L-CARD

Устройства для мобильных систем

E14-140

Внешний модуль АЦП/ЦАП/ТТЛ на шину **USB 1.1**

Библиотека *Lusbapi 3.3*. Windows

Руководство программиста

Москва. Апрель 2025 г.

Ревизия документа A8

ООО *«Л КАРД»:*

117105, г. Москва, Варшавское шоссе, д. 5, корп. 4, стр. 2.
тел. (495) 785-95-25
факс (495) 785-95-14

Адреса в Интернет:

WWW: *www.lcard.ru*
FTP: *ftp.lcard.ru*

E-Mail:

Общие вопросы:	*lcard@lcard.ru*
Отдел продаж:	*sale@lcard.ru*
Техническая поддержка:	*support@lcard.ru*
Отдел кадров:	*job@lcard.ru*

Оглавление

1. Введение

Данное описание предназначено для пользователей, собирающихся разрабатывать свои собственные приложения в операционной среде *Windows'XP(SP3)/7(SP1)/8.x/10/11* для работы с модулями *E14-140*. Предварительно настоятельно рекомендуется внимательно ознакомиться с *"E14-140. Руководство пользователя"*, где можно найти достаточно подробную информацию по подключению входных сигналов, распиновке внешних разъёмов, о характерных неисправностях и т.п.

В качестве базового языка при написании штатного программного обеспечения нами был выбран язык C++, а конкретнее, старый надёжный **Borland C++ 5.02**. Для модуля *E14-140* фирма ООО "Л Кард" поставляет **USB** драйвер устройства, готовую динамически подключаемую библиотеку `Lusbapi`, а также целый ряд законченных примеров. Причём библиотека и все примеры поставляются вместе с исходными текстами, снабжёнными достаточно подробными комментариями. Библиотека `Lusbapi` позволяет использовать практически все возможности модуля, не вдаваясь в тонкости их низкоуровневого программирования. Если же пользователь все-таки собирается программировать модуль *E14-140* на низком уровне, то библиотека `Lusbapi` может быть использована в качестве законченного и отлаженного руководства, на основе которого можно реализовывать свои собственные алгоритмы.

Модуль *E14-140* разрабатывался с главной целью – обеспечить надёжный *потоковый* ввод в компьютер аналоговой информации. Для этого библиотека `Lusbapi` содержит целый ряд штатных функций, позволяющих организовывать многоканальный *непрерывный потоковый* сбор аналоговых данных на частотах АЦП вплоть до:

- 100 *кГц* для модуля *E14-140 (Rev.'A')*;
- 200 *кГц* для модуля *E14-140-M (Rev.'B' и выше)*.

Также при вводе данных можно использовать определённый вид синхронизации: цифровую или аналоговую. Кроме того, библиотека содержит также ряд функций, позволяющий осуществлять ввод-вывод аналоговой и цифровой информации в однократном, и поэтому достаточно медленном, режиме. Модуль *E14-140 (Rev.'B' и выше)* позволяет организовывать потоковый вывод данных на ЦАП с частотой до 200 *кГц* на канал. Мы надеемся, что описываемая ниже библиотека `Lusbapi` упростит и ускорит написание Ваших собственных приложений.

Полный пакет штатного программного обеспечения модуля *E14-140* для работы в среде *Windows'XP(SP3)/7(SP1)/8.x/10/11* находится на прилагаемом к модулю фирменном CD-ROM'e в директории `\USB\Lusbapi`. **!!!ВНИМАНИЕ!!!** Далее по тексту данного описания все директории, касающиеся непосредственно модуля *E14-140*, указаны относительно неё. Также весь пакет штатного программного обеспечения можно скачать с нашего сайта *www.lcard.ru* из раздела *"Библиотека файлов"*. Там из подраздела *"Программное обеспечение под Windows"* следует выбрать самораспаковывающийся архив `lusbapi`***XY***`.exe`, где ***X.Y*** обозначает номер версии программного обеспечения. На момент написания данного руководства последняя библиотека *Lusbapi* имеет версию **3.3**, а содержащий её архив называется *lusbapi33.exe*.

2. Общие сведения

2.1. Что нового?

Как правило, в данном параграфе будут приводиться только основные изменения как аппаратного, так и программного характера. За более подробной информацией следует обращать к:

- *"E14-140. Руководство пользователя"*;
- *"E14-140. Библиотека Lusbapi. История дополнений и изменений"*

2.1.1. Библиотека Lusbapi 3.3

В июне 2009 г. начато производство новой ревизии модуля *E14-140-M (Rev.'B')*. Данная модификация представляет собой продукт основательной аппаратной модернизации *E14-140 (Rev.'A')*. Библиотека `Lusbapi` версии **3.3** призвана осуществлять полную поддержку всех новых функциональных возможностей и свойств, появившихся у модуля *E14-140-M (Rev.'B')*. С точки зрения программиста отметим лишь следующие основные отличия от предыдущей ревизии модуля:

- В качестве исполнительного устройства на модуле установлен ARM-микроконтроллер *AT91SAM7S256* от фирмы *Atmel Corporation*. Теперь конечный пользователь может программировать его и реализовывать свои собственные алгоритмы на уровне микроконтроллера модуля;
- Максимальная частота преобразования АЦП увеличена до 200 *кГц*;
- Разрядность ЦАП увеличена до 16ти бит (ранее было 12 бит);
- Добавлен потоковый режим работы ЦАП с частотой до 200 *кГц*;
- Введены расширенные возможности синхронизации при работе с АЦП.

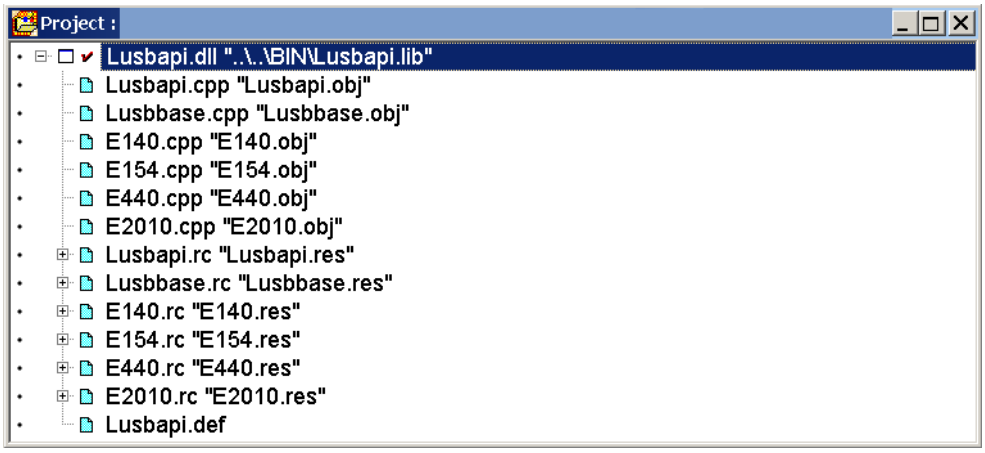
2.2. Подключение модуля E14-140 к компьютеру

Все подробности по процедуре аппаратного подключения модуля *E14-140* к компьютеру пользователя и надлежащей установке **USB** драйверов можно найти в *"E14-140. Руководство пользователя, § 3 "Инсталляция и настройка"*.

Стоит особо подчеркнуть, что, начиная с версии **3.2,** в библиотеке `Lusbapi` изменился основной файл **USB** драйвера. Теперь он называется **`Ldevusbu.sys`** вместо бывшего ранее **`Ldevusb.sys`**. Т.о. при переходе со старых версий **Lusbapi** на более новую версию **3.2** и выше, конечному пользователю следует через *"Device Manager*" ("*Диспетчер устройств*") переключить модуль *E14-140* на работу с новым **USB** драйвером.

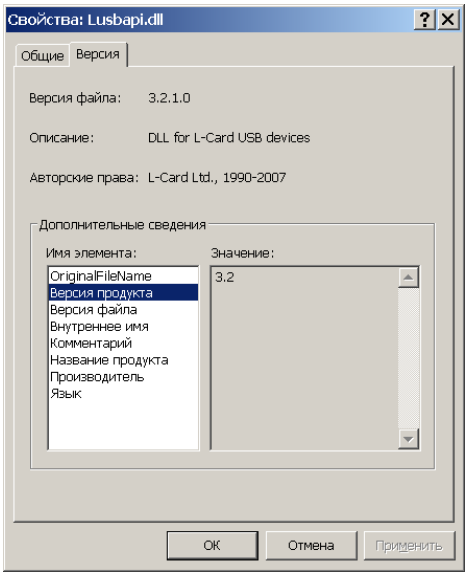
2.3. Библиотека Lusbapi

Штатная библиотека `Lusbapi` написана с использованием весьма доступного языка программирования **Borland C++ 5.02**. Кроме модуля *E14-140* в библиотеке также осуществлена поддержка модулей типа *E14-440*, *E-154* и *E20-10*. Общий вид проекта библиотеки `Lusbapi` в интегрированной среде разработки **Borland C++ 5.02** представлен на рисунке ниже:



Собственно, сама библиотека содержит всего две экспортируемые функции, одна из которых сама библиотека содержит всего две экспортируемые функции, одна из которых *CreateLInstance()* возвращает указатель на интерфейс модуля *E14-140*. В дальнейшем, используя этот указатель, можно осуществлять доступ ко всем интерфейсным функциям DLL библиотеки (см. исходные тексты примеров). **!!!Внимание!!!** Все интерфейсные функции, кроме *ReadData()* строго говоря, не обеспечивают "*потокобезопасную*" работу библиотеки. Поэтому, во избежание недоразумений, в многопоточных приложениях пользователь должен сам организовывать, если необходимо, корректную синхронизацию вызовов интерфейсных функций в различных потоках (используя, например, критические участки, мьютексы и т.д.).

В сам файл библиотеки `Lusbapi.dll` включена информация о текущей версии DLL. Для получения в Вашем приложении сведений о данной версии можно использовать вторую из экспортируемых функций из штатной библиотеки: *GetDllVersion()*. Кроме того, оперативно выявить текущую версию библиотеки можно, используя штатные возможности *Windows*. Например, в '*Windows Explorer*' щелкните правой кнопкой мышки над файлом DLL библиотеки `Lusbapi.dll`. Во всплывшем на экране монитора меню следует выбрать опцию '*Properties*' ('*Свойства*'), после чего на появившейся панели выбрать закладку '*Version*' ('*Версия*'). На этой закладке в строчке '*File version*' ('*Версия файла*') можно без труда прочитать текущий номер версии библиотеки. Выглядит это примерно так:



Сам файл штатной библиотеки Lusbapi.dll расположен на фирменном CD–ROM в директории \DLL\BIN. Её исходные тексты можно найти в директории \DLL\Source\Lusbapi. Заголовочные файлы хранятся в директории \DLL\Include, а библиотеки импорта и модули объявления для различных сред разработки можно найти в директории \DLL\Lib.

Тексты законченных примеров применения интерфейсных функций из штатной DLL библиотеки для различных сред разработки приложений можно найти в следующих директориях:

- *E14-140*\Examples\Borland C++ 5.02;
- *E14-140*\Examples\Borland C++ Builder 5.0;
- *E14-140*\Examples\Borland Delphi 6.0;
- *E14-140*\Examples\MicroSoft Visual C++ 6.0.

Например, для получения возможности вызова интерфейсных функций в Вашем проекте на **Borland C++** Вам необходимо следующее:

- создать файл проектов (например, для **Borland C++ 5.02**, test.ide);
- добавить файл библиотеки импорта \DLL\Lib\Borland\LUSBAPI.LIB;
- создать и добавить в проект Ваш файл с будущей программой (например, test.cpp);
- включить в начало вашего файла заголовочный файл #include "LUSBAPI.H", содержащий описание интерфейса модуля *E14-140*;
- в принципе желательно сравнить версию используемой библиотеки Lusbapi.dll с версией текущего программного обеспечения, используя для этого функцию *GetDllVersion()*;
- вызвать функцию *CreateLInstance()* для получения указателя на интерфейс модуля;
- в общем-то, ВСЁ! Теперь Вы можете писать свою программу и в любом месте, используя полученный указатель, вызывать соответствующие интерфейсные функции из штатной DLL библиотеки Lusbapi.dll.

Сторонникам диалекта **Microsoft Visual C++** можно порекомендовать два способа подключения штатной DLL библиотеки к своему приложению:

1. Динамическая загрузка библиотеки Lusbapi на этапе выполнения приложения. Подробности смотри в исходных текстах примера из директории *E14-140*\Examples\MicroSoft Visual C++ 6.0\DynLoad.
2. При статической компоновке библиотеки Lusbapi в Вашем проекте следует использовать файл библиотеки импорта LUSBAPI.LIB из директории \DLL\Lib\MicroSoft.

При работе с модулем типа *E14-140* в среде **Borland Delphi** рекомендуется применять модуль объявлений LUSBAPI.PAS, расположенный в директории \DLL\Lib\Delphi. Также вместо исходного модуля объявлений вполне можно задействовать уже откомпилированную версию LUSBAPI.DCU.

2.4. Микроконтроллер модуля

В зависимости от ревизии на модуле E14-140 в качестве исполнительного устройства используются следующие микроконтроллеры (MCU):

- *AVR ATmega8515* от фирмы *Atmel Corporation* для *E14-140 (Rev.'A')*;
- *ARM AT91SAM7S256* от фирмы *Atmel Corporation* для *E14-140-M (Rev.'B')*

MCU отвечает за корректное функционирование **USB** интерфейса модуля, а также разбирает все пользовательские команды, поступающие из компьютера и задающие различные режимы работы модуля. Особенностью программного софта, заложенного в основу работы MCU, является возможность его обновления. Т.е. фирменная программа, '*заливается*' в MCU на этапе наладки модуля E14-140 в ООО "Л Кард". Но конечный пользователь имеет возможность её перепрошивки чисто программным способом без наличия специального кабеля, что является крайне удобным при обновлении версий программы. Последнюю версию фирменной программы MCU всегда можно скачать с нашего сайта *www.lcard.ru* из раздела *"Библиотека файлов"*. Там из подраздела *"Firmware и BIOS"* можно выбрать следующие архивы:

- e140fw**XY**.zip для модуля *E14-140 (Rev.'A')*, где **X.Y** означает номер версии основной программы MCU модуля. На момент написания данного руководства этот архив имеет имя *e140fw26.zip*.
- e140flash_v**XYY**.zip для модуля *E14-140-M (Rev.'B')*, где **X.YY** означает номер версии основной программы MCU модуля. На момент написания данного руководства этот архив имеет имя *e140flash_v305.zip*.

