L-CARD

Устройства для мобильных систем

E14-440

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

Библиотека *Lusbapi 3.2*. Windows

Руководство программиста

Москва. Май 2008 г.

Ревизия документа A0

ЗАО *«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*

Представители в регионах:

Украина:	“ХОЛИТ Дэйта Системс, Лтд”	*www.holit.com.ua*	380 (44) 241-67-54
Санкт-Петербург:	ЗАО “АВТЭКС Санкт-Петербург”	*www.autex.spb.ru*	(812) 567-7202
Санкт-Петербург:	Компания "Ниеншанц-Автоматика"	*www.nnz-ipc.ru*	(812) 326-59-24
Новосибирск:	ООО “Сектор Т”	*www.sector-t.ru*	(3832) 22-76-20
Екатеринбург:	Группа Компаний АСК	*www.ask.ru*	(343) 371-44-44
Екатеринбург:	ООО “Авеон”	*www.aveon.ru*	(343) 381-75-75
Казань:	ООО “Шатл”	*shuttle@kai.ru*	(8432) 38-16-00
Самара:	"АСУ-Самара"	*prosoft-s@jiguli.ru*	(846)-998-29-01

Оглавление

1. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

1.1. Введение

Данный раздел описания предназначен для программистов, собирающихся писать свои собственные приложения в среде *Windows'98/2000/XP/Vista* для работы с модулями *E14-440*. Предварительно настоятельно рекомендуется внимательно ознакомиться с *"E14-440. 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 *E14-440* фирма **ЗАО "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 *E14-440* на низком уровне, то библиотека `Lusbapi` может быть использована в качестве законченного и отлаженного руководства, на основе которого можно реализовывать свои собственные алгоритмы, включая алгоритмы реального времени.

Модуль *E14-440* разрабатывался с главной целью – обеспечить быстрый бессбойный ввод в компьютер аналоговой информации. Для этого штатная библиотека `Lusbapi` содержит целый ряд функций, позволяющих организовывать многоканальный *непрерывный потоковый* сбор аналоговых данных на частотах АЦП вплоть до 400 кГц. При вводе данных можно использовать определённый вид синхронизации: цифровую или аналоговую. Наряду с вводом данных, библиотека позволяет организовывать *непрерывный потоковый* вывод аналоговой информации в одно- или двухканальном режиме (только при наличии микросхемы ЦАП на модуле). Кроме того, библиотека содержит также ряд функций, позволяющий осуществлять ввод-вывод аналоговой и цифровой информации в однократном, и поэтому достаточно медленном, режиме. Также пользователь может осуществлять конфигурацию на уровне DSP модуля *FIFO* буферов для АЦП и ЦАП, также многое другое. Мы надеемся, что описываемая ниже библиотека `Lusbapi` упростит и ускорит написание Ваших собственных приложений.

Полный пакет штатного программного обеспечения модуля *E14-440* для работы в среде *Windows'98/2000/XP/Vista* находится на прилагаемом к модулю фирменном CD-ROM'e в директории `\USB\Lusbapi`. **!!!ВНИМАНИЕ!!!** Далее по тексту данного описания все директории, касающиеся непосредственно модуля *E14-440*, указаны относительно неё. Также весь пакет штатного программного обеспечения можно скачать с нашего сайта *www.lcard.ru* из раздела *"Библиотека файлов"*. Там из подраздела *"ПО для внешних модулей"* 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 **3.2**, а содержащий её архив называется *lusbapi32.exe*.

1.2. Общие сведения

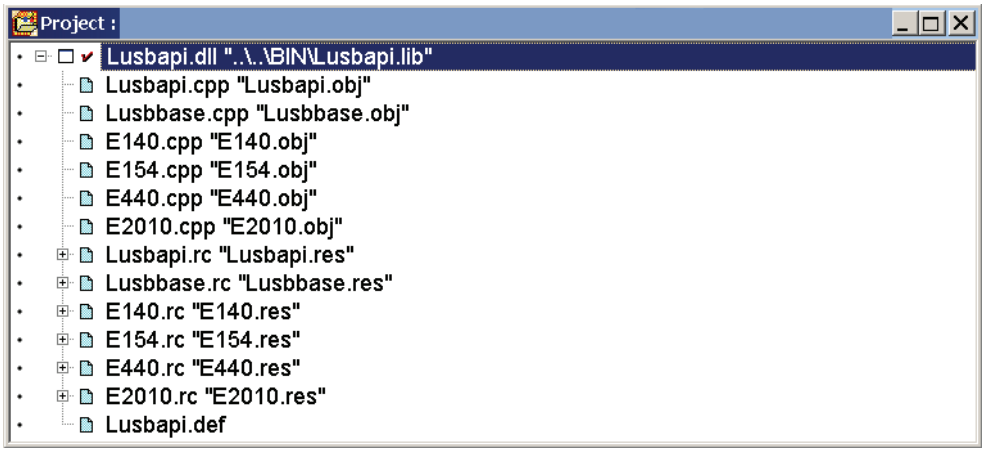
1.2.1. Подключение модуля E14-440 к компьютеру

Все подробности по процедуре аппаратного подключения модуля *E14-440* к компьютеру пользователя и надлежащей установке **USB** драйверов можно найти в *"E14-440. Руководство пользователя, § 2 "Инсталляция и настройка"*.

Стоит особо подчеркуть, 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**, конечному пользователю следует через *"Device Manager*" ("*Диспетчер устройств*") переключить модуль *E14-440* на работу с новым **USB** драйвером.

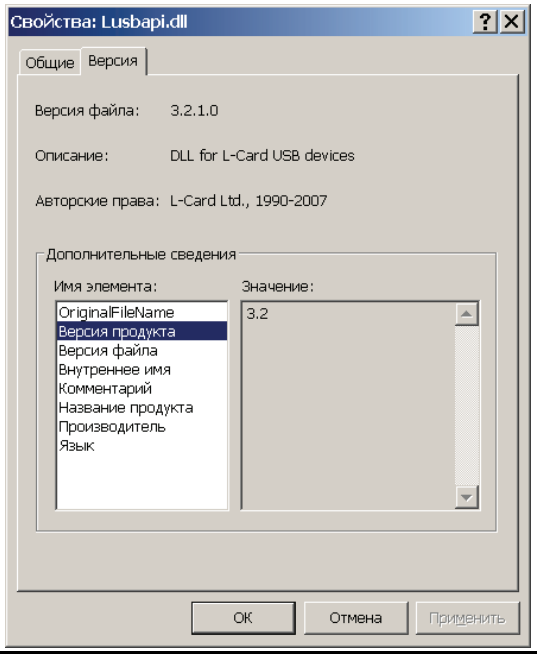
1.2.2. Библиотека Lusbapi

Штатная библиотека `Lusbapi` написана с использованием весьма доступного языка программирования **Borland C++ 5.02**. Кроме модуля *E14-440* в библиотеке также осуществлена поддержка модулей типа *E14-140*, *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 *CreateLInstance()* возвращает указатель на интерфейс модуля *E14-440*. В дальнейшем, используя этот указатель, можно осуществлять доступ ко всем интерфейсным функциям DLL библиотеки (см. исходные тексты примеров). **!!!Внимание!!!** Все интерфейсные функции, кроме *ReadData()* и *WriteData()*, 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 конечном 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 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 прочитать текущий номер версии библиотеки:



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:

- *E14-440*\Examples\Borland C++ 5.02;
- *E14-440*\Examples\Borland C++ Builder 5.0;
- *E14-440*\Examples\Borland Delphi 6.0;
- *E14-440*\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, test.ide для среды **Borland C++ 5.02**,;
- добавить файл библиотеки импорта \DLL\Lib\Borland\LUSBAPI.LIB;
- создать и добавить в проект приложения основной файл с будущей программой, например, test.cpp;
- включить в начало вашего файла заголовочный файл #include "LUSBAPI.H", содержащий описание интерфейса модуля *E14-440*;
- далее желательно сравнить версию используемой библиотеки Lusbapi с версией текущего программного обеспечения, для чего можно задействовать 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 библиотеки Lusbapi.dll.

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

1. Динамическая загрузка штатной DLL на этапе выполнения приложения. Подробности смотри в исходных текстах примера из директории *E14-440*\Examples\MicroSoft Visual C++ 6.0\DynLoad.
2. При статической компоновке штатной DLL в Вашем проекте использовать файл библиотеки импорта LUSBAPI.LIB из директории \DLL\Lib\MicroSoft.

При работе с модулем типа *E20-10* 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.

1.2.3. Загрузка управляющей программы

Предположим, что Вы уже успешно подключили модуль к компьютеру и подали на его входы сигналы. При работе с модулем следует учитывать то, что он имеет характерную особенность, отличающую его от более простых устройств ввода-вывода: на нем установлен цифровой сигнальный процессор *ADSP-2185M* от фирмы *Analog Devices, Inc.* Этот процессор необходимо предварительно запрограммировать, т.е. загрузить в него управляющую программу (драйвер DSP, *LBIOS*). В состав штатного программного обеспечения входит законченная управляющая программа, состоящая из одного бинарного файла *E14-440*\DSP\E440.bio. Данный файл содержит как выполняемый код управляющей программы, так и сегмент данных для сигнального процессора. В штатной библиотеке Lusbap для загрузки *LBIOS* в сигнальный процессор модуля имеется специальная интерфейсная функция *LOAD_MODULE()*, которая аккуратно выполняет процедуру загрузки модуля. Только **ПОСЛЕ** загрузки *LBIOS*'а приложение может переходить к управлению модулем, т.е. переводить его в различные режимы работы с АЦП, ЦАП и т. д.

1.2.4. Возможные проблемы при работе с модулем

1. Перед началом работы со штатным программным обеспечением модуля *E14-440*, 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, или скачать 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'98/2000/XP/Vista*. 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*.

1.3. Используемые термины и форматы данных

1.3.1. Термины

Название	Смысл
AdcRate	Частота работы АЦП в *кГц*
InterKadrDelay	Межкадровая задержка в *млс*
KardRate	Частота кадра отсчётов в *кГц*.
DacRate	Частота работы ЦАП 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
Address	Адрес ячейки в памяти программ или данных DSP модуля

1.3.2. Форматы данных

1.3.2.1. Формат слова данных с АЦП

Данные, считанные с $14^{ти}$ битного АЦП модуля *E14-440*, представляются в формате знакового целого двухбайтного числа от -8192 до 8191. Точностные пределы кодов АЦП, соответствующие выбранному входному диапазону, приведены в следующей таблице:

Таблица 1. Соответствие кода АЦП напряжению на аналоговом входе

Модуль	Усиление	Напряжение, В	Код	Точность, %
E14-440	1; 4; 16; 64	+MAX	+8000	2÷3
		0	0	0.25; 0.3; 0.5; 1.0
		-MAX	-8000	2÷3

где MAX – значение входного диапазона (см. *Таблица 6*).

Вышеуказанные точностные значения приведены для случая, когда *LBIOS* модуля не корректирует поступаемые с АЦП данные с помощью калибровочных коэффициентов (например, хранящихся в ППЗУ самого модуля; см. *§ 1.3.3. "Формат пользовательского ППЗУ"*). Для случая, когда *LBIOS*'у модуля предписано производить такую корректировку кодов входного сигнала, соответствующие точностные параметры АЦП приведены ниже (при температуре 25°C):

Таблица 2. Соответствие кода АЦП напряжению на аналоговом входе при разрешенной корректировке входных данных

Модуль	Усиление	Напряжение, В	Код	Точность, %
E14-440	1; 4; 16; 64	+MAX	+8000	0.05; 0.075; 0.1; 0.15
		0	0	
		-MAX	-8000	

где MAX – значение входного диапазона (см. *Таблица 6*).

