.Type*) + 3)/4]·4. Размер данных округлен до целого числа двойных слов, здесь *PortSize*(*Port.Type*) размер порта по табл. 2.8.

Данные операции *mfocWait* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_WAIT_INPUT {
       GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
       GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA     WaitData;
}   GENPORT_MULTIFUNCTION_IO_WAIT_INPUT, *PGENPORT_MULTIFUNCTION_IO_WAIT_INPUT;
```

где определение *GENPORT_GENPORT_WRITE_AND_WAIT_WAIT_DATA*, см. пункт 2.16.

Данные операции *mfocWriteAndWait* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_WRITE_AND_WAIT_INPUT {
       GENPORT_MULTIFUNCTION_IO_INPUT_HEADER        Header;
       GENPORT_WRITE_AND_WAIT_INPUT_BUFFER          Data;
}   GENPORT_MULTIFUNCTION_IO_WRITE_AND_WAIT_INPUT, *PGENPORT_MULTIFUNCTION_IO_WRITE_AND_WAIT_INPUT;
```

где определение GENPORT_WRITE_AND_WAIT_INPUT_BUFFER, см. пункт 2.16.

2.22. Форматы возвращаемых данных для многофункциональной операция ввода/вывода

Не все записи команд многофункциональной операции ввода/вывода возвращают данные. Операции *mfocWrite*, *mfocWriteBuffer* не возвращают ничего.

Операция многофункционального ввода/вывода возвращает данные в буфере вывода (OUTPUT) в следующем формате:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_OUTPUT_BUFFER {
       GENPORT_MULTIFUNCTION_IO_OUTPUT_HEADER       Header;
       ULONG                                        Data;
}   GENPORT_MULTIFUNCTION_IO_OUTPUT_BUFFER, *PGENPORT_MULTIFUNCTION_IO_OUTPUT_BUFFER;
```

где *GENPORT_MULTIFUNCTION_IO_OUTPUT_HEADER* определен

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_OUTPUT_HEADER {
       ULONG   OpCount;
       ULONG   RecordCount;
}   GENPORT_MULTIFUNCTION_IO_OUTPUT_HEADER, *PGENPORT_MULTIFUNCTION_IO_OUTPUT_HEADER;
```

где *OpCount* - число операций в буфере (из INPUT буфера); *RecordCount* - число записей в возвращаемом буфере.

Записи возвращаемые любой операцией всегда имеют размер кратный 4 байтам (двойному слову). Если реальные данные операции имеют размер меньше, то добавочные байты следует полагать неопределенными, т.е. не имеющими конкретного значения.

Возвращаемые данные операции *mfocRead* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_READ_OUTPUT {
       tLSHUnion      Data;
}   GENPORT_MULTIFUNCTION_IO_READ_OUTPUT, *PGENPORT_MULTIFUNCTION_IO_READ_OUTPUT;
```

где *Data* - содержит считанное с порта значение, соответствующей длины, остальная часть *Data* не определена; определение *tLSHUnion*, см. пункт 2.18.

Возвращаемые данные операции *mfocReadBuffer* определены так:

```
typedef struct  _GENPORT_MULTIFUNCTION_IO_READ_BUFFER_OUTPUT {
       tLSHArrayUnion Data;
}   GENPORT_MULTIFUNCTION_IO_READ_BUFFER_OUTPUT, *PGENPORT_MULTIFUNCTION_IO_READ_BUFFER_OUTPUT;
```

где *Data* - данные, считанные с порта; определение *tLSHArrayUnion*, см. пункт 2.10. Размер данных вычисляется так: [*Count*·*PortSize*(*Port.type*) + 3)/4]·4. Размер данных округлен до целого числа двойных слов, здесь *PortSize*(*Port.type*) размер порта по табл. 2.8.

Возвращаемые данные операций *mfocWait* и *mfocWriteAndWait* определены типом GENPORT_WAIT_OUTPUT_BUFFER, см. пункты 2.14 и 2.16.

2.24. Информационные функции и функции управления

2.24.2. Прекращение операций драйвера

Код функции

IOCTL_MI1201_ABORT

позволяет управлять прекращением/разрешением операций драйвера. При установке запрета немедленно прерываются все ожидания и все многофункциональные операции.

Основное назначение команды IOCTL_MI1201_ABORT — аварийное прекращение операций драйвером при зацикливании, ошибочном задании большого времени ожидания или необходимости завершить программу.

Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _MI1201_ABORT_INPUT_BUFFER {
       BOOLEAN   Stop;
}   MI1201_ABORT_INPUT_BUFFER, *PMI1201_ABORT_INPUT_BUFFER;
```