2.5. Возможные проблемы при работе с модулем

1. Перед началом работы со штатным программным обеспечением модуля *E14-140*, во избежание непредсказуемого его поведения, настоятельно рекомендуется установить драйвера для чипсета используемой материнской платы компьютера. В особенности это касается чипсетов не от ***Intel***: ***VIA***, ***SIS***, ***nVidia***, ***AMD+ATI*** и т.д. Обычно эти драйвера можно найти на фирменном CD-ROM, который поставляется вместе с материнской платой. Также их можно скачать из Интернета с сайта производителя.

2. Компьютеры, у которых материнская плата создана на основе чипсета от фирмы ***SIS*** (*Silicon Integrated System Corporation*), ***AMD+ATI*** (*Advanced Micro Devices, Inc.*) или ***nVidia*** (*NVIDIA Corporation*) не совсем корректно работают в среде *Windows'XP(SP3)/7(SP1)/8.x/10/11*. Это проявляется при запросах с большим кол-вом данных в интерфейсных функциях *ReadData()*. Например, при вызове этой функции с параметром *NumberOfWordsToRead* = 1024*1024 операционная система *Windows* вполне может, что называется, '*наглухо*' зависнуть вплоть до появления BSOD (Blue Screen Of Death). Решение данной проблемы лежит в русле уменьшения значения *NumberOfWordsToRead*. Причём величина *NumberOfWordsToRead*, при которой всё начинает нормально работать, зависит от конкретного экземпляра компьютера. Так что следует попробовать просто поварьировать величину параметра *NumberOfWordsToRead*.

3. Используемые термины и форматы данных

3.1. Термины

Название	Смысл
AdcRate	Частота работы АЦП в *кГц*
InterKadrDelay	Межкадровая задержка в *мс*
KadrRate	Частота кадра отсчётов в *кГц*.
Buffer	Указатель на целочисленный массив для данных
Npoints	Число отсчетов ввода
AdcChannel	Логический номер аналогового канала АЦП
ControlTable	Управляющая таблица, содержащая целочисленный массив с логическими номерами каналов для последовательного циклического ввода данных с АЦП
ControlTableLength	Длина управляющей таблицы

3.2. Форматы данных

3.2.1. Формат слова данных с АЦП

Данные, считанные с 14[ти] битного АЦП модуля *E14-140*, представляются в формате знакового целого двухбайтного числа от -8192 до 8191. Эти "*сырые*" отсчёты с АЦП рекомендуется откорректировать с помощью калибровочных коэффициентов, которые хранятся в модуле и доступны с помощью штатной функции *GET_MODULE_DESCRIPTION()*. Процедура корректировки данных АЦП достаточно подробно описана в *§ 4.5.1. "Корректировка данных АЦП"*. После этой процедуры взаимосвязь откорректированного кода АЦП с входным напряжением модуля можно отобразить следующим образом:

Таблица 1. Соответствие откорректированного кода АЦП входному напряжению

Диапазон, В	Код АЦП	Входное напряжение, В
±10.0; ±2.5; ±0.625; ±0.15625	+8000	+10.0; +2.5; +0.625; +0.1562
	0	0
	-8000	–10.0; –2.5; –0.625; -0.1562

3.2.2. Формат слова данных для ЦАП

3.2.2.1. Формат слова данных ЦАП для модуля E14-140 (Rev.'A')

На модуле *E14-140 (Rev.'A')* по желанию пользователя может быть установлена микросхема 2[х] канального 12[ти] битного ЦАП. Для установления какого-нибудь напряжения на выходе ЦАП в модуль *E14-140 (Rev.'A')* необходимо передать 16[ти] битное слово данных, **не все** биты которого являются значимыми. Формат этого слова данных приведен в следующей таблице:

Таблица 2. Формат слова данных ЦАП для модуля E14-140 (Rev.'A')

Модуль	Номер бита	Назначение
E14-140 (Rev.'A')	<11..0>	12[ти] битный код ЦАП
	12	Выбор номера канала ЦАП: ✓ '0' – первый канал; ✓ '1' – второй канал.
	<15..13>	Не используются

Собственно сам код ЦАП перед отправкой его в модуль рекомендуется откорректировать. Корректировочные коэффициенты хранятся в модуле и доступны с помощью штатной функции *GET_MODULE_DESCRIPTION()*. Процедура корректировки данных ЦАП достаточно подробно описана в *§ 4.6.1. "Корректировка данных ЦАП"*. После этой процедуры откорректированный код, выдаваемый модулем на 12[ти] битный ЦАП, связан с устанавливаемым на внешнем разъёме напряжением в соответствии со следующей таблицей:

Таблица 3. Соответствие откорректированного кода ЦАП выходному напряжению

Модуль	Код ЦАП	Напряжение, В
E14-140	+2047	+5.0
	0	0
	–2048	–5.0

3.2.2.2. Формат слова данных ЦАП для модуля E14-140 (Rev.'B' и выше)

На модуле *E14-140 (Rev.'B'* и выше*)* по желанию пользователя может быть установлен 2^{x} канальный $16^{ти}$ битный ЦАП. Для установления какого-нибудь напряжения на выходе ЦАП в модуль *E14-140 (Rev.'B'* и выше*)* необходимо передать $16^{ти}$ битное слово данных, **все** биты которого являются значимыми.

3.2.3. Логический номер канала АЦП

Для управления работой входного аналогового каскада модуля *E14-140* был введён такой параметр как $8^{ми}$ битный *логический номер канала АЦП*, фактически управляющее слово для АЦП. Именно массив логических номеров каналов АЦП, образующих управляющую таблицу **ControlTable**, задает циклическую последовательность работы АЦП при вводе данных. В состав логического номера канала входят несколько важных параметров, задающих различные режимы функционирования АЦП модуля:

- физический номер аналогового канала;
- управление включением режима калибровки нуля, т.е. при этом вход каскада с программируемым коэффициентом усиления (PGA) просто заземляется;
- тип подключения входных каскадов – 16 дифференциальных входных аналоговых каналов или 32 входных канала с общей землёй;
- индекс коэффициента усиления, т.е. для каждого логического канала можно установить свой индивидуальный коэффициент усиления (диапазон входного напряжения).

Битовый формат логического номера канала АЦП представлен в таблице ниже:

Таблица 4. Формат логического номера канала.

Номер бита	Обозначение	Функциональное назначение
0	**MA0**	$0^{ой}$ бит номера канала
1	**MA1**	$1^{ый}$ бит номера канала
2	**MA2**	$2^{ой}$ бит номера канала
3	**MA3**	$3^{ий}$ бит номера канала
4	**MA4**	Калибровка нуля/$4^{ый}$ бит номера канала
5	**MA5**	16 диф./32 общ.
6	**GS0**	$0^{ой}$ бит коэффициента усиления (см. *Таблицу 5*)
7	**GS1**	$1^{ый}$ бит коэффициента усиления (см. *Таблицу 5*)

Если **MA5**=0 и **MA4**=0, то **MA0÷MA3** – номер выбранной дифференциальной пары входов.
Если **MA5**=0 и **MA4**=1, то калибровка нуля, т.е. измерение собственного напряжения нуля.
Если **MA5**=1, то **MA0÷MA4** – номер выбранного входа с общей землей (X1–>Вход1, X2–> Вход2, **…**, Y1–> Вход17,**…**, Y16–> Вход32).

Например, логический номер канала для модуля *E14-140* равный 0x2 будет означать дифференциальный режим работы $3^{его}$ канала с единичным усилением, 0x82 – тот же канал с усилением равным 16. Если же логический номер равен 0x10 или 0x14, то вход каскада PGA будет просто заземлен (именно PGA, а не входы указанных каналов коммутатора).

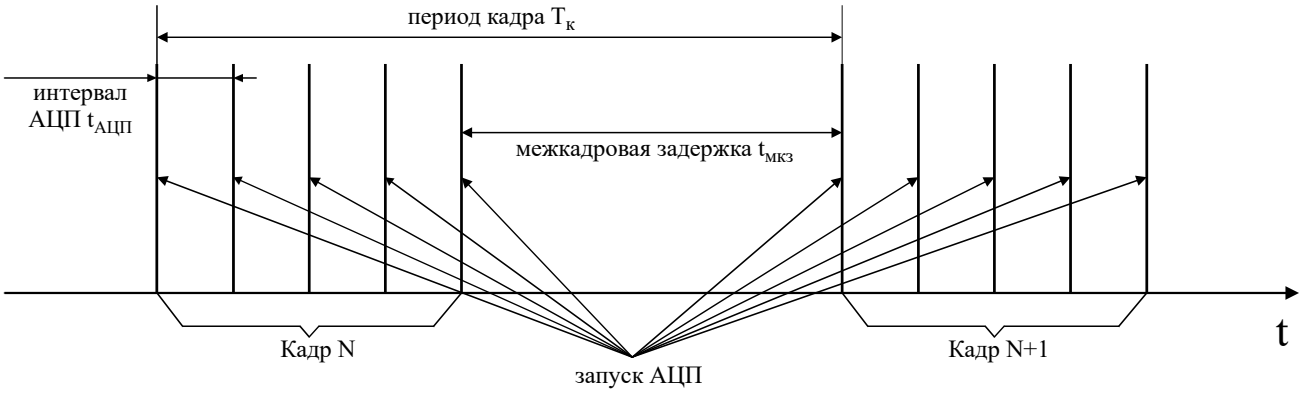
Таблица 5. Коэффициент усиления (биты GS0 и GS1)

Модуль	Бит **GS1**	Бит **GS0**	Усиление	Диапазон, В
E14-140	0	0	1	±10.0
	0	1	4	±2.5
	1	0	16	±0.625
	1	1	64	±0.15625

Например, если в логическом номере канале АЦП установить битовое поле **GS1÷GS0** равным 2, то тем самым будет выбран коэффициент усиления 16, а диапазон входного напряжения при этом составит ±0.625 В.

3.2.4. Формат кадра отсчетов

Под кадром подразумевается последовательность отсчетов с логических каналов, начиная от **ControlTable[0]** до **ControlTable[ControlTableLength–1]**, где **ControlTable** – управляющая таблица (массив логических каналов), хранящаяся в DSP модуля, а **ControlTableLength** определяет размер (длину) этой таблицы. Загрузить нужную управляющую таблицу в сигнальный процессор модуля можно с помощью интерфейсной функции ***SET_ADC_PARS()*** (см. *§ 4.5.4. "Установка параметров работы АЦП"*). Временные параметры кадра для **ControlTableLength = 5** приведены на следующем рисунке:



где $\mathbf{T_k}$ – временной интервал между соседними кадрами (фактически частота опроса фиксированного логического номера канала **KadrRate**), $\mathbf{t_{мкз}}$= **InterKadrDelay** – временной интервал между последним отсчетом текущего кадра и первым отсчетом следующего, $\mathbf{t_{АЦП}}$ – интервал запуска АЦП или межканальная задержка. Тогда $1/\mathbf{t_{АЦП}}$= **AdcRate** – частота работы АЦП или оцифровки данных, а величина $\mathbf{t_{мкз}}$ не может принимать значения меньшие, чем $\mathbf{t_{АЦП}}$. Если размер кадра, т.е. число отсчетов с АЦП в кадре, равен **ControlTableLength**, то все эти временные параметры можно связать следующей формулой:

$$\mathbf{T_k} = 1/\mathbf{KadrRate} = (\mathbf{ControlTableLength}–1) * \mathbf{t_{АЦП}} + \mathbf{t_{мкз}},$$

или

$$\mathbf{T_k} = 1/\mathbf{KadrRate} = (\mathbf{ControlTableLength}–1)/\mathbf{AdcRate} + \mathbf{InterKadrDelay}.$$

Временные параметры **AdcRate** и **InterKadrDelay** используются в интерфейсной функции *SET_ADC_PARS()* при задании необходимого режима работы АЦП.

4. Описание библиотеки Lusbapi

В настоящем разделе приведены достаточно подробные описания констант, структур и интерфейсных функций, входящих в состав штатной библиотеки Lusbapi для модуля *E14-140*.

4.1. Общие принципы работы с модулем

Целью штатной библиотеки Lusbapi, поставляемой с модулем *E14-140*, является предоставление достаточно наглядного и удобного программного интерфейса при работе с данным устройством. Библиотека содержит в себе определенный набор функций, с помощью которых Вы можете реализовывать многие стандартные алгоритмы ввода/вывода данных в/из модуля.

Перед началом работы с библиотекой в пользовательской программе необходимо сделать следующие объявление (как минимум):

```
ILE140 *pModule;                    // указатель на интерфейс модуля E14-140
MODULE_DESCRIPTION_E140 md;         // структура служебной информации о модуле
```

Первым делом с помощью функции *GetDllVersion()* следует проверить версии используемой библиотеки Lusbapi и текущего программного обеспечения.

Если версии совпадают, то в Вашем приложении необходимо получить указатель на интерфейс модуля, вызвав функцию *CreateLInstance()*. В дальнейшем для доступа ко всем интерфейсным функциям модуля необходимо применять именно этот указатель (см. *пример ниже*).

После этого, используя уже полученный указатель на интерфейс модуля, следует проинициализировать доступ к виртуальному слоту, к которому подключён модуль *E14-140*. Для этого предусмотрена интерфейсная функция *OpenLDevice()*. Если нет ошибки при выполнении этой функции, то можно быть уверенным, что устройство типа *E14-140* обнаружено в выбранном виртуальном слоте.

Теперь, в принципе, можно переходить к этапу чтения служебной информации о модуле, но иногда нелишне бывает определиться с текущей скоростью работы используемого **USB** порта. Для этого предназначена интерфейсная функция *GetUsbSpeed()*. Модуль совместно с **USB** портом должен работать в так называемом Full-Speed Mode. Это будет соответствовать пропускной способности модуля по **USB** шины порядка 1 Мбайт/с.

На следующем этапе следует прочитать служебную информацию о модуле. Она требуется при работе с некоторыми интерфейсными функциями библиотеки Lusbapi. Интерфейсная функция *GET_MODULE_DESCRIPTION()* как раз и предназначена для этой цели. Если функция не вернула ошибку, то это означает, что вся служебная информация о модуле успешно получена и можно продолжать работу.

В общем-то, предварительный этап работы с модулем *E14-140* можно считать успешно завершённым. Теперь можно спокойно управлять всей доступной периферией на модуле с помощью соответствующих интерфейсных функций библиотеки Lusbapi и организовывать различные режимы работы модуля. Например, такие режимы как:

- непрерывный *потоковый* сбор с АЦП с синхронизацией ввода данных;
- однократный, и потому достаточно медленный, вывод данных на двуканальный ЦАП;
- однократная работа с входными и выходными цифровыми линиями;
- работа с пользовательским ППЗУ модуля и т.д.