1.3.2.2. Формат слова данных для ЦАП

На модуле по желанию пользователя может быть установлена микросхема 2^{x} канального $12^{ти}$ битного ЦАП. Формат $16^{ти}$ битного слова данных, передаваемого из PC в модуль для последующей выдачи на ЦАП, приведен в следующей таблице:

Таблица 3. Формат слова данных ЦАП

Модуль	Номер бита	Назначение
E14-440	0÷11	$12^{ти}$ битный код ЦАП
	12	Выбор номера канала ЦАП: ✓ '0' – первый канал; ✓ '1' – второй канал.
	13÷15	Не используются

Собственно сам код, выдаваемый модулем на $12^{ти}$ битный ЦАП, связан с устанавливаемым на внешнем разъеме напряжением в соответствии со следующей таблицей

Таблица 4. Соответствие кода ЦАП напряжению на внешнем аналоговом разъёме

Модуль	Код	Напряжение
E14-440	+2047	+5.0 Вольт
	0	0.0 Вольт
	-2048	-5.0 Вольт

1.3.2.3. Логический номер канала АЦП

Для управления работой входного аналогового каскада модуля *E14-440* 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 5. Формат логического номера канала.

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

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 для модуля *E14-440* равный 0x2 означает дифференциальный режим работы 3его 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 6. Коэффициент усиления (биты GS0 и GS1)

Модуль	Бит **GS1**	Бит **GS0**	Усиление	Диапазон, В
E14-440	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.

1.3.3. Формат пользовательского ППЗУ

На модуле *E14-440* установлено пользовательское ППЗУ емкостью 64 Слова×16 бит. Формат данного ППЗУ представлен на следующем рисунке:

Номер ячейки в ППЗУ

Как видно из рисунка, в первых 32^{x} словах (64 байта) находится служебная область ППЗУ модуля. Формат расположения служебной информации о модуле (первые 20 ячеек ППЗУ) имеет следующий вид:

- ✓ серийный номер модуля (9 байт);
- ✓ название модуля (7 байт);
- ✓ ревизия модуля (1 байт);
- ✓ тип установленного на модуле DSP (5 байт);
- ✓ флажок присутствия ЦАП на модуле (1 байт);
- ✓ частота установленного на модуле кварца в Гц (4 байта);
- ✓ зарезервировано (13 байт);

В следующих 8 словах (16 байт) хранятся коэффициенты, используемые при корректировке драйвером DSP данных, получаемых с АЦП. Данные коэффициенты записываются в ППЗУ при наладке модуля в **ЗАО "Л-Кард"**. Благодаря этому на модуле отсутствуют подстроечные резисторы, что сильно улучшает шумовые характеристики модуля и увеличивает их надежность. Для хранения калибровочных коэффициентов использован специальный дробный формат 1.15, который предназначен специально для работы с *LBIOS*. Т.о. коэффициенты хранятся в виде чисел типа *WORD* языка C++ (2 байта) и имеют следующий порядок:

- ✓ 20 ячейка – корректировка смещения нуля при усилении '1';
- ✓ 21 ячейка – корректировка смещения нуля при усилении '4';
- ✓ 22 ячейка – корректировка смещения нуля при усилении '16';
- ✓ 23 ячейка – корректировка смещения нуля при усилении '64';
- ✓ 24 ячейка – корректировка масштаба при усилении '1';
- ✓ 25 ячейка – корректировка масштаба при усилении '4';
- ✓ 26 ячейка – корректировка масштаба при усилении '16';
- ✓ 27 ячейка – корректировка масштаба при усилении '64';

В ячейках 28÷31 (8 байт) хранятся коэффициенты, используемые для корректировки данных, выводимых на ЦАП. Данные коэффициенты записываются в ППЗУ при наладке модуля в **ЗАО "Л-Кард"**. Коэффициенты хранятся в специальном формате в виде чисел типа *WORD* языка C++ (2 байта) и имеют следующий порядок:

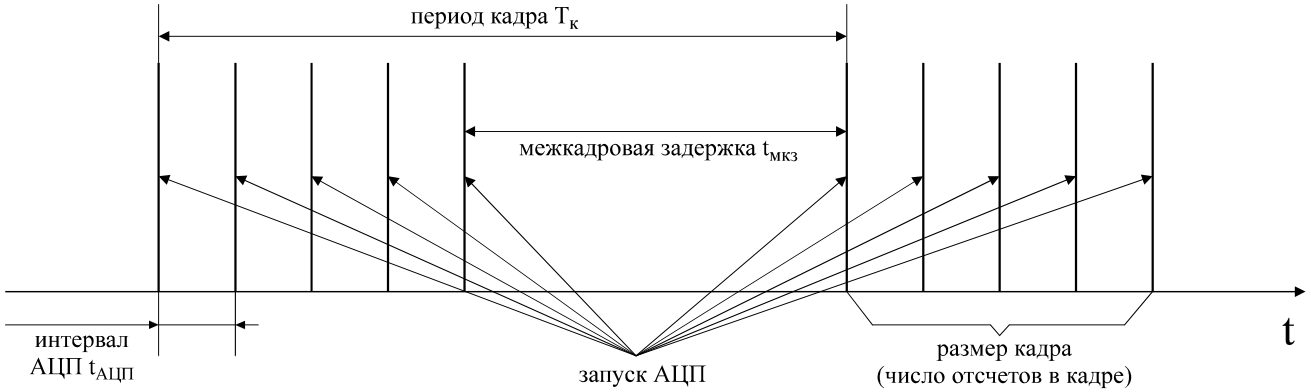
- ✓ 28 ячейка – корректировка смещения нуля первого ЦАП'а;
- ✓ 29 ячейка – корректировка смещения нуля второго ЦАП'а;
- ✓ 30 ячейка – корректировка масштаба первого ЦАП'а;
- ✓ 31 ячейка – корректировка масштаба второго ЦАП'а;

Все приведённые корректировочные коэффициенты преобразуются к более привычному формату типа *double* в соответствующих полях структуры *MODULE_DESCRIPTION_E440*, когда служебная область ППЗУ зачитывается интерфейсной функцией *GET_MODULE_DESCRIPTION()*.

Пользовательская область ППЗУ начинается с $32^{ой}$ ячейки. Туда совершенно спокойно можно записывать или считывать любую свою информацию. Для этого следует воспользоваться соответствующими интерфейсными функциями: *WRITE_FLASH_WORD()* и *READ_FLASH_WORD()*.

1.3.4. 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 штатной интерфейсной функции *SET_ADC_PARS()*. Подробнее об этой функции смотри *§ 1.5.4.3. "Установка параметров работы АЦП"*. Временные параметры кадра для случая **ControlTableLength = 5** приведены на следующем рисунке:



где $\mathbf{T_k}$ – временной интервал между соседними кадрами, т.е. это фактически частота опроса фиксированного логического номера канала **KardRate**; $\mathbf{t_{мкз}}$= **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

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

При выборе необходимого режима функционирования сбора данных с АЦП, такие важные временны**е** параметры как **AdcRate** и **InterKadrDelay** задаются через штатную интерфейсную функцию *SET_ADC_PARS()*.

1.4. Общие принципы работы с модулем E14-440

1.4.1. Общий подход к работе с интерфейсными функциями

Целью штатной библиотеки Lusbapi, поставляемой с модулем *E14-440*, 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):

```
ILE440 *pModule;                    // указатель на интерфейс модуля E14-440
MODULE_DESCRIPTION_E440 md;         // структура служебной информации о модуле
```

Первым делом с помощью функции *GetDllVersion()* следует сравнить версии используемой библиотеки Lusbapi.dll 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 *E14-440*. Для этого предусмотрена интерфейсная функция *OpenLDevice()*. Если нет ошибки при выполнении этой функции, то можно быть уверенным, что устройство типа *E14-440* обнаружено в выбранном виртуальном слоте.

Теперь, в принципе, можно переходить к этапу загрузки обнаруженного модуля, 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.

Важной особенностью модуля *E14-440* является то, что на нем установлен достаточно мощный современный цифровой сигнальный процессор (DSP – Digital Signal Processor) с фиксированной точкой *ADSP-2185M* от фирмы *Analog Devices, Inc.* Для того, чтобы его “оживить”, т.е. заставить работать по требуемому алгоритму, во внутреннюю память DSP надо записать (загрузить) либо фирменную управляющую программу, которая входит в комплект штатной поставки (файл *\E14-440*\DSP\E440.bio), либо Вашу собственную. Задачей DSP является управление всей установленной на модуле периферией (АЦП, ЦАП, цифровые линии и т.д.), а также сбор и, при необходимости, первичная обработка получаемых данных. Во внутренней памяти штатного драйвера DSP расположены программно организованные *FIFO* буфера АЦП и ЦАП, а также переменные *LBIOS* (см. *Приложение А*). О низкоуровневом взаимодействии компьютера с модулем *E14-440*, с одной стороны, и DSP с периферией, с другой, см. *Раздел 2 "НИЗКОУРОВНЕВОЕ ОПИСАНИЕ МОДУЛЯ E14-440"*

Т.о., необходимо загрузить в сигнальный процессор модуля управляющую программу (*LBIOS*). Для этого можно воспользоваться интерфейсной функцией *LOAD_MODULE()*. В случае успешного выполнения данной функции, нужно проверить работоспособность загруженного *LBIOS* с помощью интерфейсной функции *MODULE_TEST()*. Если и эта функция выполнена без ошибки, то это означает, что *LBIOS* успешно загружен и модуль полностью готов к работе.

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 продолжить работу.

В общем-то, **ВСЁ!** Теперь можно спокойно управлять всей доступной периферией на модуле и самим DSP с помощью соответствующих интерфейсных функций штатной библиотеки. Т.е. задавать различные режимы работы АЦП (приём данных с АЦП, конфигурация *FIFO* буфера АЦП,

синхронизация ввода данных с АЦП, частота оцифровки данных и т.д.) и ЦАП (конфигурация FIFO буфера ЦАП, частота выдачи данных на ЦАП и т.д.), обрабатывать входные и выходные цифровые линии, считывать и/или записывать необходимую информацию в/из пользовательского ППЗУ и т.д.