где параметр *Stop* = *TRUE* переводит драйвер в режим запрета операций; *FALSE* - разрешает операции.

При вызове в условиях запрета (после вызова IOCTL_MI1201_ABORT с параметром *Stop* = *TRUE*) любой операции ввода/вывода, кроме информационных операций описанных в пункте 2.24, операция не выполняется и *DeviceIOControl* возвращает результат *STATUS_REQUEST_ABORTED*.

При включении запрета операций ввода/вывода в момент исполнения операции ввода/вывода:

1) Все исполняющиеся операции чтения/записи одиночного значения и массива (см. пункты 2.6, 2.8, 2.10, 2.12) выполнятся до конца и *DeviceIOControl* возвратит *STATUS_SUCCESS*.
2) Все исполняющиеся операции ожидания (см. пункт 2.14) прерываются немедленно, возвращается *Event* = *TRUE*, только если событие на порту уже

имеет место при установке запрета, в поле *WaitResult* возвращается состояние *pwrABANDONED*.

3) Все исполняющиеся вызовы операции «запись в порт и ожидание» (см. пункт 2.16) прерываются этапе ожидания, запись в порт будет произведена и возвращается *Event = TRUE*, только если событие на порту ожидания имеет место немедленно (последнее маловероятно), в поле *WaitResult* возвращается состояние *pwrABANDONED*.
4) Все исполняющиеся вызовы многофункциональной операции ввода/вывода (см. пункт 2.18) прерываются, прерывание элементарных операций происходит по описанным выше правилам, возвращается *OpCount* = числу реально выполненных команд из буфера INPUT и *RecordCount* = число реальных записей в буфере OUTPUT, в буфере OUTPUT возвращаются данные *RecordCount* операций, остальные данные неопределены.

Драйвер возвращает данные в буфере вывода (OUTPUT) следующего формата:

```
typedef struct  _MI1201_ABORT_OUTPUT_BUFFER {
       BOOLEAN   PreviousState;
}   MI1201_ABORT_OUTPUT_BUFFER, *PMI1201_ABORT_OUTPUT_BUFFER;
```

где параметр *PreviousState* - предыдущее состояние запрета, до применение значения параметра *Stop*.

2.24.4. Запрос информации о драйвере

Код функции

IOCTL_MI1201_GET_CONTROL_INFO

позволяет управлять запросить данные о версии драйвера, состоянии драйвера и т.п., полный перечень запросов приведен в табл. 2.8.

Используется следующий формат буфера с данными ввода (INPUT):

```
typedef struct  _MI1201_CONTROL_INFO_INPUT_BUFFER_BASE_PART {
       ULONG   ControlCode;
       ULONG   ControlSubCode;
       ULONG   FullSize;
}   MI1201_CONTROL_INFO_INPUT_BUFFER_BASE_PART, *PMI1201_CONTROL_INFO_INPUT_BUFFER_BASE_PART;
```

где *ControlCode* - основной код операции, см. табл. 2.8; *ControlSubCode* - доп. код субоперации, в настоящее время не используется; *FullSize* - полный размер данных в буфере INPUT, должен совпадать с соответствующим размером *SizeOf*(*buf*), переданным *DeviceIOControl*, см. пункт 2.2.

Таблица 2.16

КОД ЗАПРОСА ИНФОРМАЦИИ О ДРАЙВЕРЕ

Код	Примечания
MI1201_GET_CONTROL_CODE_GET_VERSION	Запрос номера версии драйвера.