В качестве примера приведем исходный текст, а вернее сказать ‘*скелет*’, консольной программы для работы с *E14-140*, предполагая использование Lusbapi версии не ниже 3.0:

```
#include <stdlib.h>
#include <stdio.h>
#include "Lusbapi.h"              // заголовочный файл библиотеки Lusbapi

ILE140 *pModule;                  // указатель на интерфейс модуля
MODULE_DESCRIPTION_E140 md;       // структура с информацией о модуле
char ModuleName[7];               // название модуля
BYTE UsbSpeed;                    // скорость работы шины USB
```

```
int main(void)
{
   // проверим версию DLL библиотеки
   if(GetDllVersion() != CURRENT_VERSION_LUSBAPI)
   {
      printf("Неправильная версия Dll!");
      return 1;                  //выйдем из программы с ошибкой
   }
   // получим указатель на интерфейс модуля
   pModule = static_cast<ILE140 *>(CreateLInstance("e140"));
   if(!pModule)
   {
      printf("Не могу получить указатель на интерфейс");
      return 1;                  //выйдем из программы с ошибкой
   }
   // попробуем обнаружить какой-нибудь модуль
   // в нулевом виртуальном слоте
   if(!pModule->OpenLDevice(0))
   {
      printf("Не могу получить доступ к модулю!");
      return 1;                  //выйдем из программы с ошибкой
   }
   // попробуем получить скорость работы шины USB
   if(!pModule->GetUsbSpeed(&UsbSpeed))
   {
      printf("Не могу узнать скорость работы USB!\n");
      return 1;                  //выйдем из программы с ошибкой
   }
   // прочитаем название модуля в нулевом виртуальном слоте
   if(!pModule->GetModuleName(ModuleName))
   {
      printf("Не могу прочитать название модуля!\n");
      return 1;                  //выйдем из программы с ошибкой
   }
   // проверим: этот модуль - 'E14-140'?
   if(strcmp(ModuleName, "E140"))
   {
      printf(" В нулевом виртуальном слоте не 'E14-140'\n");
      return 1;                  //выйдем из программы с ошибкой
   }
   // попробуем прочитать информацию о модуле
   if(!pModule->GET_MODULE_DESCRIPTION(&md))
   {
      printf("Не выполнена функция GET_MODULE_DESCRIPTION ()!");
      return 1;                  //выйдем из программы с ошибкой
   }

   printf("Модуль E14-140 (серийный номер %s) полностью готов к\
                                    работе!", md.Module.SerialNumber);
   // далее можно располагать функции для непосредственного
   // управления модулем

   . . . . . .
```

```
    // завершим работу с модулем
    if(!pModule->ReleaseLInstance())
    {
       printf("Не выполнена функция ReleaseLInstance()!");
       return 1;                  //выйдем из программы с ошибкой
    }

    // выйдем из программы
    return 0;
}
```

4.2. Константы

Приведённые ниже основные константы настоятельно рекомендуется использовать в исходных текстах приложения при работе с модулем *E14-140*. Это весьма повышает *читаемость* и *понимаемаемость* исходных текстов, а также значительно облегчает сопровождение программ. Рассматриваемые константы расположены в файле `\DLL\Include\Lusbapi.h`.

1. Модулю *E14-140* для работы АЦП требуется наличие аппаратных *тактовых импульсов*. Данные константы определяют источник формирования этих импульсов. Местом использования этих констант, как правило, является поле *ClkSource* структуры *ADC_PARS_E140*.

Константа	Значение	Назначение
INT_ADC_CLOCK_E140	0	***Внутренние тактовые импульсы АЦП.*** При этом *тактовые импульсы* аппаратно генерируются при помощи таймера, встроенного в микроконтроллер модуля. Так же, в этом режиме можно настроить модуль на трансляцию *тактовых импульсов* АЦП на линию ввода-вывода ***SYN*** цифрового разъёма модуля. Подробности смотри поле *EnableClkOutput* структуры *ADC_PARS_E140*.
EXT_ADC_CLOCK_E140	1	***Внешние тактовые импульсы АЦП.*** В этом режиме используется внешний источник *тактовых импульсов* АЦП, который подключается к линии ввода-вывода ***SYN*** цифрового разъёма модуля.
INVALID_ADC_CLOCK_E140	2	Неправильный тип источника *тактовых импульсов* АЦП.

2. Модуль *E14-140* позволяет осуществлять трансляцию *тактовых импульсов* АЦП на линию ввода-вывода ***SYN*** цифрового разъёма. Данное поле имеет значение только при внутренних *тактовых импульсах* АЦП, когда поле *ClkSource* структуры *ADC_PARS_E140* равно нулю. Иначе значение данного поля игнорируется. Данные константы определяют, нужна ли трансляция *тактовых импульсов* АЦП на цифровой разъём модуля. Местом использования этих констант, как правило, является поле *EnableClkOutput* структуры *ADC_PARS_E140*.

Константа	Значение	Назначение
ADC_CLOCK_TRANS_DISABLED_E140	0	***Запретить трансляцию тактовых импульсов АЦП.*** При этом линия ввода-вывода ***SYN*** цифрового разъёма будет находиться в третьем, *высокоимпедансном*, состоянии.
ADC_CLOCK_TRANS_ENABLED_E140	1	***Разрешить трансляцию тактовых импульсов АЦП.*** При этом *тактовые импульсы* АЦП, генерируемые таймером микроконтроллера, будут транслироваться на линию ввода-вывода ***SYN*** цифрового разъёма модуля.
INVALID_ADC_CLOCK_TRANS_E140	2	Неправильный значение трансляции *тактовых импульсов* АЦП.

3. У модуля *E14-140* есть четыре возможных диапазона входных напряжений, каждый из которых можно задать данными константами. Местом использования этих констант, как правило, являются битовые поля **GS0÷GS1** *логического номера канала АЦП*. Массив этих логических каналов образуют управляющую таблицу поля *ControlTable* структуры *ADC_PARS_E140*.

Константа	Значение	Назначение
ADC_INPUT_RANGE_ 10000mV_E140	0	Данная константа задаёт входной диапазон, равный ±10000 мВ. Также можно эту константу можно использовать в качестве индекса для доступа к первому элементу константного массива *ADC_INPUT_RANGES_E140*.
ADC_INPUT_RANGE_ 2500mV_E140	1	Данная константа задаёт входной диапазон, равный ±2500 мВ. Также можно эту константу можно использовать в качестве индекса для доступа ко второму элементу константного массива *ADC_INPUT_RANGES_E140*.
ADC_INPUT_RANGE_ 625mV_E140	2	Данная константа задаёт входной диапазон, равный ±625 мВ. Также можно эту константу можно использовать в качестве индекса для доступа к третьему элементу константного массива *ADC_INPUT_RANGES_E140*.
ADC_INPUT_RANGE_ 156mV_E140	3	Данная константа задаёт входной диапазон, равный ±156 мВ. Также можно эту константу можно использовать в качестве индекса для доступа к четвёртому элементу константного массива *ADC_INPUT_RANGES_E140*.

4. Модуль *E14-140* в состоянии осуществлять различную синхронизацию ввода данных с АЦП. Данные константы определяют тип требуемой синхронизации. Местом использования этих констант, как правило, является поле *InputMode* структуры *ADC_PARS_E140*.

Константа	Значение	Назначение
NO_SYNC_E140	0	***Отсутствие синхронизации ввода.*** Сбор данных с АЦП модуль начинает непосредственнно после выполнения функции *START_ADC()*.
TTL_START_SYNC_ E140	1	***Цифровая синхронизация начала ввода.*** Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания прихода отрицательного перепада (⎍) у ТТЛ-совместимого одиночного импульса на входе **INT** аналогового разъёма модуля. Длительность этого синхроимпульса должна быть не менее 100 нс. Только после этого модуль приступает к сбору данных.

TTL_KADR_SYNC_E140	2	***Цифровая покадровая синхронизация ввода.*** Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания прихода отрицательного перепада (⎍) у ТТЛ-совместимого одиночного импульса на входе **INT** аналогового разъёма модуля. Длительность этого синхроимпульса должна быть не менее 100 нс. После прихода указанного синхроимпульса модуль собирает отсчеты только одного кадра. Во время ввода кадра данных, вся активность на линии **INT** игнорируется. Только после окончания сбора текущего кадра данных ожидается приход следующего импульса синхронизации и т.д.
ANALOG_SYNC_E140	3	***Аналоговая синхронизация начала ввода***. Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания выполнения условий аналоговой синхронизации. Модуль начинает собирать данные с АЦП только после выполнения заданных соотношений между полученным значением с заданного аналогового синхроканала и заданным пороговым значением.
INVALID_SYNC_E140	4	Неправильный вид синхронизации ввода данных.

5. На модуле *E14-140* по желанию пользователя может быть установлена микросхема двухканального ЦАП. Состояние поля *Dac.Active* структуры служебной информации *MODULE_DESCRIPTION_E140* отражает факт наличия ЦАП на борту модуля.

Константа	Значение	Назначение
DAC_INACCESSIBLED_E140	0	На модуле отсутствует микросхема ЦАП.
DAC_ACCESSIBLED_E140	1	На модуле присутствует микросхема ЦАП.

6. Ревизия модуля *E14-140* отражает определённые конструктивные особенности модуля. Она задаётся одной заглавной латинской буквой и размещается в поле Revision вложенной структуры *MODULE_INFO_LUSBAPI* служебной структуры *MODULE_DESCRIPTION_E2010*. Так, *первая* ревизия модуля обозначается ***'A'***.

Константа	Значение	Назначение
REVISION_A_E140	0	Данная константа может использоваться в качестве индекса для доступа к первому элементу константного массива *REVISIONS_E140*.
REVISION_B_E140	1	Данная константа может использоваться в качестве индекса для доступа ко второму элементу константного массива *REVISIONS_E140*.

7. Различные константы для работы с модулем *E14-140*.

Константа	Значение	Назначение
CURRENT_VERSION_ LUSBAPI	———	Версия используемой библиотеки `Lusbapi`. Как правило, используется совместно с функцией *GetDllVersion()*.
MAX_CONTROL_ TABLE_LENGTH_E140	128	Максимально возможное кол-во логических каналов в управляющей таблице **ControlTable**.
ADC_CALIBR_COEFS_ QUANTITY_E140	4	Кол-во корректировочных коэффициентов для данных АЦП. По одному на каждый входной диапазон. Корректировке подвергаются как смещение, так и масштаб данных АЦП.
ADC_INPUT_RANGES_ QUANTITY_E140	4	Кол-во входных диапазонов.
DAC_CHANNELS_ QUANTITY_E140	2	Кол-во физических каналов ЦАП на модуле (при условии наличия микросхемы ЦАП на модуле).
DAC_CALIBR_COEFS_ QUANTITY_E140	2	Кол-во корректировочных коэффициентов для данных ЦАП. По одному на каждый канал. Корректировке подвергаются как смещение, так и масштаб данных ЦАП.
TTL_LINES_QUANTITY_ E140	16	Кол-во входных и выходных цифровых линий.
USER_FLASH_SIZE_E140	512	Размер пользовательского ППЗУ в байтах
REVISIONS_QUANTITY_ E140	1	Кол-во ревизий (модификаций) модуля.

8. Различные *константные массивы* для работы с модулем *E14-140*.
   a. Массив доступных диапазонов напряжения АЦП в Вольтах:
   ```
   const double
            ADC_INPUT_RANGES_E140[ADC_INPUT_RANGES_QUANTITY_E140] =
   {
  10.0, 10.0/4.0, 10.0/16.0, 10.0/64.0
   };
   ```
   b. Диапазон выходного напряжения ЦАП в Вольтах:
   ```
   const double DAC_OUTPUT_RANGE_E140 = 5.0;
   ```
   c. Ревизия модуля отражает определённые конструктивные особенности модуля. Она задаётся одной заглавной латинской буквой. Например, *первая* ревизия модуля обозначается через букву ***'A'***. Текущая ревизия модуля содержится в поле *Module.Revision* структуры служебной информации *MODULE_DESCRIPTION_E140*. Массив доступных ревизий модуля задаётся следующим образом.
   ```
   const BYTE REVISIONS_E140[REVISIONS_QUANTITY_E140] = { 'A', 'B'};
   ```

4.3. Структуры

В данном разделе приведены основные типы структур, которые применяются в библиотеке `Lusbapi` при работе с модулем *E14-140*.

4.3.1. Структура MODULE_DESCRIPTION_E140

Структура *MODULE_DESCRIPTION_E140* описана в файле `\DLL\Include\Lusbapi.h` и представлена ниже:

```
struct MODULE_DESCRIPTION_E140
{
    MODULE_INFO_LUSBAPI      Module;        // общая информация о модуле
    INTERFACE_INFO_LUSBAPI   Interface;     // информация об интерфейсе
    MCU_INFO_LUSBAPI<VERSION_INFO_LUSBAPI> Mcu;     // информация о MCU
    ADC_INFO_LUSBAPI         Adc;           // информация о АЦП
    DAC_INFO_LUSBAPI         Dac;           // информация о ЦАП
    DIGITAL_IO_INFO_LUSBAPI DigitalIo;      // информация о цифровом вводе-выводе
};
```

В данной структуре представлена самая общая служебная информация об используемом экземпляре модуля *E14-140*. Эта структура используется при работе с интерфейсными функциями *SAVE_MODULE_DESCRIPTION()* и *GET_MODULE_DESCRIPTION()*. В определении этой структуры применяются вспомогательные константы и типы данных, описанные в *Приложении А*.

4.3.2. Структура ADC_PARS_E140

Структура *ADC_PARS_E140* описана в файле `\DLL\Include\Lusbapi.h` и представлена ниже:

```
struct ADC_PARS_E140
{
    WORD ClkSource;              // источник тактовых импульсов АЦП
    WORD EnableClkOutput;        // трансляции тактовых импульсов АЦП
    WORD InputMode;              // режим синхронизации ввода данных с АЦП
    WORD SynchroAdType;          // тип аналоговой синхронизации
    WORD SynchroAdMode;          // режим аналоговой синхронизации
    WORD SynchroAdChannel;       // канал АЦП при аналоговой синхронизации
    SHORT SynchroAdPorog;        // порог срабатывания аналоговой синхронизации
    WORD ChannelsQuantity;       // число активных каналов (размер кадра)
    WORD ControlTable[128];      // управляющая таблица с активными лог. каналами
    double AdcRate;              // частота работы АЦП в кГц
    double InterKadrDelay;       // межкадровая задержка в мс
    double KadrRate;             // частота кадра в кГц
};
```

Перед началом работы с АЦП необходимо заполнить поля данной структуры и передать ее в модуль с помощью интерфейсной функции *SET_ADC_PARS()*. В описании этой функции подробно прокомментированы смысл и назначение всех полей данной структуры. Также при необходимости можно считать из модуля текущие параметры функционирования АЦП, используя интерфейсную функцию *GET_ADC_PARS()*.