В качестве примера приведем исходный текст, а вернее сказать '*скелет*', консольной программы для работы с ***E14-440***, предполагая использование `Lusbapi` версии не ниже 3.0:

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

ILE440 *pModule;                // указатель на интерфейс модуля
MODULE_DESCRIPTION_E440 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<ILE440 *>(CreateLInstance("e440"));
    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 - 'E14-440'?
    if(strcmp(ModuleName, "E440"))
    {
      printf(" В нулевом виртуальном слоте не 'E14-440'\n");
      return 1;                  //выйдем из программы с ошибкой
    }
    // теперь можно попробовать загрузить из соответствующего ресурса
    // библиотеки Lusbapi код драйвера LBIOS
    if(!pModule->LOAD_MODULE())
    {
      printf("Не выполнена функция LOAD_MODULE()!");
      return 1;                  //выйдем из программы с ошибкой
    }
    // проверим работоспособность загруженного LBIOS
    if(!pModule->MODULE_TEST())
    {
      printf("Не выполнена функция MODULE_TEST()!");
      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 E14-440 (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;
}
```

1.4.2. Общая структура драйвера DSP

На модуле ***E14-440*** устанавливается так называемый цифровой сигнальный процессор (Digital Signal Processor – DSP) с фиксированной точкой *ADSP-2185M* от фирмы *Analog Devices, Inc.* Основное его назначение – это управление различного рода периферийными устройствами, установленными на модуле, а также, возможно, выполнение необходимой предварительной обработки данных. Одно из главных преимуществ применения на модуле именно цифрового сигнального процессора заключается в том, что достаточно гибко чисто программным образом можно изменять в довольно широких пределах алгоритмы работы модуля с периферийными устройствами. Для этого достаточно лишь овладеть достаточно несложным языком ассемблера DSP. Так, в штатном драйвере *LBIOS*, исходные тексты которого можно найти в директории *\E14-440*DSP\ на

прилагаемом к данному модулю фирменном CD-ROM, реализуются наиболее широко используемые алгоритмы работы с АЦП, ЦАП, входными/выходными ТТЛ линиями и т.д.

В принципе, для написания пользовательских приложений на PC достаточно знать, что установленный на модуле DSP обладает двумя независимыми типами памяти, а именно:

- ✓ $24^{х}$ битная память программ (Program Memory – PM), в которой хранятся коды инструкций управляющей программы (драйвера *LBIOS*), а также, возможно, данные;
- ✓ $16^{ти}$ битная память данных (Data Memory – DM), в которой могут находиться только данные.

Для доступа к содержимому ячеек DSP каждого типа памяти существуют штатные интерфейсные функции, которым посвящён *§ 1.5.3. "Функции для доступа к памяти DSP модуля"*. Карты распределения памяти обоих типов для различных типов сигнальных процессоров, а также взаимное расположение составных частей штатного *LBIOS*, подробно показаны в *Приложении В*. Как видно из указанного приложения, *LBIOS* состоит из:

- ✓ двух областей в PM с исполняемыми кодами инструкций и переменными *LBIOS*;
- ✓ двумя областями в DM под циклические *FIFO* буфера АЦП и ЦАП.

Исполняемый код *LBIOS* написан с учетом возможности взаимодействия данного драйвера с пользовательским приложением в PC по так называемому принципу команд. Об этом подробнее можно посмотреть в *§ 2.1. "Структурная схема модуля E14-440"*.

Адреса предопределенных переменных в PM DSP модуля, задающих важные параметры функционирования штатного *LBIOS*, а также их краткие толкования, приведены в *Таблице 7*.

В *LBIOS* программно организовано два циклических *FIFO* буфера: для приёма данных с АЦП и для вывода данных на ЦАП. Передача данных из *FIFO* буфера АЦП в PC производится порциями по *Длина_FIFO_Буфера_АЦП/2* отсчётов по мере их поступления с АЦП. Т.о., фактически чисто программным образом реализован так называемый двойной *FIFO* буфер. Т.е. при поступлении из PC команды на запуск АЦП, драйвер *LBIOS* ожидает накопления данных в первой половинке *FIFO* буфера АЦП. После того как первая половинка буфера полностью заполнится готовыми данными с АЦП, даётся команда на их передачу в PC. При этом не прекращается сбор данных во вторую половинку *FIFO* буфера. После накопления данных во второй половинке *FIFO* буфера опять дается команда на их передачу в PC и продолжается сбор данных уже в первую половинку. И так до бесконечности по циклу, пока не придет команда из PC на останов работы АЦП. Все то же самое применимо и для алгоритма работы ЦАП.

Драйвер *LBIOS* написан таким образом, что можно независимо управлять работой АЦП, ЦАП и ТТЛ линиями.

1.5. Описание библиотеки Lusbapi

В настоящем разделе приведены достаточно подробные описания констант, структур и интерфейсных функций, входящих в состав штатной библиотеки `Lusbapi` для модуля *E14-440*.

1.5.1. Константы

Приведённые ниже основные константы настоятельно рекомендуется использовать в исходных текстах приложения при работе с модулем *E14-440*. Это весьма повышает *читаемость* и *понимаемость* исходных текстов, а также значительно облегчает сопровождение программ. Рассматриваемые константы расположены в файле `\DLL\Include\Lusbapi.h`.

1. Модулю *E14-440* доступны несколько режимов сброса устройства. Местом использования этих констант, как правило, является аргумент интерфейсной функции *RESET_MODULE()*.

Константа	Значение	Назначение
INIT_E440	0	Модуль подвергается полной переинициализации. После чего DSP модуля снова готов к загрузке.
RESET_E440	1	Это константа действует на модуль *E14-440* с ревизией ‘E’ и выше. При этом модуль подвергается процедуре полного сброса. После чего DSP находится в состоянии сброса, вторичное питание отключено. Это состояние с пониженным энергопотреблением.
INVALID_RESET_TYPE_E440	2	Неправильный тип сброса модуля.

3. У модуля *E14-440* 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_E440*.

Константа	Значение	Назначение
ADC_INPUT_RANGE_10000mV_E440	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_E440*.
ADC_INPUT_RANGE_2500mV_E440	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_E440*.
ADC_INPUT_RANGE_625mV_E440	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_E440*.

ADC_INPUT_RANGE_156mV_E440	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_E440*.

2. Модуль *E14-440* 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_E440*.

Константа	Значение	Назначение
NO_SYNC_E440	0	***Отсутствие синхронизации ввода.*** Сбор данных с АЦП модуль начинает непосредственно после выполнения функции *START_ADC()*.
TTL_START_SYNC_E440	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 **TRIG** аналогового разъёма *DRB-37M*. Длительность этого синхроимпульса должна быть не менее 50 нс. Только после этого модуль приступает к сбору данных.
TTL_KADR_SYNC_E440	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 **TRIG** аналогового разъёма *DRB-37M*. Длительность этого синхроимпульса должна быть не менее 50 нс. После прихода синхроимпульса (см. выше) модуль собирает отсчеты только одного кадра. Во время ввода кадра данных, вся активность на линии **TRIG** игнорируется. Только после окончания сбора текущего кадра данных ожидается приход следующего импульса синхронизации и т.д.
ANALOG_SYNC_E440	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_E440	4	Неправильный вид синхронизации ввода данных.

4. На модуле *E14-440* по желанию пользователя может быть установлена микросхема двуканального ЦАП. Состояние поля *Dac.Active* структуры служебной информации *MODULE_DESCRIPTION_E440* отражает факт наличия ЦАП на борту модуля.

Константа	Значение	Назначение
DAC_INACCESSIBLED_E440	0	На модуле полностью отсутствует микросхема ЦАП.
DAC_ACCESSIBLED_E440	1	На модуле присутствует микросхема ЦАП.

5. Ревизия модуля *E14-440* отражает определённые конструктивные особенности модуля. Она задаётся одной заглавной латинской буквой. Например, *первая* ревизия модуля обозначается как ***'A'***.

Константа	Значение	Назначение
REVISION_A_E440	0	Данная константа может использоваться в качестве индекса для доступа к первому элементу константного массива *REVISIONS_E440*.
REVISION_B_E440	1	Данная константа может использоваться в качестве индекса для доступа к второму элементу константного массива *REVISIONS_E440*.
REVISION_C_E440	2	Данная константа может использоваться в качестве индекса для доступа к третьему элементу константного массива *REVISIONS_E440*.
REVISION_D_E440	3	Данная константа может использоваться в качестве индекса для доступа к четвёртому элементу константного массива *REVISIONS_E440*.
REVISION_E_E440	4	Данная константа может использоваться в качестве индекса для доступа к пятому элементу константного массива *REVISIONS_E440*.

6. Различные константы для работы с модулем *E14-440*.

Константа	Значение	Назначение
CURRENT_VERSION_LUSBAPI	——	Версия используемой библиотеки `Lusbapi`. Как правило, используется совместно с функцией *GetDllVersion()*.
MAX_CONTROL_TABLE_LENGTH_E440	128	Максимально возможное кол-во логических каналов в управляющей таблице **ControlTable**.
ADC_CALIBR_COEFS_QUANTITY_E440	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_E440	4	Кол-во входных диапазонов.
MAX_ADC_FIFO_SIZE_E440	12288	Максимальный размер FIFO буфера АЦП в DSP модуля.
DAC_CHANNELS_QUANTITY_E440	2	Кол-во физических каналов ЦАП на модуле (при условии наличия микросхемы ЦАП на модуле).
DAC_CALIBR_COEFS_QUANTITY_E440	2	Кол-во корректировочных коэффициентов для данных ЦАП. По одному на каждый канал. Корректировке подвергаются как смещение, так и масштаб данных ЦАП.
MAX_DAC_FIFO_SIZE_E440	4032	Максимальный размер FIFO буфера ЦАП в DSP модуля.
TTL_LINES_QUANTITY_E440	16	Кол-во входных и выходных цифровых линий.
REVISIONS_QUANTITY_E440	5	Кол-во ревизий (модификаций) модуля.

7. Различные *константные массивы* для работы с модулем *E14-440*.
   a. Массив доступных диапазонов входного напряжения АЦП в Вольтах:
   ```
   const double
           ADC_INPUT_RANGES_E440[ADC_INPUT_RANGES_QUANTITY_E440] =
           {
                   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_E440 = 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_E440*. Массив доступных ревизий модуля задаётся следующим образом.
   ```
   const BYTE REVISIONS_E440[REVISIONS_QUANTITY_E440] =
   {
           'A', 'B', 'C', 'D', 'E'
   };
   ```

1.5.2. Структуры

В данном разделе приведены основные типы структур, которые применяются в библиотеке `Lusbapi` при работе с модулем *E14-440*.

1.5.2.1. Структура MODULE_DESCRIPTION_E440

Структура *MODULE_DESCRIPTION_E440* описана в файле `\DLL\Include\Lusbapi.h` и представлена ниже:

```
struct MODULE_DESCRIPTION_E440
{
    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
    DSP_INFO_LUSBAPI        Dsp;             // информация о DSP
    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 *E14-440*. 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 С*.

1.5.2.2. Структура ADC_PARS_E440

Структура *ADC_PARS_E440* описана в файле `\DLL\Include\Lusbapi.h` и представлена ниже:

```
struct ADC_PARS_E440
{
    BOOL IsAdcEnabled;              // состояние работы АЦП (только на чтение)
    BOOL IsCorrectionEnabled;       // управление корректировкой данных
    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
    WORD AdcFifoBaseAddress;        // базовый адрес FIFO буфера АЦП в DSP модуля
    WORD AdcFifoLength;             // длина FIFO буфера АЦП в DSP модуля
    double AdcOffsetCoefs[ADC_CALIBR_COEFS_QUANTITY_E440];
                                    // корректировка смещения АЦП: 4диапазона
    double AdcScaleCoefs[ADC_CALIBR_COEFS_QUANTITY_E440];
                                    // корректировка масштаб АЦП: 4диапазона
};
```

Перед началом работы с АЦП необходимо заполнить поля данной структуры и передать её в модуль с помощью интерфейсной функции *SET_ADC_PARS()*. В описании этой функции подробно прокомментированы смысл и назначение всех полей данной структуры. При этом в качестве

корректировочных коэффициентов можно использовать значения из соответствующих полей структуры *MODULE_DESCRIPTION_E440*. При необходимости можно считать из модуля текущие параметры функционирования АЦП с помощью интерфейсной функции *GET_ADC_PARS()*.

1.5.2.3. Структура DAC_PARS_E440

Структура *DAC_PARS_E440* описана в файле `\DLL\Include\Lusbapi.h` и представлена ниже:

```
struct DAC_PARS_E440
{
    BOOL DacEnabled;              // состояние работы ЦАП (только на чтение)
    double DacRate;               // частота работы ЦАП в кГц
    WORD DacFifoBaseAddress;      // базовый адрес FIFO буфера ЦАП в DSP модуля
    WORD DacFifoLength;           // длина FIFO буфера ЦАП в DSP модуля
};
```

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 назначение всех полей данной структуры. Также при необходимости можно считать из модуля текущие параметры функционирования ЦАП, используя *GET_DAC_PARS()*.

1.5.2.4. 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.

1.5.2.5. 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`.

1.5.3. Драйвер DSP

1.5.3.1. Переменные драйвера DSP

Любой программист при желании может напрямую работать с памятью DSP модуля *E14-440* (и программ, и данных), используя соответствующие интерфейсные функции, которые обеспечивают доступ, как к отдельным ячейкам памяти, так и к целым массивам. Эта возможность позволяет программисту работать с модулем, непосредственно обращаясь к соответствующим ячейкам либо памяти программ, либо памяти данных DSP. Карта распределения памяти программ и данных для *ADSP-2185M*, который установлен на модуле, приведена в *Приложении А*.

Ниже в *Таблице 7* приводятся *предопределенные* адреса переменных штатного *LBIOS*, расположенных в памяти **программ** DSP, и их краткие описания. Собственно сами переменные *LBIOS* являются $16^{ти}$ битными и располагаются в старших $16^{ти}$ битах $24^{х}$ битного слова памяти программ DSP (при этом младшие 8 из этих $24^{х}$ бит не используются). Программист может напрямую обращаться к переменным штатного *LBIOS*, чтобы при необходимости считать и/или изменить их содержимое с помощью интерфейсных функций *GET_LBIOS_WORD()* и *PUT_LBIOS_WORD()* (см. *§ 1.5.3 "Функции для доступа к памяти DSP модуля"*).

Таблица 7. Адреса управляющих переменных драйвера DSP

Название переменной	Адрес Hex	Назначение переменной
L_READY_E440	0x31	После загрузки показывает готовность модуля к дальнейшей работе
L_TMODE1_E440	0x32	Тестовая переменная. После загрузки драйвера (*LBIOS*) по этому адресу должно читаться число 0x5555.
L_TMODE2_E440	0x33	Тестовая переменная. После загрузки драйвера (*LBIOS*) по этому адресу должно читаться число 0xAAAA.
L_TEST_LOAD_E440	0x34	Тестовая переменная.
L_COMMAND_E440	0x35	Переменная, при помощи которой драйверу передается номер команды, которую он должен выполнить. Краткое описание номеров команд приведено ниже в *Таблице 8*.
L_DAC_SCLK_DIV_E440	0x37	Текущий делитель для *SCLK0*. Используется при работе с ЦАП.
L_DAC_RATE_E440	0x38	Переменная, задающая код частоты вывода данных на ЦАП.
L_ADC_RATE_E440	0x39	Переменная, задающая код частоты работы АЦП.
L_ADC_ENABLED_E440	0x3A	Переменная, показывающая текущее состояние АЦП (работает или нет).
L_ADC_FIFO_BASE_ ADDRESS_E440	0x3B	Текущий базовый адрес FIFO буфера АЦП, который всегда должен быть равен 0x0.

L_CUR_ADC_FIFO_ LENGTH_E440	0x3C	Текущая длина FIFO буфера АЦП. По умолчанию **L_CUR_ADC_FIFO_LENGTH_E440 = 0x3000**.
L_ADC_FIFO_ LENGTH_E440	0x3E	Требуемая длина FIFO буфера АЦП. По умолчанию **L_ADC_FIFO_LENGTH_E440 = 0x3000**.
L_CORRECTION_ ENABLED_E440	0x3F	Переменная запрещающая (0)/ разрешающая (1) корректировку данных аналоговых каналов при помощи калибровочных коэффициентов. По умолчанию **L_CORRECTION_ENABLED = 0x0**.
L_ADC_SAMPLE_E440	0x41	Данная переменная используется при однократном вводе с АЦП, храня оцифрованное значение.
L_ADC_CHANNEL_E440	0x42	Данная переменная используется при однократном вводе с АЦП, задавая логический номер канала.
L_INPUT_MODE_E440	0x43	Переменная, задающая тип синхронизации при вводе данных с АЦП (аналоговая, цифровая или ее отсутствие).
L_SYNCHRO_AD_ CHANNEL_E440	0x46	При аналоговой синхронизации задает логический номер канала, по которому происходит синхронизация.
L_SYNCHRO_AD_ POROG_E440	0x47	При аналоговой синхронизации задает порог срабатывания в кодах АЦП.
L_SYNCHRO_AD_ MODE_E440	0x48	Переменная, задающая режим аналоговой синхронизации.
L_SYNCHRO_TYPE_E440	0x49	При аналоговой синхронизации задает тип срабатывания по уровню (**0x0**) либо по переходу (**0x1**).
L_CONTROL_TABLE_ LENGHT_E440	0x4B	Размер управляющей таблицы (максимум 128 логических каналов). По умолчанию – **0x8**.
L_FIRST_SAMPLE_ DELAY_E440	0x4C	Переменная, задающая код задержки для первого отсчета при старте АЦП
L_INTER_KADR_ DELAY_E440	0x4D	Переменная, задающая код межкадровой задержки при вводе данных с АЦП.
L_DAC_SAMPLE_E440	0x50	Переменная, задающая значение для однократного вывода на ЦАП
L_DAC_ENABLED_E440	0x51	Переменная, показывающая текущее состояние ЦАП (работает или нет).
L_DAC_FIFO_BASE_ ADDRESS_E440	0x52	Текущий базовый адрес FIFO буфера ЦАП, который всегда должен быть равен **0x3000**.
L_CUR_DAC_FIFO_	0x54	Текущая длина FIFO буфера ЦАП. По умолчанию

LENGTH_E440		**L_CUR_DAC_FIFO_LENGTH_E440 = 0xFC0**.
L_DAC_FIFO_ LENGTH_E440	0x55	Требуемая длина FIFO буфера ЦАП. По умолчанию **L_DAC_FIFO_LENGTH_E440 = 0xFC0**.
L_FLASH_ENABLED_E440	0x56	Флаг разрешения(1)/запрещения(0) записи в ППЗУ модуля
L_FLASH_ADDRESS_E440	0x57	Номер ячейки ППЗУ (от 0 до 63)
L_FLASH_DATA_E440	0x58	Данные в/из ППЗУ
L_ENABLE_TTL_OUT_E440	0x59	Данная переменная разрешает (0x1) либо запрещает (0x0) использование выходных цифровых линий (перевод их в третье состояние)
L_TTL_OUT_E440	0x5A	Слово (16 бит), в котором по-битово хранятся значения 16[ти] выходных цифровых линий для их выставления по команде **C_TTL_OUT_E440** (см. *Таблице 8*).
L_TTL_IN_E440	0x5B	Слово (16 бит), в котором после выполнения команды **C_TTL_IN_E440** (см. *Таблице 8*) по-битово хранятся значения 16[ти] входных цифровых линий.
L_SCALE_E440	0x60	Массив с 4[мя] калибровочными коэффициентами, используемый для корректировки масштаба данных с АЦП. По умолчанию – **{ 0x8000, 0x8000, 0x8000, 0x8000}**
L_ZERO_E440	0x64	Массив с 4[мя] калибровочными коэффициентами, используемый для корректировки смещения нуля данных с АЦП. По умолчанию – **{ 0x0, 0x0, 0x0, 0x0}**
L_CONTROL_TABLE_E440	0x80	Управляющая таблица, содержащая последовательность логических номеров каналов (максимум 128 элементов). В соответствии с ней DSP производит последовательный циклический сбор данных с АЦП. Размер этой таблицы задается переменной **L_CONTROL_TABLE_LENGHT_E440** (см. выше). По умолчанию – **{ 0, 1, 2, 3, 4, 5, 6, 7}**
L_DSP_INFO_STUCTURE_ E440	0x200	Информация о драйвере DSP, включая версию.

1.5.3.2. Номера команд драйвера DSP

Фирменный *LBIOS* для модуля *E14-440* работает по принципу команд, т.е. драйверу в модуль передается номер команды, которую он должен выполнить. Список доступных номеров команд для штатного *LBIOS* (версия 2.0 и выше) приведен ниже в таблице.

Таблица 8. Номера команд штатного драйвера DSP.

Название команды	Номер команды	Назначение	Используемые переменные
C_TEST_E440	0	Проверка загрузки модуля и его работоспособности.	**L_TEST_LOAD_E440**
C_ENABLE_FLASH_ WRITE_E440	1	Разрешение процедуры записи в пользовательское ППЗУ	**L_FLASH_ENABLED_ E440**
C_READ_FLASH_ WORD_E440	2	Чтение слова из ППЗУ	**L_FLASH_ADDRESS_ E440, L_FLASH_DATA_E440**
C_WRITE_FLASH_ WORD_E440	3	Запись слова в ППЗУ	**L_FLASH_ADDRESS_ E440, L_FLASH_DATA_E440**
C_START_ADC_E440	4	Разрешение работы АЦП	**--------**
C_STOP_ADC_E440	5	Запрещение работы АЦП	--------
C_ADC_KADR_E440	6	Получение кадра отсчетов с АЦП	--------
C_ADC_ SAMPLE_E440	7	Однократный ввод отсчета с АЦП для заданного канала	**L_ADC_ SAMPLE_E440, L_ADC_ CHANNEL_E440**
C_START_DAC_E440	8	Разрешение работы ЦАП	--------
C_STOP_DAC_E440	9	Запрещение работы ЦАП	--------
C_DAC_ SAMPLE_E440	10	Однократный вывод отсчета на ЦАП	**L_DAC_SAMPLE_E440**
C_ENABLE_TTL_ OUT_E440	11	Разрешение выходных цифровых линий	**L_ENABLE_TTL_ OUT_E440**
C_TTL_IN_E440	12	Считывание состояния 16<u>ти</u> внешних цифровых входных линий.	**L_TTL_IN_E440**
C_TTL_OUT_E440	13	Управление 16<u>тью</u> внешними цифровыми выходными линиями.	**L_TTL_OUT_E440**

1.5.4. Функции общего характера

1.5.4.1. 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. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры: нет

Возвращаемое значение: номер версии библиотеки `Lusbapi`.

1.5.4.2. Получение указателя на интерфейс модуля

Формат: **LPVOID** ***CreateInstance****(const char *DeviceName)* (версия 1.0)
LPVOID ***CreateLInstance****(PCHAR const DeviceName)* (с версии 3.0)

Назначение:

Данная функция должна обязательно вызываться в начале каждой пользовательской программы, работающей с модулями *E14-440*. 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. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры:

- *DeviceName* – строка с названием устройства (для данного модуля это – "E440") .

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

1.5.4.3. Завершение работы с интерфейсом модуля

Формат:	**bool**	***ReleaseLDevice**(void)*	(версия 1.0)
	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 избежания утечки ресурсов компьютера.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры: нет.

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

1.5.4.4. Инициализация доступа к модулю

Формат:	**bool**	***InitLDevice**(WORD VirtualSlot)*	(версия 1.0)
	BOOL	***OpenLDevice**(WORD VirtualSlot)*	(с версии 2.0)

Назначение:

С программной точки зрения, не вдаваясь в излишние тонкости, подсоединенный к компьютеру модуль *E14-440* 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 *E14-440* находится в заданном виртуальном слоте. Если функция ***OpenLDevice()*** успешно выполнилась для заданного виртуального слота, то можно переходить непосредственно к загрузке модуля и его последующему управлению с помощью соответствующих интерфейсных функций библиотеки `Lusbapi`.

InitLDevice() является устаревшим названием данной функции, хотя библиотекой по-прежнему поддерживается, кроме **Delphi**.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры:

- *VirtualSlot* – номер виртуального слота, к которому, как предполагается, подключен модуль *E14-440*.