MI1201_GET_CONTROL_CODE_GET_CAPABILITIES	Запрос возможностей драйвера, коды возможностей перечислены в .
MI1201_GET_CONTROL_CODE_GET_ABORT_STATE	Состояние запрета на операции, см. пункт 2.24.2.

Формат возвращаемого буфера вывода (OUTPUT), зависит от кода запроса информации, см. табл. 2.16.

Формат возвращаемого буфера MI1201_GET_CONTROL_CODE_GET_-VERSION:

```
typedef struct  _MI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_VERSION {
    ULONG   InternalVersion;     // внутренний номер версии
}MI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_VERSION, *PMI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_VERSION;
```

Формат возвращаемого буфера MI1201_GET_CONTROL_CODE_GET_-CAPABILITIES:

```
typedef struct  _MI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_CAPABILITIES {
    ULONG   Capabilities;     // флаги функциональных возможностей, см. табл. 2.9
}MI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_CAPABILITIES,
*PMI1201_CONTROL_INFO_OUTPUT_BUFFER_GET_CAPABILITIES;
```

Таблица 2.18

БИТОВЫЕ КОДЫ ВОЗМОЖНОСТЕЙ ДРАЙВЕРА

Код (значение)	Примечания
MI1201_CAP_FLAG_ARRAY_IO_SINGLE_PORT (1)	Ввод/вывод массива в один порт.
MI1201_CAP_FLAG_ARRAY_IO_RAW_PORTS (2)	Ввод/вывод массива в разные порты (не используется).
MI1201_CAP_FLAG_WAIT_ON_SINGLE_PORT (4)	Ожидание на порту.
MI1201_CAP_FLAG_WRITE_AND_WAIT_ON_SINGLE_PORT (8)	Запись в порт 1 и ожидание на порту 2.
MI1201_CAP_FLAG_ARRAY_IO_RAW_PORTS_AND_WAIT (16)	Многофункциональный ввод/вывод, см. пункт 2.18

Формат возвращаемого буфера MI1201_GET_CONTROL_CODE_GET_-ABORT_STATE:

```
typedef struct  _MI1201_CONTROL_INFO_OUTPUT_GET_ABORT_STATE {
    BOOLEAN   State;     // состояние запрета;
}MI1201_CONTROL_INFO_OUTPUT_GET_ABORT_STATE, *PMI1201_CONTROL_INFO_OUTPUT_GET_ABORT_STATE;
```

2.24.6. Управление драйвером

Код функции

IOCTL_MI1201_GET_CONTROL_INFO.

Данная функция в настоящей версии драйвера не реализована.

3. ОПИСАНИЕ ИНТЕРФЕЙСНОГО МОДУЛЯ DELPHI ДЛЯ ВЫЗОВА ФУНКЦИЙ ДРАЙВЕРА

Для удобства использования функций драйвера из программ для Delphi, с драйвером поставляется интерфейсный модуль, в котором описаны функции, осуществляющие обращение к драйверу.

Функции интерфейсного модуля ориентированы только на порты размером 1 байт (myPORT_UCHAR, см. табл. 2.8).

Интерфейсный модуль совместим со старой версией драйвера для Windows NT.

3.2. Состав интерфейсного модуля

Файлы интерфейсного модуля располагаются в папке «Delphi-интерфейс», где нахдятся:

1. «MI1201ioctl.pas» — описание типов данных и констант файла «MI1201ioctl.h» в переложении для Delphi.
2. «PortsIO.pas» — главный файл интерфейса, обеспечивающий автоматический выбор способа обращения к оборудованию (через драйвер или напрямую), в зависимости от ОС. Кроме того, «PortsIO.pas» позволяет подключить внешние процедуры ввода/вывода в одиночный порт, последнее необходимо для подключения эмулятора оборудования. Совместим со старой версией драйвера для Windows NT.
3. «DirectPortsIO.pas» - файл интерфейса, обеспечивающий функции прямого доступа к портам в ОС Win9x.
4. «DriverPortsIO.pas» - файл интерфейса, обеспечивающий функции доступа к портам через драйвер в ОС Windows NT/2000. Совместим со старой версией драйвера для Windows NT, но вызов процедур обращения к дополнительным функциям драйвера Windows 2000 будет возвращать ошибку.
5. «IOBuffer.pas» — вспомогательный файл, поддерживающий блокируемый буфер для функций ввода/вывода массива, и обеспечивающий возможность вызова этих функций из различных потоков (THREADs).