4.3.3. Структура DAC_PARS_E140

Структура *DAC_PARS_E140* описана в файле \DLL\Include\Lusbapi.h и представлена ниже:

```
struct DAC_PARS_E140
{
    BYTE SyncWithADC;           // 0 – обычный пуск ЦАП
                                // 1 – синхронизировать с пуском АЦП
    BYTE SetZeroOnStop          // 1 – при остановке потокового вывода установить
                                // на выходе ЦАП 0 В
    double DacRate;             // частота работы ЦАП в кГц
};
```

Перед началом работы с ЦАП необходимо заполнить поля данной структуры и передать ее в модуль с помощью интерфейсной функции *SET_DAC_PARS()*. В описании этой функции подробно прокомментированы смысл и назначения всех полей данной структуры. Также при необходимости можно считать из модуля текущие параметры функционирования ЦАП, используя *GET_DAC_PARS()*.

4.3.4. Структура IO_REQUEST_LUSBAPI

Структура *IO_REQUEST_LUSBAPI* описана в файле \DLL\Include\LusbapiTypes.h и представлена ниже:

```
struct IO_REQUEST_LUSBAPI
{
    SHORT * Buffer;                 // буфер для передаваемых данных
    DWORD   NumberOfWordsToPass;    // кол-во отсчётов, которые требуется передать
    DWORD   NumberOfWordsPassed;    // кол-во реально переданных отсчётов
    OVERLAPPED * Overlapped;  // для синхронного запроса – NULL, а для асинхроннного
                              // запроса – указатель на стандартную WinAPI
                              // структуру типа OVERLAPPED
    DWORD   TimeOut;          // для синхронного запроса – таймаут в мс, а для
                              // асинхронного запроса не используется
};
```

Данная структура используются функциями *ReadData()* и *WriteData()* при организации *потоковых* передач данных между модулем и компьютером. В описании этих функций подробно прокомментированы смысл и назначения полей данной структуры.

4.3.5. Структура LAST_ERROR_INFO_LUSBAPI

Структура *LAST_ERROR_INFO_LUSBAPI* описана в файле \DLL\Include\LusbapiTypes.h и представлена ниже:

```
struct LAST_ERROR_INFO_LUSBAPI
{
    BYTE  ErrorString[256];     // строка с кратким описанием последней ошибки
                                // библиотеки Lusbapi
    DWORD ErrorNumber;          // номер последней ошибки библиотеки Lusbapi
};
```

Данная структура используется функцией *GetLastErrorInfo()* при выявлении ошибок выполнения интерфейсных функций библиотеки Lusbapi.

4.4. Функции общего характера

4.4.1. Получение версии библиотеки

Формат: **DWORD** ***GetDllVersion****(void)*

Назначение:

Данная функция является одной из *двух* экспортируемых из штатной библиотеки `Lusbapi` функцией. Она возвращает текущую версию используемой библиотеки. Формат номера версии следующий:

Битовое поле	Назначение
<31..16>	Старшее слово версии библиотеки
<15..0>	Младшее слово версии библиотеки

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 4.1. "Общие принципы работы с модулем"*.

Передаваемые параметры: нет

Возвращаемое значение: номер версии библиотеки `Lusbapi`.

4.4.2. Получение указателя на интерфейс модуля

Формат: **LPVOID** ***CreateInstance****(const char *DeviceName)* (версия 2.1)
LPVOID ***CreateLInstance****(PCHAR const DeviceName)* (с версии 3.0)

Назначение:

Данная функция должна обязательно вызываться в начале каждой пользовательской программы, работающей с модулями *E14-140*. Она является одной из *двух* экспортируемых из штатной библиотеки `Lusbapi` функцией и возвращает указатель на интерфейс для устройства с названием *DeviceName*. Все последующие интерфейсные функции штатной библиотеки вызываются именно через этот возвращаемый указатель.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 4.1. "Общие принципы работы с модулем"*.

Передаваемые параметры:

- *DeviceName* – строка с названием устройства (для данного модуля это – “E140”) .

Возвращаемое значение: В случае успеха — указатель на интерфейс, иначе — **NULL**.

4.4.3. Завершение работы с интерфейсом модуля

Формат:	**bool**	***ReleaseLDevice**(void)*	(версия 2.1)
	BOOL	***ReleaseLInstance**(void)*	(с версии 3.0)

Назначение:

Данная интерфейсная функция реализует корректное высвобождение интерфейсного указателя, проинициализированного с помощью интерфейсной функции *CreateLInstance()*. Используется для аккуратного завершения сеанса работы с модулем, если предварительно удачно выполнилась функция *CreateLInstance()*. **!!!Внимание!!!** Данная функция должна обязательно вызываться в Вашем приложении перед непосредственным выходом из него во избежание утечки ресурсов компьютера.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 4.1. "Общие принципы работы с модулем"*.

Передаваемые параметры: нет.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.4.4. Инициализация доступа к модулю

Формат:	**BOOL**	***OpenLDevice**(WORD VirtualSlot)*

Назначение:

С программной точки зрения, не вдаваясь в излишние тонкости, подсоединенный к компьютеру модуль *E14-140* можно рассматривать как устройство, подключённое к некоему виртуальному слоту с сугубо индивидуальным номером. Основное назначение данной интерфейсной функции как раз в том и состоит, чтобы определить, что именно модуль *E14-140* находится в заданном виртуальном слоте. Если функция ***OpenLDevice()*** успешно выполнилась для заданного виртуального слота, то можно переходить к чтению служебной информации и управлению модулем с помощью соответствующих интерфейсных функций штатной библиотеки `Lusbapi`.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 4.1. "Общие принципы работы с модулем"*.

Передаваемые параметры:

- *VirtualSlot* – номер виртуального слота, к которому, как предполагается, подключен модуль *E14-140*.

Возвращаемое значение: *TRUE* – модуль *E14-140* находится в выбранном виртуальном слоте и можно переходить к работе с ним;
FALSE – устройства типа модуль *E14-140* в выбранном виртуальном слоте нет. Следует попробовать другой номер виртуального слота.

4.4.5. Завершение доступа к модулю

Формат:	**BOOL**	***CloseLDevice*** *(void)*

Назначение:

Данная интерфейсная функция прерывает всякое взаимодействие с *текущим* виртуальным слотом, к которому подключён модуль. При этом данный виртуальный слот аккуратно закрывается и выполняется освобождение связанных с ним ресурсов *Windows*. После её применения всякий доступ к модулю *E14-140* становится невозможным. Для возобновления нормального доступа к устройству необходимо вновь воспользоваться интерфейсной функцией *OpenLDevice()*. Таким образом, эта функция, по своей сути, противоположна интерфейсной функции *OpenLDevice()*. Фактически данная функция используется в таких интерфейсных функциях, как *OpenLDevice()* и *ReleaseLInstance()*.

Передаваемые параметры: нет.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.4.6. Получение названия модуля

Формат:	**BOOL**	***GetModuleName****(PCHAR const ModuleName)*

Назначение:

Данная вспомогательная интерфейсная функция позволяет получить название подключенного к слоту модуля. Массив под название модуля *ModuleName* (не менее 6 символов плюс признак конца строки ‘\0’, т.е. нулевой байт) должен быть заранее определен.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 4.1. "Общие принципы работы с модулем"*.

Передаваемые параметры:

- *ModuleName* – возвращается строка, не менее 6 символов, с названием модуля (в нашем случае это должна быть строка "E140").

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.4.7. Получение скорости работы модуля

Формат:	**BOOL**	***GetUsbSpeed****(BYTE * const UsbSpeed)*

Назначение:

Данная функция позволяет определить, на какой скорости шина **USB** работает с модулем.

Передаваемые параметры:

- *UsbSpeed* – возвращаемое значение этой переменной может принимать следующее:
  - ✓ 0 – модуль работает с **USB** шиной в режиме *Full-Speed Mode* (12 Мбит/с)
  - ✓ 1 – модуль работает с **USB** шиной в режиме *High-Speed Mode* (480 Мбит/с).

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.4.8. Получение дескриптора устройства

Формат: **HANDLE** ***GetModuleHandle****(void)*
Назначение: Данная функция позволяет получить дескриптор (*handle*) используемого модуля *E14-140*.
Передаваемые параметры: нет
Возвращаемое значение: В случае успеха – дескриптор модуля *E14-140*; в противном случае – **INVALID_HANDLE_VALUE**.

4.4.9. Получение описания ошибок выполнения функций

Формат: **int** ***GetLastErrorString****(LPTSTR lpBuffer, DWORD nSize)* (версия 2.1) **BOOL** ***GetLastErrorInfo***(*LAST_ERROR_INFO_LUSBAPI * const LastErrorInfo*) (с версии 3.0)
Назначение: Если в процессе работы с библиотекой `Lusbapi` какая-нибудь интерфейсная функция штатной библиотеки вернула ошибку, то только непосредственно после этого с помощью вызова данной интерфейсной функции можно получить краткое толкование произошедшего сбоя. Для некоторых функций, например *ReadData()* или *WriteData()*, при выявлении причины ошибки может потребоваться дополнительный вызов стандартной *Windows API* функции ***GetLastError()*** для классификации ошибок самой *Windows*.
Передаваемые параметры: • *LastErrorInfo* – указатель на структуру типа *LAST_ERROR_INFO_LUSBAPI*, в которой возвращается краткое описание и номер последней ошибки.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.5. Функции для работы с АЦП

Интерфейсные функции штатной библиотеки `Lusbapi` позволяют реализовывать разнообразные алгоритмы работы модуля *E14-140* с АЦП. Но основным режимом работы модуля является, конечно же, непрерывный *потоковый* сбор данных с АЦП. А вообще модуль, с точки зрения состояния АЦП, может находиться как бы в двух режимах:

1. режим "*покоя*";
2. потоковый, т.е. *перманентный*, сбор данных с АЦП.

Функция *START_ADC()* позволяет переводить модуль во второе из этих состояний, а *STOP_ADC()* — в первое. Прежде чем запустить модуль на сбор данных с АЦП, необходимо в него передать все требуемые параметры функционирования АЦП: тип синхронизации, частота работы АЦП, управляющую таблицу и т.д. Эту операцию можно осуществить с помощью интерфейсной функции *SET_ADC_PARS()*. После этого, в принципе, можно запускать модуль на сбор данных, выполнив функцию *START_ADC()*. Для извлечения из модуля уже полученных с АЦП данных следует пользоваться функцией *ReadData()*. При этом функция *ReadData()* может выполняться как в *синхронном*, так и в *асинхронном* режимах.

В зависимости от ревизии модуля максимальная частота работы АЦП может быть:

- 100 *кГц* для модуля *E14-140 (Rev.'A')*;
- 200 *кГц* для модуля *E14-140 (Rev.'B' и выше)*.

Для повышения надёжности сбора данных с АЦП, отсчёты помещаются во внутренний промежуточный *FIFO* буфер АЦП модуля. Размер этого буфера в зависимости от ревизии модуля может составлять:

- 32 КСлова для модуля *E14-140 (Rev.'A')*;
- 16 КСлова для модуля *E14-140 (Rev.'B' и выше)*.

Этот *FIFO* буфер позволяет демпфировать временны**е** задержки в моменты, когда компьютер по каким-либо причинам не может забирать из модуля полученные с АЦП данные. Так, например, размер буфера 32 КСлова соответствует промежутку времени сбора данных, равному ~330 мс при частоте дискретизации АЦП равной 100 *кГц*. Приложение может забирать отсчёты АЦП из *FIFO* буфера модуля при помощи функции *ReadData()*.

Простой пример применения интерфейсных функций для работы с АЦП можно найти в директории `\E14-140\Example\`.

!!!Внимание!!! На частотах сбора данных больших 20 кГц вследствие большой загруженности управляющего микроконтроллера модуля параллельные процессы обращения к модулю *E14-140 (Rev.'A')*, такие как запросы по управлению линиями цифрового ввода/вывода, чтение/запись пользовательского ППЗУ, выставление значений на каналах ЦАП и т.д., могут приводить к потерям в получаемых данных АЦП.

4.5.1. Корректировка данных АЦП

Схемотехника и использованные компоненты обеспечивают линейность передаточной характеристики тракта АЦП модуля *E14-140*. Однако на *сегодняшний* день модуль не умеет производить автоматическую корректировку собираемых с АЦП данных. Это приводит к тому, что входные показания АЦП могут иметь некоторое смещение нуля и неточность в передаче масштаба. Поэтому на уровне приложения необходима реализация всей этой нудной задачи по корректировке данных АЦП. Зато благодаря программной коррекции данных на модуле полностью отсутствуют какие-либо подстроечные резисторы, что позволяет улучшить шумовые характеристики модуля и увеличить их надёжность.

В качестве корректировочных коэффициентов вполне можно использовать как *штатные*, так и свои собственные, т.е. *пользовательские*.

Пользовательские корректировочные коэффициенты могут быть использованы, например, для целей компенсации погрешностей целого измерительного тракта какого-нибудь стенда, составной частью которого вполне может служить модуль *E14-140*. При этом вся ответственность за фор-

мирование и корректное применение *пользовательских* корректировочных коэффициентов полностью ложится на плечи конечного пользователя.

Штатные корректировочные коэффициенты располагаются в полях *Adc.OffsetCalibration[]* и *Adc.ScaleCalibration[]* структуры служебной информации *MODULE_DESCRIPTION_E140*. Вся служебная информация совместно с корректировочными коэффициентами записывается в модуль на этапе при его наладке в ООО "Л Кард". Поля коэффициентов представляют собой массивы типа *double*. Для модуля *E14-140* в каждом из этих массивов используются только первые *ADC_CALIBR_COEFS_QUANTITY_E140* элементов. Массив *Adc.OffsetCalibration* содержит коэффициенты для корректировки смещение нуля, а массив *Adc.ScaleCalibration* – для корректировки масштаба. Если в *логическом номере канала АЦП* индекс коэффициента усиления равен *i*, то корректировочные коэффициенты этого логического канала могут быть получены следующим образом:

- смещение: *Adc.OffsetCalibration[i]*;
- масштаб: *Adc.ScaleCalibration[i]*.

В общем виде процедура корректировки отсчётов АЦП выполняется по следующей формуле:

$$Y = (X+A)*B,$$

где: X – некорректированные данные АЦП [в отсчётах АЦП],
Y – скорректированные данные АЦП [в отсчётах АЦП],
A – коэффициент смещения нуля [в отсчётах АЦП],
B – коэффициент масштаба [безразмерный].

Например, пусть со второго канала АЦП, настроенного на входной диапазон ±2.5 В (индекс коэффициента усиления равен 1 или *ADC_INPUT_RANGE_2500mV_E140*), получены следующие данные: X1 = 1000, X2 = –1000 и X3 = 0. Тогда коэффициенты и скорректированные данные можно представить так:

- A = Adc.OffsetCalibration[1];
- B = Adc.ScaleCalibration[1];
- Y1 = (A+1000)*B, Y2 = (A-1000)*B, Y3 = A*B.

4.5.2. Запуск АЦП.