Возвращаемое значение: *TRUE* – модуль *E14-440* находится в выбранном виртуальном слоте и можно переходить непосредственно к загрузке модуля;
FALSE – устройства типа модуль *E14-440* в выбранном виртуальном слоте нет. Следует попробовать другой номер виртуального слота.

1.5.4.5. 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 *E14-440* 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.

1.5.4.6. Загрузка модуля

Формат:	**bool**	***LOAD_LBIOS***(*PCHAR FileName*)	(версия 1.0)
	bool	***LOAD_LBIOS***(*PCHAR FileName = NULL*)	(с версии 2.0)
	BOOL	***LOAD_MODULE***(*PCHAR const FileName = NULL*)	(с версии 3.0)

Назначение:

Данная интерфейсная функция выполняет операцию загрузки драйвера (штатного *LBIOS* или Вашего) в DSP модуля. Файл *FileName* с кодом драйвера должен находиться в текущей директории Вашего приложения. С версии 2.0 в библиотеке появилась дополнительная возможность загружать *LBIOS*, содержимое которого хранится в самом теле библиотеки в виде соответствующего ресурса. Для этого достаточно параметр *FileName* задать в виде **NULL**. **NULL** является также значением по умолчанию для параметра *FileName*.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры:

- *FileName* – строка с названием файла, содержащим код загружаемой управляющей программы. Например, для штатного драйвера DSP это строка `"E440.BIO"`. Начиная с версии 2.0, если данный параметр задан как **NULL**, то загрузка модуля будет осуществляться тем *LBIOS*'ом, который находится в виде ресурса в теле штатной библиотеки.

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

1.5.4.7. Проверка загрузки модуля

Формат:	**bool**	***MODULE_TEST**(void)*	(версия 1.0)
	BOOL	***TEST_MODULE**(void)*	(с версии 3.0)

Назначение:

Данная интерфейсная функция проверяет правильность загрузки модуля и его работоспособность. **!!!Внимание!!!** Данная функция работает надлежащим образом только после выполнения интерфейсной функции *LOAD_MODULE()*.

Рекомендованную последовательность вызовов интерфейсных функций см. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры: нет

Возвращаемое значение:
TRUE – драйвер DSP успешно загружен и функционирует надлежащим образом,
FALSE – произошла ошибка загрузки или функционирования драйвера DSP.

1.5.4.8. 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. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*.

Передаваемые параметры:

- *ModuleName* – возвращается строка, не менее 6 символов, с названием модуля (в нашем случае это должна быть строка "E440").

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

1.5.4.9. 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.

1.5.4.10. Сброс модуля

Формат:	**bool**	***DSP_RESET**(void)*	(версия 1.0)
	BOOL	***RESET_MODULE**(BYTE ResetFlag = INIT_E440)*	(с версии 3.0)

Назначение:

Данная интерфейсная функция осуществляет сброс модуля. Эта процедура используется при перезагрузке драйвера DSP или для полной остановки работы модуля. Сброс может быть произведён различными способами в зависимости от значения переменной *ResetFlag*, а также можно использовать *константы сброса модуля*:

Значение	Константа	Назначение
0	**INIT_E440**	Модуль подвергается полной переинициализации. После чего DSP модуля снова готов к загрузке.
1	**RESET_E440**	Это константа действует только на модуль *E14-440* с ревизией ‘E’ и выше. При этом модуль подвергается процедуре полного сброса. После чего DSP находится в состоянии сброса, вторичное питание отключено. Это состояние с пониженным энергопотреблением.
2	**INVALID_RESET_TYPE_E440**	Неправильный тип сброса модуля.

Необходимо помнить, что после выполнения данной функции работа DSP модуля полностью останавливается. В дальнейшем для приведения модуля в ‘*чувство*’ требуется перезагрузить драйвер DSP, используя штатную функцию *LOAD_MODULE()*.

Передаваемые параметры:

- *ResetFlag* – режим сброса модуля.

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

1.5.4.11. Передача номер команды в драйвер DSP

Формат: **BOOL** **SEND_COMMAND***(WORD Command)*

Назначение:

Данная интерфейсная функция записывает в предопределенную переменную *L_COMMAND_E440* номер команды и вызывает командное прерывание *IRQE* в DSP модуля. В ответ на это прерывание *LBIOS* выполняет действия, строго соответствующие номеру переданной команды. **!!!Внимание!!!** Данная функция работает надлежащим образом только после успешного выполнения интерфейсных функций *LOAD_MODULE()* и *TEST_MODULE()*.

Передаваемые параметры:

- *Command* – номер команды, передаваемый в драйвер DSP.

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

1.5.4.12. Получение дескриптора устройства

Формат: **HANDLE** ***GetModuleHandle****(void)*
Назначение: Данная функция позволяет получить дескриптор (*handle*) используемого модуля *E14-440*.
Передаваемые параметры: нет
Возвращаемое значение: В случае успеха – дескриптор модуля *E14-440*; в противном случае – **INVALID_HANDLE_VALUE**.

1.5.4.13. Получение описания ошибок выполнения функций

Формат: **int** ***GetLastErrorString****(LPTSTR lpBuffer, DWORD nSize)* (версия 1.0) **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. **!!!Внимание!!!** Данная интерфейсная функция не выполняет классификацию ошибок для интерфейсных функций *ReadData()* и *WriteData()*. Т.к. эти функции фактически является слепком со стандартных *Windows API* функций ***ReadFile()*** и ***WriteFile()***, то для выявления ошибок следует пользоваться классификацией ошибок, присущей системе *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.

1.5.5. Функции для доступа к памяти DSP модуля

Интерфейсные функции данного раздела обеспечивают доступ, как к отдельным ячейкам, так и к целым массивам памяти DSP модуля. Эта возможность позволяет программисту работать с модулем напрямую, непосредственно обращаясь к соответствующим ячейкам памяти. При этом для работы с этими функциями, в принципе, совсем не требуется загруженного в модуль драйвера *LBIOS*.

1.5.5.1. Чтение слова из памяти данных DSP

Формат: **BOOL** ***GET_DM_WORD****(WORD Address, SHORT * const Data)*
Назначение: Данная функция считывает значение слова, находящееся по адресу *Address* в памяти данных DSP модуля.
Передаваемые параметры: • *Address* – адрес ячейки в памяти данных DSP, значение которой необходимо считать; • *Data* – указатель на переменную, куда функция положит считанное $16^{\text{ти}}$ битное слово.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.5.2. Чтение слова из памяти программ DSP

Формат: **BOOL** ***GET_PM_WORD****(WORD Address, long * const Data)*
Назначение: Данная функция считывает значение слова, находящееся по адресу *Address* в памяти программ DSP модуля.
Передаваемые параметры: • *Address* – адрес ячейки в памяти программ DSP, значение которой необходимо считать; • *Data* – указатель на переменную, куда функция положит считанное $24^{\text{х}}$ битное слово.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.5.3. Запись слова в память данных DSP

Формат: **BOOL** ***PUT_DM_WORD****(WORD Address, SHORT Data)*
Назначение: Данная функция записывает значение *Data* в ячейку с адресом *Address* в памяти данных DSP модуля.
Передаваемые параметры: • *Address* – адрес ячейки в памяти данных DSP, куда необходимо записать значение *Data*; • *Data* – значение записываемого $16^{\text{ти}}$ битного слова.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.5.4. Запись слова в память программ DSP

Формат: **BOOL** ***PUT_PM_WORD****(WORD Address, long Data)*

Назначение:
Данная функция записывает значение *Data* в ячейку с адресом *Address* в памяти программ DSP модуля.

Передаваемые параметры:
- *Address* – адрес ячейки в памяти программ DSP, куда записывается значение *Data*;
- *Data* – значение записываемого 24^{x} битного слова.

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

1.5.5.5. Чтение массива слов из памяти данных DSP

Формат: **BOOL** ***GET_DM_ARRAY****(WORD BaseAddress, WORD NPoints, SHORT * const Buffer)*

Назначение:
Данная функция считывает массив слов длиной *NPoints* в буфер *Buffer*, начиная с адреса ячейки *BaseAddress* в памяти данных DSP модуля. Буфер *Buffer* надлежащей длины необходимо заранее определить.

Передаваемые параметры:
- *BaseAddress* – стартовый адрес в памяти данных DSP, начиная с которого производится чтение массива;
- *NPoints* – длина считываемого массива;
- *Buffer* – указатель на буфер, в который передаются считываемые значения.

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

1.5.5.6. Чтение массива слов из памяти программ DSP

Формат: **BOOL** ***GET_PM_ARRAY****(WORD BaseAddress, WORD NPoints, long * const Buffer)*

Назначение:
Данная функция считывает массив слов длиной *NPoints* в буфер *Buffer*, начиная с адреса ячейки *BaseAddress* в памяти программ DSP модуля. Буфер *Buffer* надлежащей длины необходимо заранее определить. При использовании этой функции следует помнить, что одно слово памяти программ DSP является 24^{x} битным.

Передаваемые параметры:
- *BaseAddress* – стартовый адрес в памяти программ DSP, начиная с которого производится чтение массива;
- *NPoints* – число считываемых 24^{x} битных слов;
- *Buffer* – указатель на буфер, в который передаются считываемые значения.

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

1.5.5.7. Запись массива слов в память данных DSP

Формат: **BOOL** ***PUT_DM_ARRAY****(WORD BaseAddress, WORD Npoints, SHORT * const Buffer)*

Назначение:

Данная функция записывает массив слов длиной *NPoints* из буфера *Buffer* в память данных DSP модуля, начиная с адреса *BaseAddress*.

Передаваемые параметры:

- *BaseAddress* – стартовый адрес в памяти данных DSP, начиная с которого производится запись массива;
- *NPoints* – длина записываемого массива;
- *Buffer* – указатель на буфер, из которого идет запись.

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

1.5.5.8. Запись массива слов в память программ DSP