3.4. Основные функции интерфейсного модуля

Для краткости будут описаны только функции из «PortsIO.pas», описание функций других частей см. в комментариях соответствующих файлов.

В модуле «PortsIO.pas» определены следующие функции:

```
procedure OutByte(b:Byte; Port:word); register; —
```

Вывод байта *b* в порт *Port*.

```
function InByte(Port:word):byte; register;
```

Ввод байт из порта *Port*.

```
function OutByteBuffer(Port:word; var Bytes; Size:word):boolean; register;
```

Вывод массива байт *Bytes* размером *Size* байт в порт *Port*, возвращает *TRUE*, если нет ошибки.

```
function InByteBuffer(Port:word; var Bytes; Size:word):boolean; register;
```

Ввод массива байт *Bytes* размером *Size* байт из порта *Port*, возвращает *TRUE*, если нет ошибки.

```
function Wait(Port:word; Mask,Patt:Byte; Time:longint; var WaitTime:longint):boolean; register;
```

Ожидание на порту *Port* размером в байт с маской *Mask* и образцом *Patt* в течение интервала времени не более *Time* мс, возвращает *TRUE*, если дождались и реальное время ожидания [мс] в *WaitTime*.

```
function LazyWait(Port:word; Mask,Patt:Byte; Time:longint;
                      TimeStep:word; var WaitTime:longint):boolean; register;
```

Ленивое ожидание на порту *Port* размером в байт с маской *Mask* и образцом *Patt* в течение интервала времени не более *Time* мс, с интервалом опроса порта *TimeStep* мс, возвращает *TRUE*, если дождались и реальное время ожидания [мс] в *WaitTime*.

```
function OutByteAndWait(Port:word; aByte:byte;
                      WaitPort:word; Mask,Patt:Byte; Time:longint; TimeStep:word;
                      var WaitTime:longint):boolean; register;
```

Вывод байта *aByte* в порт *Port* и ожидание на порту *WaitPort* размером в байт с маской *Mask* и образцом *Patt* в течение интервала времени не более *Time* мс, с интервалом опроса порта *TimeStep* мс, возвращает *TRUE*, если дождались и реальное время ожидания [мс] в *WaitTime*.

```
function MultifunctionIO(const CommandData; CommandDataSize:word;
                        var ResultData;  ResultDataSize:word):boolean; register;
```

Выполнение многофункциональной операции ввода/вывода (см. пункт 2.18), заданной буфером *CommandData*. Возвращает *TRUE*, если нет ошибки и результаты в буфере *ResultData*.

```
function IsError:boolean;
```

Возвращает *TRUE*, если предыдущий вызов любой функции ввода/вывода завершился с ошибкой.

```
function LastErrorCode:longint;
```

Возвращает код ошибки, если предыдущий вызов любой функции ввода/вывода завершился с ошибкой, иначе возвращает 0.

```
function AbortState:boolean; register;
```

Возвращает состояние запрета исполнения команд: *TRUE* — выполнение запрещено.

```
function AbortOperation(State:boolean):boolean; register;
```

Устанавливает/снимает состояние запрета исполнения команд: *State = TRUE* — выполнение запрещено и возвращает предшествующее вызову *AbortOperation* значение состояния запрета исполнения команд.

```
function IOType:tIOType;
```

Возвращает тип используемого ввода/вывода:

```
tIOType=(ioDirect, ioViaFileSystem, ioError, ioEmulator, ioUnknown);
```

где *ioDirect* - используется прямой доступ к портам; *ioViaFileSystem* - используется драйвер; *ioEmulator* — используется внешнее подключение.

```
procedure SetIOType(aIOType:tIOType);
```

Устанавливает тип используемого ввода/вывода, реагирует только на *aIOType = ioViaFileSystem|ioDirect*. Состояние *ioEmulator* может быть установлено вызовом *SetExternalInOutByte*, см. описание ниже. Установка неверного типа подключения может привести к неработоспособности процедур ввода/вывода.

```
procedure SetDefaultIOType;
```