Формат:	**BOOL**	***START_ADC**(void)*

Назначение:

Данная функция запускает модуль *E14-140* на непрерывный *потоковый* сбор данных с АЦП. Перед любым запуском сбора данных настоятельно рекомендуется выполнять функцию *STOP_ADC()*. Перед началом сбора можно установить требуемые параметры работы АЦП, которые передаются в модуль с помощью интерфейсной функции *SET_ADC_PARS()*. Извлечение из модуля уже собранных с АЦП данных можно осуществлять с помощью интерфейсной функции *ReadData()*.

Передаваемые параметры: нет

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.3. Останов АЦП

Формат:	**BOOL**	***STOP_ADC**(void)*

Назначение:

Данная функция останавливает в модуле *E14-140* механизм сбора данных с АЦП. Попутно эта функция '*приводит в чувство*' основную программу микроконтроллера модуля, а также сбрасывает используемый канал передачи данных по **USB** шине. Поэтому настоятельно рекомендуется применять эту функцию перед каждым запуском сбора данных функцией *START_ADC()*.

Передаваемые параметры: нет

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.4. Установка параметров работы АЦП

Формат:	**bool**	***FILL_ADC_PARS**(ADC_PARS_E140 *am)*	(версия 2.1)
	BOOL	***SET_ADC_PARS**(ADC_PARS_E140 * const AdcPars)*	(с версии 3.0)

Назначение:

Данная функция передает в *E14-140* всю необходимую информацию, которая используется модулем для организации заданного режима сбора данных с АЦП. Всю нужную информацию описываемая интерфейсная функция извлекает из полей передаваемой структуры типа *ADC_PARS_E140*. Собственно, использование модулем именно этой переданной информации начинается только после выполнения интерфейсной функции *START_ADC()*. Также крайне не рекомендуется вызывать эту функцию собственно в процессе сбора данных. Её следует применять только после выполнения функции *STOP_ADC()*.

Описание структуры *ADC_PARS_E140* приведено ранее в *§ 4.3.2. "Структура ADC_PARS_E140"*, а назначение отдельных ее полей описано ниже.

- Поле AdcPars->*ClkSource*. Запись. Модулю *E14-140* для работы АЦП требуется наличие аппаратных *тактовых импульсов*. Значение данного поля определяет источник формирования этих импульсов. Это поле может принимать одно из двух значений

от 0 до 1, также можно пользоваться *константами источника тактовых импульсов*. Смысловая нагрузка значений этого поля представлена в таблице ниже:

Значение	Константа	Назначение
0	**INT_ADC_CLOCK_E140**	***Внутренние тактовые импульсы АЦП.*** При этом *тактовые импульсы* аппаратно генерируются при помощи таймера, встроенного в микроконтроллер модуля. Также, в этом режиме можно настроить модуль на трансляцию этих *тактовых импульсов* АЦП на линию ввода-вывода ***SYN*** цифрового разъёма модуля. Подробности смотри в описании поля *AdcPars->EnableClkOutput*.
1	**EXT_ADC_CLOCK_E140**	***Внешние тактовые импульсы АЦП.*** В этом режиме используется внешний источник *тактовых импульсов* АЦП, который подключается к линии ввода-вывода ***SYN*** цифрового разъёма модуля.
2	**INVALID_ADC_ CLOCK_E140**	Неправильный тип источника *тактовых импульсов* АЦП.

- Поле AdcPars->*EnableClkOutput*. Запись. Модуль *E14-140* позволяет осуществлять трансляцию *тактовых импульсов* АЦП на линию ввода-вывода ***SYN*** цифрового разъёма. Данное поле имеет значение только при внутренних *тактовых импульсах* АЦП, когда поле *AdcPars->ClkSource* равно нулю. Иначе значение данного поля игнорируется. Данное поле определяет, нужна ли *трансляция тактовых импульсов* АЦП на цифровой разъём модуля. Это поле может принимать одно из двух значений от 0 до 1, также можно пользоваться *константами трансляции тактовых импульсов*. Смысловая нагрузка значений этого поля представлена в таблице ниже:

Значение	Константа	Назначение
0	**ADC_CLOCK_TRANS_ DISA- BLED_E140**	***Запретить трансляцию тактовых импульсов АЦП.*** При этом линия ввода-вывода ***SYN*** цифрового разъёма будет находиться в третьем, *высокоимпедансном*, состоянии.
1	**ADC_CLOCK_TRANS_ ENABLED_E140**	***Разрешить трансляцию тактовых импульсов АЦП.*** При этом *тактовые импульсы* АЦП, генерируемые таймером микроконтроллера, будут транслироваться на линию ввода-вывода ***SYN*** цифрового разъёма модуля.
2	**INVALID_ADC_ CLOCK_TRANS_E140**	Неправильное значение трансляции *тактовых импульсов* АЦП.

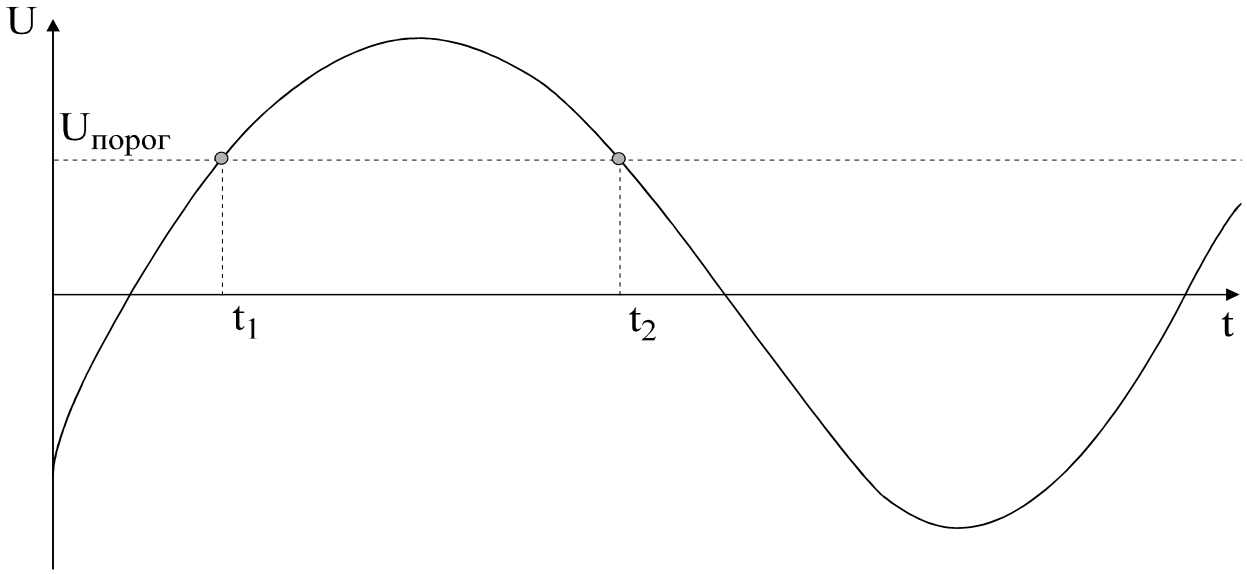
- Поле AdcPars->InputMode. Запись. Значение данного поля может задавать различные виды синхронизации ввода данных с АЦП. Это поле может принимать одно из четырёх значений от 0 до 3, также можно пользоваться *константами синхронизации ввода*. Смыс-

ловая нагрузка значений этого поля представлена в таблице ниже:

Значение	Константа	Назначение
0	**NO_SYNC_E140**	***Отсутствие синхронизации ввода.*** Сбор данных с АЦП модуль начинает непосредственно после выполнения функции *START_ADC()*.
1	**TTL_START_SYNC_E140**	***Цифровая синхронизация начала ввода.*** Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания прихода отрицательного перепада (⎍) у ТТЛ-совместимого одиночного импульса на входе **INT** аналогового разъёма. Длительность этого синхроимпульса должна быть не менее 100 нс. Только после этого модуль приступает к сбору данных.
2	**TTL_KADR_SYNC_E140**	***Цифровая покадровая синхронизация ввода.*** Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания прихода отрицательного перепада (⎍) у ТТЛ-совместимого одиночного импульса на входе **INT** аналогового разъёма. Длительность этого синхроимпульса должна быть не менее 100 нс. После прихода указанного синхроимпульса модуль собирает отсчёты только одного кадра. Во время ввода кадра данных вся активность на линии **INT** игнорируется. Только после окончания сбора текущего кадра данных ожидается приход следующего импульса синхронизации и т.д.
3	**ANALOG_SYNC_E140**	***Аналоговая синхронизация начала ввода***. Непосредственно после выполнения функции *START_ADC()* модуль переходит в состояние ожидания выполнения условий аналоговой синхронизации. Модуль начинает собирать данные с АЦП только после выполнения заданных соотношений между получаемым значением с заданного аналогового синхроканала и заданным пороговым значением. При этом условия аналоговой синхронизации ввода данных задаются через посредство полей AdcPars->SynchroAdType, AdcPars->SynchroAdMode, AdcPars->SynchroAdChannel и AdcPars->SynchroAdPo-rog.
4	**INVALID_SYNC_E140**	Неправильная синхронизация ввода данных.

- Поля AdcPars->SynchroAdType, AdcPars->SynchroAdMode, AdcPars->SynchroAdChannel, *AdcPars->SynchroAdPorog*. Запись. Эти поля используются исключительно при ана-

логовой синхронизации ввода данных. При этом различные моменты старта аналоговой синхронизации приведены на следующем рисунке:



Если, например, задана *аналоговая* синхронизация по **переходу** (AdcPars->SynchroAdType ≠ 0) '*снизу-вверх*' (AdcPars->SynchroAdMode = 0) при пороговом значении равном AdcPars->SynchroAdPorog = $U_{порог}$(в кодах АЦП), то модуль начнет собирать данные только после наступления момента времени t_1. При этом уровень входного сигнала на логическом синхроканале AdcPars->SynchroAdChannel пересечет пороговую линию в направлении '*снизу-вверх*'. Аналогично, если задан **переход** '*сверху-вниз*' (AdcPars->SynchroAdMode ≠ 0), то старт начало сбора наступит в момент времени t_2, когда уровень входного сигнала на синхроканале пересечет пороговую линию в направлении сверху вниз. Если же *аналоговая* синхронизация задается по **уровню** (AdcPars->SynchroAdType = 0), то сбор модулем данных начнется в тот момент, когда уровень входного сигнала на синхроканале окажется либо выше (AdcPars->SynchroAdMode = 0), либо ниже (AdcPars->SynchroAdMode ≠ 0) пороговой линии $U_{порог}$.

Все вышесказанное относительно аналоговой синхронизации ввода данных можно кратко свести к следующему:

- поле AdcPars->SynchroAdType может принимать следующие значения:
  - ✓ AdcPars->SynchroAdType = 0 — аналоговая синхронизация по **уровню**,
  - ✓ AdcPars->SynchroAdType ≠ 0 — аналоговая синхронизация по **переходу**;
- поле AdcPars->SynchroAdMode может принимать следующие значения:
  - ✓ при AdcPars->SynchroAdMode = 0:
    - аналоговая синхронизация по **уровню** — '*выше*',
    - аналоговая синхронизация по **переходу** — '*снизу-вверх*',
  - ✓ при AdcPars->SynchroAdMode ≠ 0:
    - аналоговая синхронизация по **уровню** — '*ниже*',
    - аналоговая синхронизация по **переходу** — '*сверху-вниз*';
- поле AdcPars->SynchroAdChannel – логический номер синхроканала АЦП;
- поле AdcPars->SynchroAdPorog – пороговое значение в кодах АЦП;

- Поле AdcPars->ChannelsQuantity. Запись–Чтение. Данное поле задаёт количество активных *логических каналов* в управляющей таблице **ControlTable**. Т.е. модуль при сборе данных с АЦП будет использовать первые AdcPars->ChannelsQuantity элементов массива AdcPars->ControlTable. Предельным значением для данного поля является величина *MAX_CONTROL_TABLE_LENGTH_E140*.

- Поле AdcPars->ControlTable[]. Запись. Данное поле является массивом типа *WORD*. Оно задаёт управляющую таблицу **ControlTable**. Т.е. тот массив *логических каналов*, который будет использоваться модулем при работе с АЦП для задания циклической последовательности отсчётов с входных аналоговых каналов.
- Поля AdcPars->AdcRate и AdcPars->InterKadrDelay. Запись–Чтение. Данные поля имеют силу только при использовании модулем внутренних *тактовых импульсов*, что определяется полем *AdcPars->ClkSource*. При этом поле AdcPars->InterKadrDelay не принимается в рассмотрение при использовании *покадровой синхронизации ввода*, которая может задаваться полем *AdcPars->InputMode*. При входе в функцию в данных полях должны содержаться требуемые временны**ы**е параметры сбора данных: частота работы АЦП **AdcRate** (обратная величина *межканальной* задержки) и межкадровая задержка **InterKadrDelay**. При этом **AdcRate** задаётся в *кГц*, а **InterKadrDelay** – в *мс*. После выполнения функции ***SET_ADC_PARS()*** в этих полях находятся реально установленные значения величин межканальной и межкадровой задержек, максимально близкие к изначально задаваемым. Это происходит вследствие того, что реальные значения **AdcRate** и **InterKadrDelay** не являются непрерывными величинами, а образуют некую сетку частот. Так, в общем виде, частота работы АЦП определяется по следующей формуле: **AdcRate** = $F_{clock}/2/N$, где F_{clock} – тактовая частота установленного на модуле микроконтроллера, равная 16000 кГц, а N – целое число, находящееся в диапазоне:
  - от 80 до 65535 для модуля *E14-140 (Rev.'A')*;
  - от 40 до 65535 для модуля *E14-140 (Rev.'B' и выше)*.

  Поэтому данная функция просто вычисляет ближайшую к задаваемой дискретную величину **AdcRate**, передает её в модуль в виде целого числа N, а также возвращает её значение в поле AdcPars->AdcRate. Все тоже самое верно и для межкадровой задержки **InterKadrDelay**, с той лишь разницей, что она задается по формуле **InterKadrDelay** = K/**AdcRate**, где K – целое число в диапазоне от 1 до 256. Причём здесь используется уже откорректированная величина **AdcRate**. В итоге минимальное значение **AdcRate** может составлять 0.122 кГц, максимальное – 100/200 *кГц* (в зависимости от ревизии модуля). Например, если задать AdcPars->AdcRate = 0, то ***SET_ADC_PARS()*** установит и возвратит минимально возможную величину для данного поля, т.е. 0.122 кГц. Аналогично, если задать AdcPars->InterKadrDelay = 0, то функция установит и возвратит минимально возможную межкадровую задержку, т.е. 1/AdcPars->AdcRate.
- Поле AdcPars->KadrRate. Чтение. В данном поле возвращается частота кадра **KadrRate** в *кГц*. Но это поле имеет силу только при использовании внутренних *тактовых импульсов*, что задаётся полем *AdcPars->ClkSource*. При этом поле AdcPars->KadrRate нельзя принимать в рассмотрение при использовании *покадровой синхронизации ввода*, которая определяется полем *AdcPars->InputMode*. **KadrRate** рассчитывается исходя из величины AdcPars->ChannelsQuantity, а также уже скорректированных AdcPars->AdcRate и AdcPars->InterKadrDelay. Дополнительно про соотношения между упомянутыми выше величинами AdcPars->ChannelsQuantity, AdcPars->AdcRate, AdcPars->InterKadrDelay и AdcPars->KadrRate смотри *§ 3.2.4. "Формат кадра отсчетов"*.