Формат: **BOOL** ***PUT_PM_ARRAY****(WORD BaseAddress, WORD NPoints, long * const Buffer)*

Назначение:

Данная функция записывает массив слов длиной *NPoints* из буфера *Buffer* в память программ DSP модуля, начиная с адреса *BaseAddress*. При использовании этой функции следует помнить, что одно слово памяти программ DSP является 24^{x} битным.

Передаваемые параметры:

- *BaseAddress* – стартовый адрес в памяти программ DSP, начиная с которого производится запись массива;
- *NPoints* – число записываемых 24^{x} битных слов;
- *Buffer* – указатель на буфер, из которого идет запись.

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

1.5.5.9. Чтение переменной драйвера DSP

Формат: **BOOL** ***GET_LBIOS_WORD****(WORD Address, SHORT * const Data)*
Назначение: Данная функция осуществляет аккуратное считывание $16^{ти}$ битной переменной штатного *LBIOS*, расположенной по адресу *Address* в $24^{х}$ битной памяти *программ* DSP модуля (см. *§ 1.5.3.1. "Переменные драйвера DSP"*).
Передаваемые параметры: • *Address* – адрес ячейки переменной *LBIOS* в памяти *программ* DSP, значение которой необходимо считать; • *Data* – указатель на переменную, куда функция положит считанное $16^{ти}$ битное значение переменной штатного *LBIOS*.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.5.10. Запись переменной драйвера DSP

Формат: **BOOL** ***PUT_LBIOS_WORD****(WORD Address, SHORT Data)*
Назначение: Данная функция осуществляет аккуратную запись $16^{ти}$ битного значения *Data* в переменную штатного *LBIOS*, расположенную по адресу *Address* в $24^{х}$ битной памяти *программ* DSP модуля (см. *§ 1.5.3.1. "Переменные драйвера DSP"*).
Передаваемые параметры: • *Address* – адрес ячейки переменной *LBIOS* в памяти *программ* DSP, куда необходимо записать значение *Data*; • *Data* – значение записываемого $16^{ти}$ битного значения переменной штатного *LBIOS*.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.6. Функции для работы с АЦП

Интерфейсные функции штатной библиотеки `Lusbapi` позволяют реализовывать разнообразные алгоритмы работы модуля *E14-440* с АЦП *независимо* от состояния ЦАП. 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, длина и базовый адрес FIFO буфера 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. При одновременной работе АЦП и ЦАП необходимо помнить, что максимально возможная пропускная способность шины **USB** для данного модуля не более 500 кСлов/с. Примеры корректного применения интерфейсных функций для работы с АЦП можно найти в директории `\E14-440\Example\`.

1.5.6.1. Корректировка данных АЦП

Схемотехника и использованные компоненты обеспечивают линейность передаточной характеристики аналогового тракта АЦП модуля *E14-440*. Но каждый экземпляр модуля обладает своими, сугубо индивидуальными, значениями смещения нуля и неточности в передаче масштаба отсчёта АЦП. Это происходит вследствие того, что на модуле отсутствуют какие-либо подстроечные резисторы, что в целом позволяет улучшить шумовые характеристики модуля и увеличивает их надежность.

Пользователь может возложить всю работу по корректировке поступающих с модуля данных АЦП либо на своё приложение в PC, либо потребовать, чтобы сам модуль осуществлял всю эту процедуру на уровне драйвера DSP. В последнем случае это приводит к тому, что показания АЦП, поступающие из модуля в PC, имеют уже полностью откорректированный вид. И поэтому на уровне приложения в PC нет никакой необходимости в реализации всей этой нудной задачи по корректировке данных АЦП.

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 *E14-440*. При этом вся ответственность за формирование 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_E440*. Вся служебная информация совместно с корректировочными коэффициентами записывается в модуль на этапе при его наладке в **ЗАО "Л-Кард"**. Поля коэффициентов представляют собой массивы типа *double*. Для модуля *E14-440* в каждом из этих массивов имеют смысл только первые *ADC_CALIBR_COEFS_QUANTITY_E440* элементов. Массив *Adc.OffsetCalibration* содержит коэффициенты корректировки смещение нуля, а массив *Adc.ScaleCalibration* –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]*.

Поле *IsCorrectionEnabled* структуры *ADC_PARS_E440* определяет необходимость проведения автоматической корректировки данных АЦП на уровне драйвера DSP. Если же такая корректировка нужна, то в полях *AdcOffsetCoefs* и *AdcScaleCoefs* структуры *ADC_PARS_E440* требуется передать в DSP модуля все корректировочные коэффициенты.

В общем виде при использовании штатных коэффициентов корректировка 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_E440*), 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.

1.5.6.2. Запуск сбора данных АЦП

Формат: **BOOL** ***START_ADC****(void)*
Назначение: Данная функция запускает модуль *E14-440* 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.

1.5.6.3. Останов сбора данных АЦП

Формат: **BOOL** ***STOP_ADC****(void)*
Назначение: Данная функция останавливает в модуле *E14-440* механизм сбора данных с АЦП. Попутно эта функция как бы ‘*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.

1.5.6.4. Установка параметров работы АЦП

Формат:	**bool**	***FILL_ADC_PARS**(ADC_PARS_E440 *am)*	(версия 1.0)
	BOOL	***SET_ADC_PARS**(ADC_PARS_E440 * const AdcPars)*	(с версии 3.0)

Назначение:

Данная функция передает в *E14-440* 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_E440*. 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()*.

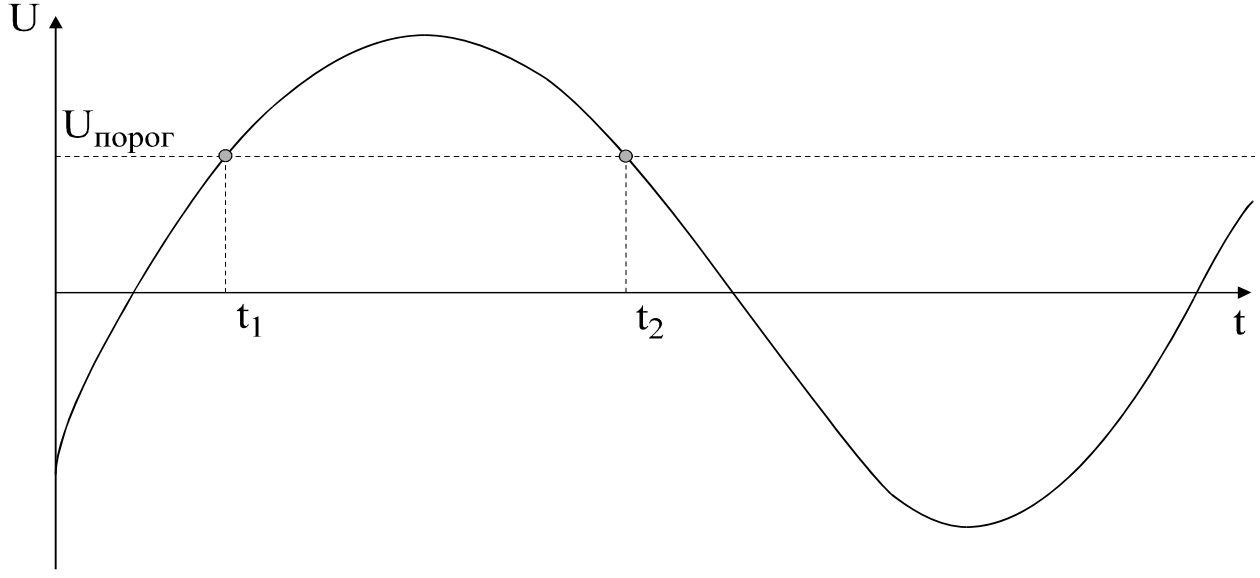
Описание структуры *ADC_PARS_E440* приведено ранее в *§ 1.5.2.2. "Структура ADC_PARS_E440"*, а назначение отдельных ее полей описано ниже.

- Поле AdcPars->IsCorrectionEnabled. Запись. Данное поле определяет возможность драйвера DSP осуществлять автоматическую корректировку получаемых с АЦП данных. Т.е. из модуля приложение получает уже откорректированные данные с АЦП, если установить поле равным *TRUE*. При использовании корректировки собственно сами корректировочные коэффициенты должны находиться в поле AdcPars->AdcOffsetCoefs и AdcPars->AdcScaleCoefs (см. ниже).
- 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*. Смысловая нагрузка значений этого поля представлена в таблице ниже:

Значение	Константа	Назначение
0	**NO_SYNC_E440**	***Отсутствие синхронизации ввода.*** Сбор данных с АЦП модуль начинает непосредственно после выполнения функции *START_ADC()*.
1	**TTL_START_SYNC_E440**	***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 **TRIG** аналогового разъёма *DRB-37M*. Длительность этого синхроимпульса должна быть не менее 50 нс. Только после этого модуль приступает к сбору данных.

2	**TTL_KADR_SYNC_E440**	***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 **TRIG** аналогового разъёма *DRB-37M*. Длительность этого синхроимпульса должна быть не менее 50 нс. После прихода синхроимпульса (см. выше) модуль собирает отсчеты только одного кадра отсчётов. Во время ввода кадра данных вся активность на линии **TRIG** игнорируется. Только после окончания сбора кадра ожидается приход следующего импульса синхронизации и т.д.
3	**ANALOG_SYNC_E440**	***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_E440**	Неправильная синхронизация ввода данных.

- Поля AdcPars->SynchroAdType, AdcPars->SynchroAdMode, AdcPars->SynchroAdChannel, *AdcPars->SynchroAdPorog*. Запись. Эти поля используются исключительно при аналоговой 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 $\mathbf{t_1}$. Тогда, когда уровень входного сигнала на логическом синхро канале 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 $\mathbf{t_2}$, 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 для аналоговой синхронизации;
- поле AdcPars->SynchroAdPorog – пороговое значение для аналоговой синхронизации (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 использоваться первые AdcPars->ChannelsQuantity элементов массива AdcPars->ControlTable. Предельным значением для данного поля является величина *MAX_CONTROL_TABLE_LENGTH_E440*. Если переданное в функцию значение превышает указанное предельное значение, то функция автоматически выполняет необходимую корректировку и по завершении оной в поле AdcPars->ChannelsQuantity будет находиться реально установленное количество активных *логических каналов*.
- 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. Эти поля при входе в данную функцию должны содержать требуемые временн***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** = $F_{clockout}/(2*(N+1))$, где $F_{clockout}$ – тактовая частота установленного на модуле DSP, равная 48000 кГц, а N – целое число. Поэтому данная функция просто вычисляет ближайшую к задаваемой дискретную величину **AdcRate**, передает её в модуль в виде целого $16^{ти}$ битного числа N, а также возвращает её значение в поле AdcPars->AdcRate. Все то же самое верно и для межкадровой задержки **InterKadrDelay**, с той лишь разницей, что она задается в единицах 1/**AdcRate**, причем уже откорректированной **AdcRate**. При этом минимальное значение **AdcRate** составляет 0.366 кГц, максимальное – 400 кГц. Например, если задать AdcPars->AdcRate = 0, то ***SET_ADC_PARS()*** установит и возвратит минимально возможную величину для данного поля, т.е. 0.366 кГц. Аналогично: если задать AdcPars->InterKadrDelay = 0, то данная функция установит и возвратит минимально возможную межкадровую задержку, т.е. 1/AdcPars->AdcRate.

- Поле AdcPars->KadrRate. Чтение. В данном поле возвращается частота кадра **KardRate** в *кГц*. Но это поле не имеет силу при использовании *покадровой синхронизации ввода*, которая может задаваться полем *AdcPars->InputMode*. **KardRate** 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 *§ 1.3.4. "Формат кадра отсчетов"*.
- Поле AdcPars->AdcFifoBaseAddress. Запись. Данное поле задаёт базовый адрес *FIFO* буфера АЦП в DSP модуля. Для данного модуля он всегда равен 0x0.