Устанавливает тип используемого ввода/вывода, автоматически, распознавая тип ОС: *ioViaFileSystem* — для Windows NT/2000 или *ioDirect* — для Windows 9x и инициализирует соответствующие подключения. Если было установлено внешнее подключение (*ioEmulator*), то оно будет сброшено.

```
procedure SetExternalInOutByte(
              aInB:TInB_Func;
              aOutB:TOutB_Proc;
              aLastError:TLastError_Func); register;
```

Устанавливает внешнее подключение для *OutByte*, *InByte* и *LastErrorCode* и подключение к процедурам из модуля DirectPortsIO для остальных функций. Все функции модуля DirectPortsIO используют в этом случае подключенные варианты *OutByte*, *InByte* и *LastErrorCode*. Устанавливает тип используемого ввода/вывода: *ioEmulator*.

```
procedure SetExternalProcedures(
              aInB:TInB_Func;
              aOutB:TOutB_Proc;
              aOutBBuf:TOutByteBuffer_Func;
              aInBBuf:TInByteBuffer_Func;
              aWait:TWait_Func;
              aLazyWait:TLazyWait_Func;
              aOutByteAndWait:TOutByteAndWait_Func;
              aMultifunctionIO:TMultifunctionIO_Func;
              aAbortState:TAbortState_Func;
              aAbortSet:TAbortOperation_Func;
              aLastError:TLastError_Func); register;
```

Устанавливает внешнее подключение для всех процедур модуля. Устанавливает тип используемого ввода/вывода: *ioEmulator*.

4. КРАТКОЕ ОПИСАНИЕ КОДА ДРАЙВЕРА

4.2. Описание глобальных данных драйвера

Все структуры данных и заголовки основных процедур описаны в файле «MI1201.h». Глобальные данные драйвера размещаются в памяти и инициализируются при вызове системой процедуры *GpdAddDevice* и, частично, иницциализируются при запуске драйвера, см. *GpdStartDevice*. Глобальные данные доступны при каждом вызове драйвера, см. *GpdDispatch* посредством вызова

```
NTSTATUS
GpdDispatch(IN PDEVICE_OBJECT pDO, IN PIRP pIrp)
{  PLOCAL_DEVICE_INFO pLDI;

pLDI = (PLOCAL_DEVICE_INFO)pDO->DeviceExtension;    // Получить указатель на данные драйвера
...
```

Структура данных драйвера *LOCAL_DEVICE_INFO* определена так

```
typedef struct _LOCAL_DEVICE_INFO {
    PDEVICE_OBJECT    DeviceObject;          // Ссылка на device object.
    PDEVICE_OBJECT    NextLowerDriver;      // Сссылка на нижележащий драйвер стека PnP
    IO_REMOVE_LOCK    RemoveLock;            // данные блокировки от удаления драйвера
                                             // при выполнении операций

    BOOLEAN           Started;               // TRUE если драйвер успешно запущен
    BOOLEAN           Removed;               // TRUE если драйвер получил запрос на удаление
                                             // и останавливается

    BOOLEAN           Filler[1];             // bug fix ???
    BOOLEAN           AbortStateBeforeRemoving;     // состояние запрета при начале
                                                    // удаления/остановки драйвера
                                                    // при отмене удаления/остановки
                                                    // состояние запрета восстанвливается отсюда
    KEVENT            AbortEvent;            // запрос на немедленное завершение операций,
                                             // устанавливается в SIGNALED при запрете операций

    // ТРАНСЛИРОВАННЫЕ диапазоны используемых портов (максимально cMaxRangeNumber диапазонов)
    // список предоставляется системой при вызове GpdDispatchPnp с IRP_MN_START_DEVICE,
    // заполняется в GpdStartDevice
    tTransPortInfoArray       TranslatedPortRangesInfo;
    // ИСХОДНЫЕ (ФИЗИЧЕСКИЕ) диапазоны портов (максимально cMaxRangeNumber диапазонов)
    // список предоставляется системой при вызове GpdDispatchPnp с IRP_MN_START_DEVICE,
    // заполняется в GpdStartDevice
    tInitialPortInfoArray     InitialPortRangesInfo;

    tLastReadWriteInfo        LastRead;  // данные портов последней операции чтения
    tLastReadWriteInfo        LastWrite; // данные портов последней операции записи
```

```
    tWaitData                WaitData; // данные операции ожидания (не используются)
    ULONG                    ULFiller[2];          // на всякий случай
} LOCAL_DEVICE_INFO, *PLOCAL_DEVICE_INFO;
```