Передаваемые параметры:

- *AdcPars* – адрес структуры типа *ADC_PARS_E140* с требуемыми параметрами функционирования АЦП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.5. Получение текущих параметров работы АЦП

Формат:	**bool**	***GET_CUR_ADC_PARS***(*ADC_PARS_E140 *am)*	(версия 2.1)
	BOOL	***GET_ADC_PARS***(*ADC_PARS_E140 * const AdcPars)*	(с версии 3.0)

Назначение:

Данная функция считывает из модуля *E14-140* всю текущую информацию, которая используется при сборе данных с АЦП.

Передаваемые параметры:

- *AdcPars* – адрес структуры типа *ADC_PARS_E140* с полученными из модуля текущими параметрами функционирования АЦП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.6. Получение массива данных с АЦП

Формат:	**bool**	***ReadData***(*SHORT *Buffer, DWORD *NumberOfWordsToRead, LPDWORD NumberOfBytesRead, LPOVERLAPPED Overlapped*)	(версия 2.1)
	BOOL	***ReadData***(*IO_REQUEST_LUSBAPI * const ReadRequest*)	(с версии 3.0)

Назначение:

Данная функция предназначена для извлечения из модуля *E14-140* очередной порции собранных данных АЦП. Эта функция должна использоваться совместно с функциями *START_ADC()* и *STOP_ADC()*.

Поля передаваемой структуры типа *IO_REQUEST_LUSBAPI* определяют параметры и требуемый режим получения данных с модуля *E14-140*. Назначения полей этой структуры приведены в таблице ниже:

Название поля	Описание
Buffer	***Буфер данных.*** Чтение. **Buffer** предназначен для хранения получаемых с модуля данных АЦП. Перед его использованием в функции приложение само должно позаботиться о выделении достаточного кол-ва памяти под этот буфер. Полученные данные в буфере будут располагаться покадрово: 1ый кадр, 2ой кадр и т.д. Причём положение отсчётов в кадрах будет совпадать с порядком размещения соответствующих *логических каналов* в управляющей таблице **ControlTable**.
NumberOf WordsToPass	***Кол-во передаваемых данных.*** Запись–Чтение. Данный параметр задаёт то кол-во отсчётов АЦП, которое данная функция просто обязана стребовать с модуля. Величина параметра NumberOfWordsToPass должна находиться в диапазоне от 32 до (1024*1024), а также быть кратной 32. В противном случае данная функция сама подкорректирует величину этого поля, и по возвращении из функции в нём будет находиться *реально* использованное значение кол-ва затребованных данных.

NumberOfWord-sPassed	***Кол-во переданных данных.*** Чтение. В данном параметре возвращается то кол-во отсчётов АЦП, которое данная функция реально получила из модуля. Для *асинхронного* режима работы данной функции (см. ниже поле Overlapped) в этом параметре вполне может вернуться число 0, что *не является* ошибкой, учитывая специфику данного режима.
Overlapped	***Структура Overlapped.*** Данное поле определяет, в каком именно режиме будет выполняться данная функция: *синхронном* или *асинхронном*: • Overlapped = NULL. В этом случае от функции требуется *синхронный* режим её выполнения. При этом функция честно пытается получить из модуля все затребованные данные, причём в течение всего этого времени функция не возвращает управление вызвавшему её приложению. Если в течение времени TimeOut *мс* (см. ниже) все требуемые данные из модуля не получены, то функция завершается и возвращает ошибку. • Overlapped != NULL. В этом случае от функции требуется *асинхронный* режим её выполнения. Подразумевается, что этому полю приложение уже присвоило указатель на заранее подготовленную структуру типа *OVERLAPPED*. В этом режиме данная функция выставляет системе, т.е. Windows, *асинхронный* запрос на получение требуемого кол-ва данных из модуля и сразу возвращает управление приложению. Т.е. происходит как бы полное перекладывание задачи по сбору данных на *ядро* системы. Так как *асинхронный* запрос выполняется уже на уровне *ядра*, то пока оно его обрабатывает, приложение вполне может заниматься своими собственными задачами. Окончание же текущего *асинхронного* запроса приложение можно отслеживать с помощью стандартных *Windows API* функций, таких как: *WaitForSingleObject()*, *GetOverlappedResult()* или *HasOverlappedIoCompleted()*. Данные функции используют событие *Event*, которые предварительно уже должно было быть определено приложением в соответствующем поле структуры Overlapped. Событие *Event* активируется системой по окончании сбора *всех* затребованных данных, завершая тем самым текущий *асинхронный* запрос. В ряде случаев бывает просто необходимо прервать выполняющийся *асинхронный* запрос. Для этой цели и существует штатная *Windows API* функция *CancelIo()*. К сожалению, существует эта функция только в *Windows NT* подобных системах.
TimeOut	***Время ожидания сбора данных.*** Это поле предназначено для использования только в *синхронном* режиме. Задаёт максимальное время ожидания выполнения *синхронного* запроса на сбор данных в *мс*. Если по истечении этого времени *все* затребованные данные недополучены, функция завершается и возвращает ошибку.

В модуле *E14-140* существует внутренний *FIFO* буфер АЦП, размер которого составляет 16 КСлова. Такой буфер необходим для уверенного и надёжного сбора данных на предельных частотах работы АЦП. Так при размере *FIFO* буфера 16 КСлов и частотах сбора порядка 100 кГц переполнение буфера произойдёт только через ~160 мс, что является вполне достаточным периодом времени даже для такой '*задумчивой*' операционной системы как *Windows*.

Теперь следует упомянуть о некоторой специфике, присущей режимам данной функции:

1. *Синхронный* режим. Данный режим рекомендуется применять при организации однократного сбора данных, в котором количество отсчётов не превышает 1024*1024 = 1 МСлова. В этом режиме функция ***ReadData()*** должна вызываться только после успешного исполнения *START_ADC()*, которой, в принципе, должна предшествовать функция *STOP_ADC()*. Следует с большой осторожностью использовать данный режим при достаточно медленных частотах сбора и большом количестве запрашиваемых данных. Иначе данная функция может надолго '*уйти*' в сбор данных и, следовательно, весьма длительное время не возвращать управление приложению. Пример корректного использования функций библиотеки `Lusbapi` в *синхронном* режиме в виде консольного приложения можно найти на нашем CD-ROM в директории `\E14-140\Examples\Borland C++ 5.02\ReadDataSynchro`.
2. *Асинхронный* режим. Этот режим функционально намного гибче, чем *синхронный* режим и его рекомендуется использовать при организации непрерывного *потокового* сбора данных, когда количество вводимых отсчётов превышает 1024*1024 = 1 МСлов. Данный режим позволяет организовывать в системе *Windows* очередь *асинхронных* запросов. Так можно сформировать очередь предварительных запросов даже непосредственно перед запуском сбора данных, но после функции *STOP_ADC()*. Такое использование очереди запросов позволяет резко повысить надёжность сбора данных. Операционная система *Windows* не является, что называется, средой реального времени. Поэтому, работая в ней, как это обычно бывает, всего лишь на *пользовательском* уровне, а не на уровне *ядра*, никогда нельзя быть полностью уверенным в том, что система в самый нужный момент не отвлечётся на свои собственные нужды на более или менее продолжительный промежуток времени. Например, если для частоты сбора 100 кГц после *START_ADC()*, но перед началом выполнения функции получения данных ***ReadData()*** система '*задумалась*' более чем на *327 мс* (что бывает очень редко, но вполне возможно), то сбой в принимаемых данных практически обеспечен. Однако если непосредственно перед *START_ADC()* выставить с помощью ***ReadData()*** несколько (можно и один) предварительных запросов, которые будут отрабатываться уже на уровне *ядра* системы, то сбоев не будет. Это происходит потому, что время отклика на отработку какого-нибудь события (в нашем случае это запрос) на уровне *ядра* существенно меньше, чем на *пользовательском* уровне. Т.о. получается, что после выполнения функции *START_ADC()* у нас уже есть готовые к обслуживанию запросы на уровне *ядра* системы. Практически никаких задержек. А теперь, пока система выполняет наши предварительные запросы, можно не торопясь, по мере надобности, выставлять один или несколько следующих запросов. Важно понимать, что для каждого выставленного в очередь или уже выполняющегося запроса у приложения должен существовать свой экземпляр структуры типа *IO_REQUEST_LUSBAPI* со своим индивидуальным событием *Event*.

Передаваемые параметры:

- *ReadRequest* – структура типа *IO_REQUEST_LUSBAPI* с параметрами извлечения готовых данных АЦП из модуля *E14-140*.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.7. Ввод кадра отсчетов с АЦП

Формат: **BOOL** ***ADC_KADR***(*SHORT * const Data*)

Назначение:

Данная интерфейсная функция позволяет осуществлять ввод кадра отсчётов с АЦП модуля. Эта функция удобна для осуществления достаточно *медленного* (порядка нескольких десятков *Гц*) асинхронного ввода целого кадра данных с требуемых входных аналоговых каналов. Такие параметры сбора кадра, как разрешение корректировки данных с АЦП, количество опрашиваемых каналов, частота работы АЦП и т.д., должны предварительно быть заданы с помощью штатной интерфейсной функции *SET_ADC_PARS()*. В зависимости от ревизии модуля кол-во опрашиваемых каналов в кадре может быть не более:

- 32 для модуля *E14-140 (Rev.'A')*;
- 128 для модуля *E14-140 (Rev.'B' и выше)*.

При этом информация о синхронизации ввода данных никак не используется, т.е. просто игнорируется. Массив *Data* необходимой длины для получаемых с модуля данных следует заранее определить. Более подробно смотри исходные тексты примера из директории `\E14-140\Example\Borland C++ 5.02\AdcKadr`.

Передаваемые параметры:

- *Data* – указатель на массив, в который складываются полученный кадр отсчётов с АЦП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.5.8. Однократный ввод с АЦП

Формат: **BOOL** ***ADC_SAMPLE***(*SHORT * const AdcData, WORD AdcChannel*)

Назначение:

Данная функция устанавливает заданный логический канал и осуществляет его однократное аналого-цифровое преобразование. Эта функция удобна для осуществления достаточно *медленного*, порядка нескольких десятков *Гц*, асинхронного ввода данных с задаваемового логического канала АЦП (см. *§ 3.2.3. "Логический номер канала АЦП"*). Более подробно смотри исходные тексты примера из директории `\E14-140\Example\Borland C++ 5.02\AdcSample`.

Передаваемые параметры:

- *AdcSample* – результат преобразования по заданному логическому каналу АЦП *AdcChannel*;
- *AdcChannel* – требуемый логический номер канала АЦП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

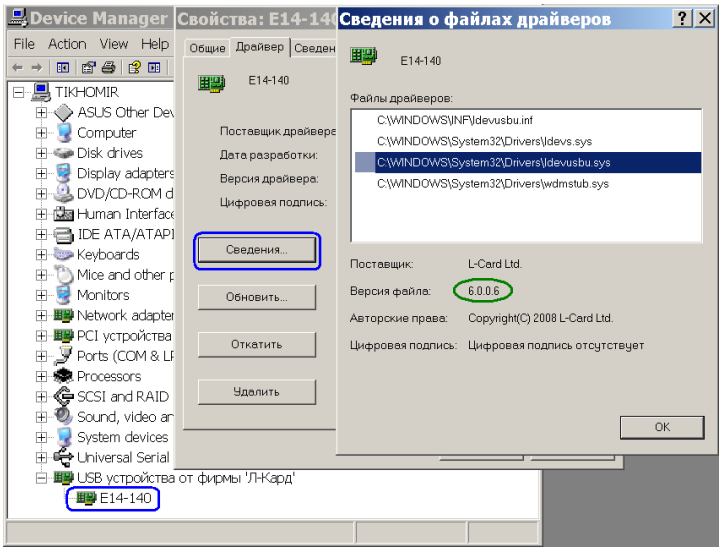
4.6. *Функции для работы с ЦАП*

Аппаратура модуля *E14-140 (Rev.'A')* позволяет управлять выводом на ЦАП только асинхронным (однократным) образом. Т.о. вывод на ЦАП получается сравнительно медленной операцией. На модуль *E14-140 (Rev.'A')* по желанию пользователя может быть установлен двух канальный 12[ти] битный ЦАП. Пример асинхронной работы с ЦАП можно найти в директории `\E14-140\Examples\Borland C++ 5.02\DacSample`.

Модуль *E14-140 (Rev.'B'* и выше*)* помимо асинхронной работы ЦАП обладает полновесной аппаратной поддержкой *потокового вывода* на ЦАП. На модуль *E14-140 (Rev.'B'* и выше*)* по желанию пользователя может быть установлен двух канальный 16[ти] битный ЦАП. Причём вывод данных на ЦАП происходит *синхронно* сразу на оба канала. Перед запуском потокового вывода на ЦАП предварительно необходимо передать в модуль требуемые параметры его функционирования: частота работы ЦАП, режим запуска и вид останова ЦАП. Эту операцию можно выполнить с помощью интерфейсной функции *SET_DAC_PARS()*. Для собственно запуска потокового вывода на ЦАП следует использовать функцию *START_DAC()*. Для *'подкачки'* в модуль выводимых на ЦАП данных следует пользоваться функцией *WriteData()*, причём вызов этой функции можно осуществлять как до *START_DAC()*, так и после неё. Завершение потокового вывода на ЦАП можно выполнять с помощью функции *STOP_DAC()*. Примеры потокового вывода данных на ЦАП можно найти в директории `\E14-140\Examples\Borland C++ 5.02\WriteData`.

4.6.1. USB драйвер для E14-140 (Rev.'B' и выше)

Одним из основных нововведений модуля *E14-140 (Rev.'B' и выше)* является возможность потокового вывода данных на ЦАП. Для организации корректной работы потокового ЦАП необходимо наличие в системе *Windows* (директория `%SystemRoot%\system32\drivers`) драйвера **`Ldevusbu.sys`** версии не ниже 6.0.0.6. Поверить версию текущего **USB** драйвера можно, например, с помощью *"Device Manager"* (*"Диспетчера устройств"*). В нём следует штатным образом посмотреть на свойства подключённого модуля *E14-140* как это показано на рисунке ниже:



!!! ВНИМАНИЕ !!! Использование потокового вывода на ЦАП со старыми **USB** драйверами с неизбежностью приведёт к полному краху операционной системы вплоть до появления синего экрана смерти (BSOD – Blue Screen Of Death). Но ситуация BSOD возникнуть не должна при использовании старого программного обеспечения (где потокового вывода на ЦАП не было) и старого драйвера для модуля *E14-140 (Rev.'A')*.