- Поле AdcPars->AdcFifoLength. Запись–Чтение. Данное поле задаёт длину *FIFO* буфера АЦП в DSP модуля. Для данного модуля эта величина может находиться в диапазоне от 0x40 (64) до 0x3000 (12288), а также быть обязательно кратной 0x40 (64). Если переданное в функцию значение не удовлетворяет указанным требованиям, то автоматически выполняется необходимая корректировка и по завершении оной в поле dm->AdcFifoLength будет находиться реально установленное значение длины *FIFO* буфера АЦП. **!!!ВАЖНО!!!** Настоятельно рекомендуется выбирать размер *FIFO* буфера АЦП так, чтобы полное время его заполнения было не менее 15÷30 *мс*. Передача данных из *FIFO* буфера АЦП в PC производится только порциями по AdcPars->AdcFifoLength/2 отсчётов (по мере их готовности).
- Поля AdcPars->AdcOffsetCoefs[] и AdcPars->AdcScaleCoefs[]. Запись. Данные поля являются массивами типа *double*. Они должны содержать коэффициенты необходимые при проведении модулем процедуры автоматической корректировки получаемых с АЦП данных. Разрешение применять модулем такую процедуру корректировки данных задаётся с помощью поля AdcPars->CorrectionEnabled. Подробнее смотри *§ 1.5.6.1. "Корректировка данных АЦП"*.

Передаваемые параметры:

- *AdcPars* – адрес структуры типа *ADC_PARS_E440* с требуемыми параметрами функционирования АЦП.

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

1.5.6.5. Получение текущих параметров работы АЦП

Формат:	**bool**	***GET_CUR_ADC_PARS***(*ADC_PARS_E440 *am)*	(версия 1.0)
	BOOL	***GET_ADC_PARS***(*ADC_PARS_E440 * const AdcPars)*	(с версии 3.0)

Назначение:

Данная функция считывает из модуля *E14-440* всю текущую информацию, которая используется при сборе данных с АЦП.

Передаваемые параметры:

- *AdcPars* – адрес структуры типа *ADC_PARS_E440* с полученными из модуля текущими параметрами функционирования АЦП.

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

1.5.6.6. Получение массива данных с АЦП

Формат: **bool** ***ReadData***(*SHORT *Buffer, DWORD *NumberOfWordsToRead, LPDWORD NumberOfBytesRead, LPOVERLAPPED Overlapped*) (версия 1.0)
BOOL ***ReadData***(*IO_REQUEST_LUSBAPI * const ReadRequest*) (с версии 3.0)

Назначение:

Данная функция предназначена для извлечения из модуля *E14-440* 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 *E14-440*. 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 по-кадрово: 1ый кадр, 2ой кадр и т.д. Причём положение отсчётов в кадрах будет совпадать с порядком размещения соответствующих *логических каналов* в управляющей таблице **ControlTable**.
NumberOfWordsToPass	*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.

NumberOfWordsPassed	*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, эта функция присутствует 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 DSP модуле *E14-440* организован *FIFO* буфер АЦП с регулируемым размером от 64 до 12288 отсчётов. Такой буфер необходим уверенного сбора данных на предельных частотах работы АЦП. Так при частотах сбора порядка 400 кГц переполнение полного буфера произойдёт только через 31 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 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, для порядку, может 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 `\E14-440\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 400 кГц после *START_ADC()*, но перед началом выполнения функции получения данных ***ReadData()*** система '*задумалась*' более чем на *31 мс* (что является вполне рядовым событием), 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 *E14-440*.

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

1.5.6.7. Ввод кадра отсчетов с АЦП

Формат:	**BOOL**	***ADC_KADR***(*SHORT * const Data*)	(с версии 2.0)

Назначение:

С версии 2.0 в библиотеке Lusbapi появилась дополнительная интерфейсная функция, которая позволяет осуществлять ввод кадра отсчетов 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()*. 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 `\E14-440\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.

1.5.6.8. 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. *§ 1.3.2.3. "Логический номер канала АЦП"*). Предварительно, с помощью штатной интерфейсной функции *SET_ADC_PARS()*, можно разрешить корректировку данных с заданного канала АЦП. Более подробно смотри исходные тексты примера из директории `\E14-440\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.

1.5.7. Функции для работы с ЦАП

Интерфейсные функции штатной библиотеки `Lusbapi` позволяют реализовывать разнообразные алгоритмы работы модуля *E14-440* с ЦАП *независимо* от состояния АЦП. Вообще-то модуль, с точки зрения работы с ЦАП, может находиться как бы в двух состояниях:

1. режим "*покоя*";
2. потоковая, т.е. *перманентная*, выдача данных на ЦАП.

Функция *START_DAC()* позволяет переводить модуль во второе из этих состояний, а *STOP_DAC()* — в первое. Перед запуском ЦАП предварительно необходимо передать в модуль требуемые параметры его функционирования: частота работы ЦАП, длина и базовый адрес *FIFO* буфера ЦАП. Эту операцию можно выполнить с помощью интерфейсной функции *SET_DAC_PARS()*. Непосредственно после выполнения функции *START_DAC()* модуль приступает к выводу данных на ЦАП. Причём данные для этого берутся из *FIFO* буфера ЦАП, который расположен в памяти данных DSP модуля (см. *§ 1.4.2. "Общая структура драйвера DSP"*). Поэтому очень важно, перед запуском ЦАП, проинициализировать этот буфер требуемыми данными, например, с помощью интерфейсной функции *PUT_DM_ARRAY()*. О формате данных передаваемых в *FIFO* буфер ЦАП смотри *§ 1.3.2.2. "Формат слова данных для ЦАП"*. Для '*подкачки*' же из приложения в *FIFO* буфер ЦАП *новых* данных уже в процессе потокового вывода, следует пользоваться функцией *WriteData()*. При одновременной работе АЦП и ЦАП необходимо помнить, что максимально возможная пропускная способность шины **USB** для данного модуля не более 500 кСлов/с. Примеры корректного применения интерфейсных функций для работы с ЦАП можно найти в директории `\E14-440\Example`.

1.5.7.1. Корректировка данных ЦАП

Схемотехника и использованные компоненты обеспечивают линейность передаточной характеристики ЦАП тракта модуля *E14-440*. Однако на *сегодняшний* день модуль не умеет производить 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 *штатные* калибровочные коэффициенты, хранящиеся в служебной информации модуля. Служебная информация совместно с нужными коэффициентами записывается в модуль на этапе при его наладке в **ЗАО "Л-Кард"**. Благодаря этому на модуле отсутствуют подстроечные резисторы, что улучшает шумовые характеристики модуля и увеличивает их надежность.

Штатные коэффициенты располагаются в полях *Dac.OffsetCalibration[]* и *Dac.ScaleCalibration[]* структуры служебной информации *MODULE_DESCRIPTION_E440*. Эти поля представляют собой массивы типа *double*. Для модуля *E14-440* в каждом из этих массивов используются только первые *DAC_CALIBR_COEFS_QUANTITY_E440* элементов. Массив *Dac.OffsetCalibration* содержит коэффициенты для корректировки смещение нуля, а массив *Dac.ScaleCalibration* – для корректировки масштаба.

В общем виде при использовании *штатных* коэффициентов корректировка данных ЦАП производится по следующей формуле:

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

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.

1.5.7.2. Запуск вывода данных на ЦАП

Формат:	**BOOL**	***START_DAC***(*void*)
Назначение: Данная функция запускает модуль *E14-440* на непрерывный *потоковый* вывод данных на ЦАП. Перед любым запуском вывода данных настоятельно рекомендуется выполнять функцию *STOP_DAC()*. После выполнения функции *START_DAC()* данные из *FIFO* буфера ЦАП начинают последовательно, с заданной частотой, выводится на ЦАП. Параметры работы ЦАП должны быть предварительно переданы в модуль с помощью интерфейсной функции *SET_DAC_PARS()*. Также предварительно перед запуском ЦАП следует проинициализировать *FIFO* буфер ЦАП необходимыми начальными значениями с помощью, например, интерфейсной функции *PUT_DM_ARRAY()*. Подробности о начальной инициализации *FIFO* буфера ЦАП и формате данных для ЦАП см. в прилагаемых к модулю примерах, а также см. *§ 1.3.2.2. "Формат слова данных для ЦАП"*). '*Подкачку*' же в модуль *новых* данных (уже в ходе работы ЦАП) для *FIFO* буфера ЦАП можно осуществлять с помощью интерфейсной функции *WriteData()*.		
Передаваемые параметры: нет		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

1.5.7.3. Останов вывода данных на ЦАП

Формат:	**BOOL**	***STOP_DAC***(*void*)
Назначение: Данная функция останавливает в модуле *E14-440* механизм вывода данных на ЦАП. Попутно эта функция '*приводит в чувство*' основную программу микроконтроллера модуля, 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()*.		
Передаваемые параметры: нет		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

1.5.7.4. Установка параметров работы ЦАП

Формат:	**bool**	***FILL_DAC_PARS***(*DAC_PARS_E440 *dm*)	(версия 1.0)
	BOOL	***SET_DAC_PARS***(*DAC_PARS_E440 * const DacPars)*	(с версии 3.0)

Назначение:

Данная функция передает в *E14-440* всю необходимую информацию, которая используется модулем для организации вывода данных на ЦАП. Всю нужную информацию описываемая интерфейсная функция извлекает из полей передаваемой структуры типа *DAC_PARS_E440*. 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 структуры *DAC_PARS_E440* приведено ранее в *§ 1.5.2.3. "Структура DAC_PARS_E440"*, а назначение отдельных ее полей описано ниже.

- Поле DacPars->DacRate. Запись–Чтение. При входе в функцию данное поле содержит требуемую частоту работы ЦАП **DacRate** в *кГц*. 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** = $F_{clockout}/(12*(N+1))$, где $F_{clockout}$ – тактовая частота DSP равная 48000 кГц, N – целое $16^{ти}$ битное число. Поэтому данная функция просто вычисляет ближайшую к задаваемой дискретную величину **DacRate**, передает ее в модуль (в виде числа N), а также возвращает её значение в поле DacPars->DacRate. Минимальное значение **DacRate** равно 0.061 кГц, максимальное – 125 кГц.