Другие типы данных из *LOCAL_DEVICE_INFO* определены так:

```
typedef struct _LastPortInfo {
       ULONG   PortType; // тип последнего порта с которым проводились операция
       ULONG   Port;     // последний порт с которым проводились операция
} tLastPortInfo, *ptLastPortInfo;

typedef PORT_DATA tTranslatedPortInfo;
typedef tTranslatedPortInfo *ptTranslatedPortInfo;

typedef struct _LastReadWriteInfo {
       tLastPortInfo                Initial;       // исходный порт
       tTranslatedPortInfo          Translated;    // транслированный порт
       FAST_MUTEX                   LockMutex;     // запрет одновременного доступа
} tLastReadWriteInfo, *ptLastReadWriteInfo;

typedef struct _tWaitData {
    ULONG             ULFiller;              // на всякий случай
} tWaitData, *ptWaitData;
```

4.4. Описание основных внутренних процедур драйвера

Все процедуры драйвера можно подразделить:

1) Системные вызовы, т.е. процедуры необходимые для взаимодействия драйвера с ОС.
2) Специальные, т.е. процедуры необходимые для обеспечения функционирования драйвера и выполнения команд.

К первому типу относятся:

DriverEntry — загрузка драйвера в память;

и процедуры устанавливаемые при выполнении *DriverEntry*

```
    DriverObject->MajorFunction[IRP_MJ_CREATE]          = GpdDispatch;
    DriverObject->MajorFunction[IRP_MJ_CLOSE]           = GpdDispatch;
    DriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL]  = GpdDispatch;
    DriverObject->DriverUnload                          = GpdUnload;
    DriverObject->MajorFunction[IRP_MJ_PNP]             = GpdDispatchPnp;
    DriverObject->MajorFunction[IRP_MJ_POWER]           = GpdDispatchPower;
    DriverObject->MajorFunction[IRP_MJ_SYSTEM_CONTROL]  = GpdDispatchSystemControl;
    DriverObject->DriverExtension->AddDevice            = GpdAddDevice;
```

вообще говоря, все эти функции может выполнять и одна процедура, см. например, *GpdDispatch*, который обрабатывает три вызова *IRP_MJ_CREATE*, *IRP_MJ_CLOSE* и *IRP_MJ_DEVICE_CONTROL*. О назначении всех вызовов можно прочитать в документации к DDK.

Специальные вызовы приходят как субфункции вызова *IRP_MJ_-DEVICE_CONTROL*, см. *GpdDispatch*

```
       case IRP_MJ_DEVICE_CONTROL:
              switch (pIrpStack->Parameters.DeviceIoControl.IoControlCode)
                     case IOCTL_GPD_WAIT_ON_PORT:
```

```
                    Status = GpdIoctlWaitPort(pLDI, pIrp, pIrpStack);
                break;
...
```

в процедуре *GpdDispatch*, в соответствии с кодом операции, вызывается процедура обработки, например, *GpdIoctlWaitPort* — выполняет запрос на ожидание.

Заголовки функций и описание специальных функций приведены комментариями в файлах «MI1201.c», «MI1201.h», «FuncMi1201.c», «FuncMi1201.h», «MI1201Get_SetControlInfo.c», «MI1201Get_SetControlInfo.h».

ЗАКЛЮЧЕНИЕ

Таким образом создан драйвер обеспечивающий надежное функционирование программ управления масс-спектрометром МИ-1201 АГМ в среде наиболее стабильных и защищенных

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Драйвер Windows NT для МИ 1201-AGM: Инструкция по эксплуатации. УГТУ, 2000.