4.6.2. Корректировка данных ЦАП

Схемотехника и использованные компоненты обеспечивают линейность передаточной характеристики ЦАП тракта модуля *E14-140*. Однако на *сегодняшний* день модуль не умеет произво-

дить автоматическую корректировку выводимых на ЦАП данных. Это приводит к тому, что выходные показания ЦАП могут иметь некоторое смещение нуля и неточность в передаче масштаба. Поэтому на уровне приложения необходима реализация всей этой нудной задачи по корректировке данных ЦАП. Для этих целей предназначены соответствующие калибровочные коэффициенты, хранящиеся в служебной информации модуля. Служебная информация совместно с нужными коэффициентами записывается в модуль на этапе при его наладке в ООО “Л Кард”. Благодаря этому на модуле отсутствуют подстроечные резисторы, что улучшает шумовые характеристики модуля и увеличивает их надежность.

Сами коэффициенты располагаются в полях *Dac.OffsetCalibration[]* и *Dac.ScaleCalibration[]* структуры служебной информации *MODULE_DESCRIPTION_E140*. Эти поля представляют из себя массивы типа *double*. Для модуля *E14-140* в каждом из этих массивов используются только первые *DAC_CALIBR_COEFS_QUANTITY_E140* элементов. Массив *Dac.OffsetCalibration* содержит коэффициенты для корректировки смещение нуля первого и второго каналов ЦАП, а массив *Dac.ScaleCalibration* – для корректировки масштаба.

Корректировка данных ЦАП производится следующим образом: Y = (X+A)*B, где: X – некорректированные данные ЦАП [в кодах ЦАП], Y – скорректированные данные ЦАП [в кодах ЦАП], A – коэффициент смещения нуля [в кодах ЦАП], B – коэффициент масштаба [безразмерный].

Например, на втором канале ЦАП необходимо выставить напряжения, соответствующие следующим кодам ЦАП: X1 = 1000, X2 = -1000, X3 = 0. Тогда коэффициенты коррекции и данные для второго канала ЦАП можно представить так: A = Dac.OffsetCalibration[1], B = Dac.ScaleCalibration[1], Y1=(A+1000)*B, Y2=(A-1000)*B, Y3=A*B.

4.6.3. Запуск потокового вывода данных на ЦАП

Формат:	**BOOL**	***START_DAC***(*void*)
Назначение: Данная функция запускает непрерывный *потоковый* вывод данных на ЦАП только для модуля *E14-140 (Rev.'B'* и выше*)*. Перед любым запуском потокового вывода настоятельно рекомендуется выполнять функцию *STOP_DAC()*. Параметры потоковой работы ЦАП должны быть предварительно переданы в модуль с помощью интерфейсной функции *SET_DAC_PARS()*. ‘*Подкачку*’ в модуль выводимых данных можно осуществлять с помощью интерфейсной функции *WriteData()*.		
Передаваемые параметры: нет		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

4.6.4. Останов потокового вывода данных на ЦАП

Формат:	**BOOL**	***STOP_DAC***(*void*)
Назначение: Данная функция останавливает в модуле *E14-140 (Rev.'B'* и выше*)* механизм вывода данных на ЦАП, а также сбрасывает используемый канал передачи данных по **USB** шине. Поэтому весьма настоятельно рекомендуется применять эту функцию перед каждым запуском вывода данных функцией *START_DAC()_Запуск_вывода_данных*.		
Передаваемые параметры: нет		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

4.6.5. Установка параметров потоковой работы ЦАП

Формат: **BOOL** ***SET_DAC_PARS****(DAC_PARS_E140 * const DacPars)*

Назначение:

Данная функция передает в модуль *E14-140 (Rev.'B'* и выше*)* всю необходимую информацию, которая используется модулем для организации потокового вывода данных на ЦАП. Всю нужную информацию описываемая интерфейсная функция извлекает из полей передаваемой структуры типа *DAC_PARS_E140*. Собственно использование модулем именно этой переданной информацией начинается только после выполнения интерфейсной функции *START_DAC()*. Также крайне не рекомендуется вызывать эту функцию собственно в процессе сбора данных. Её следует применять только после выполнения функции *STOP_DAC()*.

Описание полей структуры *DAC_PARS_E140* приведено ранее в *§ 4.3.3. "Структура DAC_PARS_E140"*, а назначение отдельных ее полей описано ниже.

- Поле DacPars->SyncWithADC. Запись. Данное поле задаёт режим синхронного старта АЦП и ЦАП и может принимать значения 0 или 1. Если это поле принимает значение 1, то запуск ЦАП будет синхронизирован с запуском АЦП.
- Поле DacPars->SetZeroOnStop. Запись. Данное поле задаёт тип останова потоковой работы АЦП и может принимать значения 0 или 1. Если это поле принимает значение 1, то при останове потокового вывода при помощи функции *STOP_DAC()* на выходе ЦАП установится 0 В.
- Поле DacPars->DacRate. Запись–Чтение. При входе в функцию данное поле содержит требуемую частоту работы ЦАП **DacRate** в *кГц*. Причём это частота *синхронного* вывода сразу на оба канала ЦАП. После выполнения данной интерфейсной функции в этом поле возвращается реально установленное значение частоты вывода данных на ЦАП, максимально близкое к изначально задаваемой величине. Это происходит вследствие того, что реальные значения **DacRate** не являются непрерывной величиной, а образуют некую сетку частот. Так, в общем виде, частота работы ЦАП определяется по следующей формуле: **DacRate** = 200/(N+1) *кГц*, где N – целое число от 0 до 7. Поэтому данная функция просто вычисляет ближайшую к задаваемой дискретную величину **DacRate**, передает ее код в модуль в виде числа N, а также возвращает её значение в поле DacPars->DacRate. Минимальное значение **DacRate** равно 25 кГц, максимальное – 200 кГц.

Передаваемые параметры:

- *DacPars* – адрес структуры типа *DAC_PARS_E140* с параметрами потоковой работы ЦАП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.6.6. Получение текущих параметров работы ЦАП

Формат: **BOOL** ***GET_DAC_PARS***(*DAC_PARS_E140 * const DacPars*)

Назначение:

Данная функция считывает из модуля всю текущую информацию, которая используется в процессе потоковой выдачи данных на ЦАП.

Передаваемые параметры:

- *DacPars* – адрес структуры *DAC_PARS_E140* с полученными из модуля текущими параметрами потоковой работы ЦАП.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.6.7. Передача массива данных в ЦАП

Формат: **BOOL** ***WriteData***(*IO_REQUEST_LUSBAPI * const WriteRequest*)

Назначение:

Данная функция предназначена для передачи очередной порции данных в модуль *E14-140 (Rev.'B'* и выше*)* для их последующего потокового вывода на ЦАП. Эта функция должна использоваться совместно с функциями *START_DAC()* и *STOP_DAC()*.

Поля передаваемой структуры типа *IO_REQUEST_LUSBAPI* определяют параметры и требуемый режим передачи данных в модуль *E14-140*. Назначения полей этой структуры приведены в таблице ниже:

Название поля	Описание
Buffer	*Буфер данных.* Запись. **Buffer** предназначен для хранения очередной порции 16т^{ти} битных данных, которые необходимо будет переслать в модуль. Перед использованием **Buffer** в функции приложение само должно позаботиться о выделении достаточного кол-ва памяти под этот буфер. Также необходимо сформировать и расположить в **Buffer** сами данные для ЦАП. Формат отсчётов для ЦАП в буфере следующий: 1ый отчёт для первого канала, 1ый отчёт для второго канала, 2ой отчёт для первого канала, 2ой отчёт для второго канала и т.д.
NumberOfWordsToPass	*Кол-во передаваемых данных.* Запись–Чтение. Данный параметр задаёт то кол-во отсчётов ЦАП, которое данная функция просто обязана передать в модуль. Величина параметра NumberOfWordsToPass должна находится в диапазоне от 128 до (1024*1024), а также быть кратной 128. В противном случае данная функция сама подкорректирует величину этого поля, и по возвращении из функции в нём будет находиться *реально* использованное значение кол-ва передаваемых данных.

NumberOfWordsPassed	*Кол-во переданных данных.* Чтение. В данном параметре возвращается то кол-во отсчётов ЦАП, которое данная функция реально передала в модуль. Для *асинхронного* режима работы данной функции (см. ниже поле Overlapped) в этом параметре вполне может вернуть число 0, что *не является* ошибкой, учитывая специфику данного режима.
Overlapped	*Структура Overlapped*. Данное поле определяет, в каком именно режиме будет выполняться данная функция: *синхронном* или *асинхронном*: • Overlapped = NULL. В этом случае от функции требуется *синхронный* режим её выполнения. При этом функция честно пытается переслать в модуль все требуемые данные, причём в течение всего этого времени функция не возвращает управление вызвавшему её приложению. Если в течение времени TimeOut *мс* (см. ниже) все требуемые данные в модуль не переданы, то функция завершается и возвращает ошибку. • Overlapped != NULL. В этом случае от функции требуется *асинхронный* режим её выполнения. Подразумевается, что этому полю приложение уже присвоила указатель на заранее подготовленную структуру типа *OVERLAPPED*. В этом режиме данная функция выставляет системе, т.е. Windows, *асинхронный* запрос на передачу требуемого кол-ва данных в модуль и сразу возвращает управление приложению. Т.е. происходит как бы полное перекладывание задачи по передаче данных на *ядро* системы. Так как *асинхронный* запрос выполняется уже на уровне *ядра*, то пока оно его обрабатывает, приложение вполне может занимать своими собственными делами. Окончание же текущего *асинхронного* запроса приложение можно отслеживать с помощью стандартных *Windows API* функций, таких как: *WaitForSingleObject()*, *GetOverlappedResult()* или *HasOverlappedIoCompleted()*. Данные функции используют событие *Event*, которые предварительно уже должно было быть определено приложением в соответствующем поле структуры Overlapped. Событие *Event* активируется системой по окончании передачи *всех* затребованных данных, завершая тем самым текущий *асинхронный* запрос. В ряде случаев бывает просто необходимо прервать выполняющийся *асинхронный* запрос. Для этой цели и существует штатная *Windows API* функция *CancelIo()*. К сожалению, существует эта функция только в *Windows NT* подобных системах.
TimeOut	Время ожидания передачи данных. Это поле предназначено для использования только в *синхронном* режиме. Задаёт максимальное время ожидания выполнения *синхронного* запроса на сбор данных в *мс*. Если по истечении этого времени не *все* затребованные данные переданы в модуль, то функция завершается и возвращает ошибку.

В модуле *E14-140 (Rev.'B'* и выше*)* организован программный *FIFO* буфер ЦАП размером 12288 отсчёта. Такой буфер необходим уверенного вывода данных на предельных частотах работы ЦАП. Так при частотах вывода порядка 200 кГц переполнение *FIFO* буфера произойдёт только через 30 мс, что является вполне достаточным периодом времени даже для такой '*задумчивой*' системы как *Windows*.

Теперь следует упомянуть о некоторой специфике присущей режимам данной функции:

1. *Синхронный* режим. Данный режим рекомендуется применять при организации однократного вывода данных на ЦАП, в котором количество отсчётов не превышает 1024*1024 = 1 МСлова. В этом режиме функция ***WriteData()*** должна вызываться только после успешного исполнения *START_DAC()*, которой, для порядку, может предшествовать функция *STOP_DAC()*. Следует с великой аккуратностью использовать данный режим при достаточно медленных частотах вывода и большом количестве передаваемых данных. Иначе данная функция может надолго '*уйти*' в вывод данных и, следовательно, весьма длительное время не возвращать управление приложению.
2. *Асинхронный* режим. Этот режим функционально намного гибче, чем *синхронный* режим и его рекомендуется использовать при организации непрерывного *потокового* вывода данных на ЦАП, когда количество выводимых отсчётов превышает 1024*1024 = 1 МСлов. Данный режим позволяет организовывать в системе *Windows* очередь *асинхронных* запросов. Так можно сформировать очередь предварительных запросов даже непосредственно перед запуском вывода данных, но после функции *STOP_DAC()*. Такое использование очереди запросов позволяет резко повысить надёжность вывода данных. Операционная система *Windows* не является, что называется, средой реального времени. Поэтому, работая в ней, как это обычно бывает, всего лишь на *пользовательском* уровне, а не на уровне *ядра*, никогда нельзя быть полностью уверенным в том, что система в самый нужный момент не отвлечётся на свои собственные нужды на более или менее продолжительный промежуток времени. Например, если для частоты работы ЦАП равной 200 кГц после *START_DAC()*, но перед началом выполнения функции ***WriteData()*** система '*задумалась*' более чем на *30 мс* (что является вполне рядовым событием), то сбой в передаваемых данных практически обеспечен. Однако если непосредственно перед *START_DAC()* выставить с помощью
   WriteData() несколько (можно и один) предварительных запросов, которые будут отрабатываться уже на уровне *ядра* системы, то сбоев не будет. Это происходит потому, что время отклика на отработку какого-нибудь события (в нашем случае это запрос) на уровне *ядра* существенно меньше, чем на *пользовательском* уровне. Т.о. получается, что после выполнения функции *START_DAC()* у нас уже есть готовые к обслуживанию запросы на уровне *ядра* системы. Практически никаких задержек. А теперь пока система выполняет наши предварительные запросы, можно не торопясь, по мере надобности, выставлять один или несколько следующих запросов. Важно понимать, что для каждого выставленного в очередь или уже выполняющегося запроса у приложения должен существовать свой экземпляр структуры типа *IO_REQUEST_LUSBAPI* со своим индивидуальным событием *Event*.

Передаваемые параметры:

- *WriteRequest* – структура типа *IO_REQUEST_LUSBAPI* с параметрами вывода данных на ЦАП модуля *E14-140*.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.6.8. Однократный вывод на заданный канал ЦАП

Формат: **BOOL** ***DAC_SAMPLE***(*SHORT * const DacData, WORD DacChannel*)
Назначение: Данная функция позволяет однократно устанавливать на указанном канале ЦАП *DacChannel* напряжение в соответствии с 12ти битным значением *DacData* (в кодах ЦАП). Ожидается, что величина *DacData* должна находиться в диапазоне от –2048 до 2047, иначе функция автоматически ограничит значение *DacData* указанными пределами. Функция ***DAC_SAMPLE()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты вывода данных на ЦАП порядка нескольких десятков *Гц*. О соответствии кода ЦАП величине устанавливаемого на выходе модуля аналогового напряжения см. *§ 3.2.2.1. "Формат слова данных ЦАП для модуля E14-140 (Rev.'A')"*.
Передаваемые параметры: • *DacData* – устанавливаемое значение напряжения в кодах ЦАП (от –2048 до 2047). • *DacChannel* – требуемый номер канала ЦАП (0 или 1).
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.6.9. Однократный вывод на два канала ЦАП

Формат: **BOOL** ***DAC_SAMPLES***(*SHORT * const DacData1,* *SHORT * const DacData12*)
Назначение: Для модуля *E14-140 (Rev.'B'* и выше*)* данная функция позволяет однократно устанавливать сразу на обоих каналах ЦАП напряжение в соответствии с 16ти битными значениями *DacData1* и *DacData2* (в кодах ЦАП). Функция ***DAC_SAMPLES()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты вывода данных на ЦАП порядка нескольких десятков *Гц*. О соответствии кода ЦАП величине устанавливаемого на выходе модуля аналогового напряжения см. *§ 3.2.2.2. "Формат слова данных ЦАП для модуля E14-140 (Rev.'B' и выше)"*.
Передаваемые параметры: • *DacData1* – значение для первого канала ЦАП; • *DacData2* – значение для второго канала ЦАП.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.7. Функции для работы с цифровыми линиями

Модуль *E14-140* оборудован внешними цифровыми линиями: 16<u>ть</u> *входных* и 16<u>ть</u> *выходных*. Физически все эти цифровые линии располагаются на внешнем разъёме модуля под названием ***DIGITAL I/O***. По умолчанию, непосредственно после подачи на модуль питания все цифровые *выходные* линии находятся в *высокоимпедансном* состоянии.

Аппаратура модуля *E14-140* и, соответственно, библиотека `Lusbapi` позволяет работать с цифровыми линиями только асинхронным (однократным) образом. Т.о. работа с цифровыми линиями получается сравнительно медленной операцией, т.к. на модуле не предусмотрена аппаратная поддержка *потоковой* работы с ними.

4.7.1. Разрешение выходных цифровых линий.

Формат: **BOOL** ***ENABLE_TTL_OUT***(*BOOL EnableTtlOut*)
Назначение: Данная интерфейсная функция позволяет осуществлять управление разрешением *всех* цифровых *выходных* линий внешнего разъёма ***DIGITAL I/O*** модуля *E14-140*. Т.о. существует возможность программного перевода выходных линий в *высокоимпедансное* состояние и обратно.
Передаваемые параметры: • *EnableTtlOut* – флажок, управляющий состоянием разрешения всех цифровых выходных линий.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.7.2. Чтение внешних цифровых линий

Формат: **BOOL** ***TTL_IN***(*WORD * const TtlIn*)
Назначение: Данная интерфейсная функция осуществляет однократное чтение состояний *всех* 16<u>ти</u> цифровых входных линий внешнего разъёма ***DIGITAL I/O*** модуля *E14-140*. Функция ***TTL_IN()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты ввода данных с цифровых линий порядка нескольких сотен *Гц*.
Передаваемые параметры: • *TtlIn* – переменная, содержащая побитовое состояние входных цифровых линий.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.7.3. Вывод на внешние цифровые линии

Формат:	**BOOL**	***TTL_OUT***(*WORD TtlOut*)

Назначение:

Данная интерфейсная функция осуществляет однократную установку требуемых состояний сразу на *всех* 16ти цифровых выходных линиях внешнего разъёма ***DIGITAL I/O*** модуля *E14-140*. При этом в параметре *TtlOut* передается **побитовая** информация о требуемых состояниях для всех цифровых выходных линий. Так 1ый бит параметра *TtlOut* определяет состояние выходной линии DO1, 2ой – DO2 и т.д. Например, если нужно установить только выходную линию DO3, то нужно выполнить ***TTL_OUT(***0x0004***)***. Если же нужно установить сразу все 16ть выходных линии, то – ***TTL_OUT(***0xFFFF***)***.

Перед началом работы цифровые выходные линии предварительно должны быть разрешены с помощью интерфейсной функции *ENABLE_TTL_OUT()*. Сама функция ***TTL_OUT()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты вывода данных на выходные цифровые линии порядка нескольких сотен *Гц*.

Передаваемые параметры:

- *TtlOut* – переменная, содержащая побитовое состояние устанавливаемых выходных цифровых линий.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.8. Функции для работы с пользовательским ППЗУ

На модуле *E14-140* организовано так называемое *пользовательское* ППЗУ. Размер этой области составляет *USER_FLASH_SIZE_E140* байт и представляет собой 256 16[ти] разрядных слов. Всю эту область пользователь может смело использовать в своих сугубо частнособственнических интересах.

Для предотвращения случайной порчи содержимого пользовательского ППЗУ модуля, предусмотрена специальная последовательность записи. Для осуществления операции записи, предварительно следует разблокировать ППЗУ для записи при помощи интерфейсной функции *ENABLE_FLASH_WRITE()*. А по окончании всей процедуры записи заблокировать ППЗУ при помощи той же интерфейсной функции *ENABLE_FLASH_WRITE()*.

4.8.1. Разрешение записи в ППЗУ

Формат: **BOOL** ***ENABLE_FLASH_WRITE***(*BOOL IsUserFlashWriteEnabled*)

Назначение:

Данная интерфейсная функция разрешает (*TRUE*) либо запрещает (*FALSE*) режим записи в пользовательское ППЗУ модуля с помощью штатной интерфейсной функции *WRITE_FLASH_ARRAY()*. Следует помнить, что после завершения всех требуемых операций записи информации в пользовательское ППЗУ, необходимо с помощью данной интерфейсной функции запретить режим записи.

Передаваемые параметры:

- *EnableFlashWrite* – переменная может принимать следующие значения:
  - ✓ если *TRUE*, то режим записи в пользовательское ППЗУ разрешен,
  - ✓ если *FALSE*, то режим записи в пользовательское ППЗУ запрещен.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.8.2. Запись массива данных в ППЗУ

Формат: **BOOL** ***WRITE_FLASH_ARRAY***(*USER_FLASH_E140 * const UserFlash*)

Назначение:

Данная интерфейсная функция выполняет запись массива байт размером *USER_FLASH_SIZE_E140* в ППЗУ. Т.о. перезаписывается сразу *вся* доступная область пользовательского ППЗУ. Перед началом процедуры записи в пользовательское ППЗУ необходимо разрешить эту операцию с помощью интерфейсной функции *ENABLE_FLASH_WRITE()*. После окончания процедуры записи всей требуемой информации необходимо запретить режим записи с помощью той же функции *ENABLE_FLASH_WRITE()*.

Передаваемые параметры:

- *UserFlash* – фактически это байтовый массив, который должен быть записан в пользовательское ППЗУ.

Возвращаемое значение: *TRUE* – функция успешно выполнена;
FALSE – функция выполнена с ошибкой.

4.8.3. Чтение массива данных из ППЗУ

Формат: **BOOL** ***READ_FLASH_ARRAY***(*USER_FLASH_E140 * const UserFlash*)
Назначение: Данная интерфейсная функция вычитывает содержимое сразу всей области пользовательского ППЗУ.
Передаваемые параметры: • *UserFlash* – в этом байтовом массиве возвращается образ всего пользовательского ППЗУ.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.9. Функции для работы со служебной информацией

Служебная информация содержит самые общие данные об используемом модуле *E14-140*: название модуля, его серийный номер и ревизию, корректировочные коэффициенты для АЦП и ЦАП, версию текущей прошивки MCU, тактовую частоту работы исполнительного устройства MCU и многое другое. Некоторые данные из этой служебной информации необходимы функциям штатной библиотеки `Lusbapi` для своей корректной работы.

4.9.1. Чтение служебной информации

Формат:	**BOOL** ***GET_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E140 * const ModuleDescription*)
Назначение: Данная интерфейсная функция осуществляет чтение всей служебной информации в структуру типа *MODULE_DESCRIPTION_E140*. Эта информация требуется при работе с некоторыми интерфейсными функциями штатной библиотеки `Lusbapi`. Поэтому данную функцию, во избежания непредсказуемого поведения приложений, следует обязательно вызывать непосредственно после загрузки ПЛИС модуля и проверки его работоспособности (см. *§ 4.1. "Общие принципы работы с модулем"*)	
Передаваемые параметры: • *ModuleDescription* – указатель на структуру типа *MODULE_DESCRIPTION_E140*, в которую считывается вся служебная информация модуля.	
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

4.9.2. Запись служебной информации

Формат:	**BOOL** ***SAVE_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E140 * const ModuleDescription*)
Назначение: Данная интерфейсная функция позволяет сохранять в модуле всю служебную информацию из структуры типа *MODULE_DESCRIPTION_E140*. **!!!Внимание!!!** Применять данную функцию нужно только в случае крайней необходимости. Например, когда по тем или иным обстоятельствам испортилось содержимое служебной информации.	
Передаваемые параметры: • *ModuleDescription* – указатель на структуру типа *MODULE_DESCRIPTION_E140*, из которой служебная информация переносится в модуль.	
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

Приложение А.
Вспомогательные константы и типы

Вспомогательные константы и типы данных описаны в заголовочном файле `\DLL\Include\LusbapiTypes.h` и рассмотрены в нижеследующих разделах.

А.1. Константы

Вспомогательные константы, определённые в библиотеке `Lusbapi`, приведены в следующей таблице:

Название	Значение	Смысл
NAME_LINE_LENGTH_LUSBAPI	25	Длина строки с названием чего-либо. Например, название производителя или изделия, имя автора и т.д.
COMMENT_LINE_LENGTH_LUSBAPI	256	Длина строки с комментарием в какой-либо вспомогательной структуре.
ADC_CALIBR_COEFS_QUANTITY_LUSBAPI	128	Максимально возможное число корректировочных коэффициентов АЦП.
DAC_CALIBR_COEFS_QUANTITY_LUSBAPI	128	Максимально возможное число корректировочных коэффициентов ЦАП.

А.2. Структура VERSION_INFO_LUSBAPI

Вспомогательная структура *VERSION_INFO_LUSBAPI* содержит более или менее подробную информацию о программном обеспечении, работающем в каком-либо исполнительном устройстве: MCU, DSP, PLD и т.д. Данная структура описывается следующим образом:

```
struct VERSION_INFO_LUSBAPI
{
    BYTE Version[10];                    // версия ПО для исполнительного устройства
    BYTE Date[14];                       // дата сборки ПО
    BYTE Manufacturer[NAME_LINE_LENGTH_LUSBAPI];   // производитель ПО
    BYTE Author[NAME_LINE_LENGTH_LUSBAPI];         // автор ПО
    BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];     // строка комментария
};
```

А.3. Структура MODULE_INFO_LUSBAPI

Данная вспомогательная структура *MODULE_INFO_LUSBAPI* содержит самую общую информацию о модуле: название фирмы-изготовителя изделия, название изделия, серийный номер изделия, ревизия изделия и строка комментария. Эта структура описывается следующим образом:

```
struct MODULE_INFO_LUSBAPI
{
    BYTE CompanyName[NAME_LINE_LENGTH_LUSBAPI];  // название фирмы-изготовите-
                                                 // ля изделия
    BYTE DeviceName[NAME_LINE_LENGTH_LUSBAPI];   // название изделия
    BYTE SerialNumber[16];                       // серийный номер изделия
    BYTE Revision;                               // ревизия изделия
    BYTE Modification;                          // исполнение (вариант) модуля;
    BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];   // строка комментария
};
```

A.4. Структура INTERFACE_INFO_LUSBAPI

Вспомогательная структура *INTERFACE_INFO_LUSBAPI* содержит самую общую информацию об используемом интерфейсе для доступа к модулю. Данная структура описывается следующим образом:

```
struct INTERFACE_INFO_LUSBAPI
{
    BOOL Active;                    // флаг достоверности остальных полей структуры
    BYTE Name[NAME_LINE_LENGTH_LUSBAPI];          // название интерфейса
    BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];    // строка комментария
};
```

A.5. Структура MCU_INFO_LUSBAPI

Вспомогательная структура *MCU_INFO_LUSBAPI* содержит самую общую информацию об используемом исполнительном устройстве типа микроконтроллер (MCU). Данная структура описывается следующим образом:

```
template <class VersionType>
struct MCU_INFO_LUSBAPI
{
    BOOL   Active;                  // флаг достоверности остальных полей структуры
    BYTE   Name[NAME_LINE_LENGTH_LUSBAPI]; // название MCU
    double ClockRate;                      // тактовая частота работы MCU в кГц
    VersionType Version;                   // информация о программе MCU
    BYTE   Comment[COMMENT_LINE_LENGTH_LUSBAPI];    // строка комментария
};
```

A.6. Структура ADC_INFO_LUSBAPI

Вспомогательная структура *ADC_INFO_LUSBAPI* содержит самую общую информацию об используемом устройстве типа АЦП. Данная структура описывается следующим образом:

```
struct ADC_INFO_LUSBAPI
{
    BOOL   Active;          // флаг достоверности остальных полей структуры
    BYTE   Name[NAME_LINE_LENGTH_LUSBAPI];       // название АЦП
    double OffsetCalibration[ADC_CALIBR_COEFS_QUANTITY_LUSBAPI];
    // корректировочные коэффициенты смещения нуля АЦП
    double ScaleCalibration[ADC_CALIBR_COEFS_QUANTITY_LUSBAPI];
    // корректировочные коэффициенты масштаба АЦП
    BYTE   Comment[COMMENT_LINE_LENGTH_LUSBAPI];    // строка комментария
};
```

A.7. Структура DAC_INFO_LUSBAPI

Вспомогательная структура *DAC_INFO_LUSBAPI* содержит самую общую информацию об используемом устройстве типа ЦАП. Данная структура описывается следующим образом:

```
struct DAC_INFO_LUSBAPI
{
    BOOL   Active;                    // флаг достоверности остальных полей структуры
    BYTE   Name[NAME_LINE_LENGTH_LUSBAPI];        // название ЦАП
    double OffsetCalibration[DAC_CALIBR_COEFS_QUANTITY_LUSBAPI];
    // корректировочные коэффициенты смещения нуля ЦАП
    double
ScaleCalibration[DAC_CALIBR_COEFS_QUANTITY_LUSBAPIADC_CALIBR_COEFS_QUANTITY_E2010];
    // корректировочные коэффициенты масштаба ЦАП
    BYTE   Comment[COMMENT_LINE_LENGTH_LUSBAPI];      // строка комментария
};
```

A.8. Структура DIGITAL_IO_INFO_LUSBAPI

Вспомогательная структура *DIGITAL_IO_INFO_LUSBAPI* содержит самую общую информацию об используемых устройствах цифрового ввода-вывода. Данная структура описывается следующим образом:

```
struct DIGITAL_IO_INFO_LUSBAPI
{
    BOOL Active;                    // флаг достоверности остальных полей структуры
    BYTE Name[NAME_LINE_LENGTH_LUSBAPI];      // название цифровой микросхемы
    WORD InLinesQuantity;                     // кол-во входных линий
    WORD OutLinesQuantity;                    // кол-во выходных линий
    BYTE Comment[COMMENT_LINE_LENGTH_LUSBAPI];      // строка комментария
};
```