- Поле DacPars->DacFifoBaseAddress. Запись. Данное поле задаёт базовый адрес *FIFO* буфера ЦАП в DSP модуля. Для данного модуля он всегда равен 0x3000.
- Поле DacPars->DacFifoLength. Запись–Чтение. Данное поле задаёт длину *FIFO* буфера ЦАП в DSP модуля. Для данного модуля эта величина может находиться в диапазоне от 0x40 (64) до 0xFC0 (4032), а также быть обязательно кратной 0x40(64). Если переданное в функцию значение не удовлетворяет указанным требованиям, то автоматически выполняется необходимая корректировка и по завершении даной функции в поле DacPars->DacFifoBaseAddress будет находиться реально установленное значение длины *FIFO* буфера ЦАП. Передача (‘*подкачка*’) новых данных из PC в *FIFO* буфер ЦАП модуля производится порциями по DacPars->DacFifoBaseAddress/2 отсчетов (по мере их необходимости).

Передаваемые параметры:

- *DacPars* – адрес структуры типа *DAC_PARS_E440* с параметрами функционирования ЦАП.

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

1.5.7.5. Получение текущих параметров работы ЦАП

Формат:	**bool**	***GET_CUR_DAC_PARS***(*DAC_PARS_E440 *dm*)	(версия 1.0)
	BOOL	***GET_DAC_PARS***(*DAC_PARS_E440 * const DacPars*)	(с версии 3.0)

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_E440* с полученными из модуля текущими параметрами функционирования ЦАП.

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

1.5.7.6. Передача массива данных в ЦАП

Формат:	**bool**	***WriteData***(*WORD *Buffer, DWORD *NumberOfWordsToWrite, LPDWORD NumberOfBytesWritten, LPOVERLAPPED Overlapped*)	(версия 1.0)
	BOOL	***WriteData***(*IO_REQUEST_LUSBAPI * const WriteRequest*)	(с версии 3.0)

Назначение:

Данная функция предназначена для передачи очередной порции *новых* данных в *FIFO* буфер ЦАП модуля для их последующего 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 *E14-440*. Назначения полей этой структуры приведены в таблице ниже:

Название поля	Описание
Buffer	*Буфер данных.* Запись. **Buffer** предназначен для хранения очередной порции данных, которые необходимо будет переслать в *FIFO* буфер ЦАП модуля. 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 данных для ЦАП см *§ 1.3.2.2. "Формат слова данных для ЦАП"*.
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 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 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 *все* затребованные данные недопереданы, функция завершается и возвращает ошибку.

В DSP модуле *E14-440* организован программный *FIFO* буфер ЦАП с регулируемым размером от 64 до 4032 отсчётов. Такой буфер необходим уверенного вывода данных на предельных частотах работы ЦАП. Так при частотах вывода порядка 125 кГц переполнение полного буфера произойдёт только через 32 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 125 кГц после *START_DAC()*, но перед началом выполнения функции ***WriteData()*** система '*задумалась*' более чем на *32 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 *E14-440*.

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

1.5.7.7. Однократный вывод на ЦАП

Формат:	**bool**	***DAC_SAMPLE***(*WORD DacData, WORD DacChannel*)	(версия 1.0)
	BOOL	***DAC_SAMPLE***(*SHORT * const DacData, WORD DacChannel*)	(с версии 3.0)

Назначение:

Данная функция позволяет однократно устанавливать на задаваемом канале ЦАП *DacChannel* напряжение в соответствии со 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. *§ 1.3.2.2. "Формат слова данных для ЦАП"*.

Передаваемые параметры:

- *DacChannel* – требуемый номер канала ЦАП (0 или 1).
- *DacData* – устанавливаемое значение напряжения в кодах ЦАП (от –2048 до 2047).

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

1.5.8. Функции для работы с внешними цифровыми линиями

Все доступные цифровые линии модуля *E14-440* располагаются на внешнем разъёме *DRB-37F*. Дополнительную информацию об этом разъёме можно найти в *"E14-440. Руководство пользователя", § 3.3.2. "Внешний разъём для подключения цифровых сигналов"*.

Аппаратура модуля *E14-440* 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.

1.5.8.1. Разрешение выходных цифровых линий

Формат:	**BOOL**	***ENABLE_TTL_OUT***(*BOOL EnableTtlOut*)
Назначение: Данная интерфейсная функция позволяет осуществлять управление разрешением *всех* выходных линий внешнего цифрового разъёма *DRB-37F*. Т.о. существует возможность перевода их в третье, *высокоимпедансное*, состояние и обратно. Непосредственно после подачи на модуль питания выходные цифровые линии находятся в третьем состоянии.		
Передаваемые параметры: • *EnableTtlOut* – флажок, управляющий состоянием разрешения всех цифровых выходных линий.		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

1.5.8.2. Чтение внешних цифровых линий

Формат:	**BOOL**	***TTL_IN***(*WORD * const TtlIn*)
Назначение: Данная интерфейсная функция осуществляет однократное асинхронное чтение состояний всех $16^{ти}$ входных цифровых линий на внешнем разъёме *DRB-37F* модуля *E14-440*. Функция ***TTL_IN()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты ввода данных с цифровых линий порядка нескольких десятков *Гц*.		
Передаваемые параметры: • *TtlIn* – переменная, в которой возвращается побитовое состояние входных цифровых линий модуля.		
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.	

1.5.8.3. Вывод на внешние цифровые линии

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

Назначение:

Данная интерфейсная функция осуществляет установку *всех* $16^{ти}$ выходных цифровых линий на внешнем разъёме *DRB-37F* модуля *E14-440* в соответствии с битами передаваемого параметра *TtlOut*. Работа с цифровыми выходами предварительно должна быть разрешена с помощью интерфейсной функции *ENABLE_TTL_OUT()*. Функция ***TTL_OUT()*** выполняется достаточно *медленно* и, используя её, можно достичь частоты вывода данных на цифровые линии порядка нескольких десятков *Гц*.

Передаваемые параметры:

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

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

1.5.9. Функции для работы с пользовательским ППЗУ

На модуле *E14-440* установлено последовательное пользовательское ППЗУ емкостью 64 ячейки × 16 бит. Первые 32 ячейки данного ППЗУ используются под хранения служебной информации: название модуля, тип DSP, серийный номер, коэффициенты для корректировки отсчётов АЦП и ЦАП и т.д. А оставшиеся 32 ячейки предназначены для целей пользователя.

1.5.9.1. Разрешение/запрещение записи в ППЗУ

Формат: **BOOL** ***ENABLE_FLASH_WRITE***(*BOOL EnableFlashWrite*)
Назначение: Данная интерфейсная функция осуществляет управление режимом записи данных в пользовательское ППЗУ. Если запись разрешена, то с помощью штатной интерфейсной функции *WRITE_FLASH_WORD()* можно реализовать саму процедуру записи данных. Следует помнить, что после завершения всех требуемых операций записи информации в ППЗУ, настоятельно рекомендуется запретить с помощью данной интерфейсной функции режим записи данных в пользовательское ППЗУ.
Передаваемые параметры: • *EnableFlashWrite* – переменная может принимать следующие значения: ✓ если *TRUE*, то режим записи в ППЗУ разрешен, ✓ если *FALSE*, то режим записи в ППЗУ запрещен.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.9.2. Запись слова в ППЗУ

Формат: **BOOL** ***WRITE_FLASH_WORD***(*WORD FlashAddress, SHORT FlashWord*)
Назначение: Данная интерфейсная функция выполняет запись 16^{ти} битного слова *FlashWord* в ячейку пользовательского ППЗУ с номером *FlashAddress*. Перед началом записи в ППЗУ необходимо разрешить эту операцию с помощью интерфейсной функции *ENABLE_FLASH_WRITE()*. После окончания цикла записи всей требуемой информации настоятельно рекомендуется запретить режим записи в пользовательское ППЗУ с помощью той же функции *ENABLE_FLASH_WRITE()*. Т.к. в первых 32 ячейках ППЗУ находится служебная информация, которая используется библиотекой `Lusbapi` в процессе работы с модулем, то для пользователя доступны адреса ячеек только с 32 по 63.
Передаваемые параметры: • *FlashAddress* – номер ячейки ППЗУ, куда будет записано слово *FlashWord*; • *FlashWord* – слово, значение которого должно быть записано в ППЗУ.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.9.3. Чтение слова из ППЗУ

Формат: **BOOL** ***READ_FLASH_WORD***(*WORD FlashAddress, SHORT * const FlashWord*)
Назначение: Данная интерфейсная функция возвращает значения слова, находящегося в ячейке пользовательского ППЗУ с номером *FlashAddress*.
Передаваемые параметры: • *FlashAddress* – номер ячейки ППЗУ, откуда должно быть считано слово; • *FlashWord* – считанное значение.
Возвращаемое значение: *TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.10. Функции для работы со служебной информацией

Служебная информация содержит самые общие данные об используемом модуле *E14-440*: название модуля, его серийный номер и ревизию, корректировочные коэффициенты для АЦП и ЦАП, версии используемых прошивок MCU и DSP, тактовые частоты работы исполнительных устройств (MCU, DSP) и многое другое. Некоторые данные из этой служебной информации необходимы функциям штатной библиотеки `Lusbapi` для своей корректной работы.

1.5.10.1. Чтение служебной информации

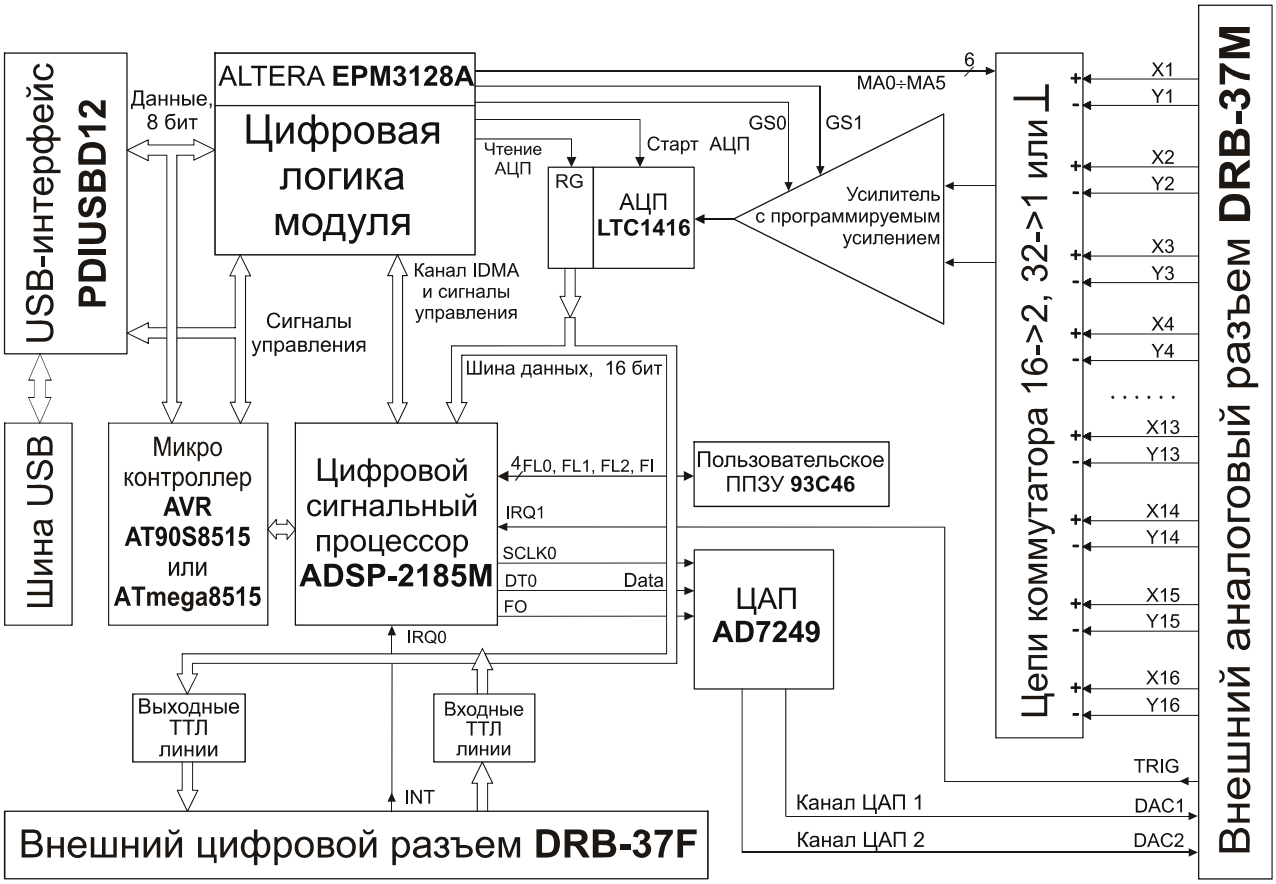
Формат:	**BOOL** ***GET_MODULE_DESCR***(*MODULE_DESCR_E440 *md*) (версия 1.0) **BOOL** ***GET_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E440 * const ModuleDescription*) (с версии 3.0)
Назначение: Данная интерфейсная функция осуществляет чтение всей служебной информации о модуле *E14-440* в структуру типа *MODULE_DESCRIPTION_E440* (см. *§ 1.5.2.1 "Структура MODULE_DESCRIPTION_E440"*). Эта информация требуется при работе с некоторыми интерфейсными функциями библиотеки `Lusbapi`. Поэтому данную функцию, во избежания непредсказуемого поведения приложений, следует обязательно вызывать непосредственно после загрузки в модуль драйвера DSP и проверки его работоспособности (см. *§ 1.4.1. "Общий подход к работе с интерфейсными функциями"*)	
Передаваемые параметры: • *ModuleDescription* – указатель на структуру типа *MODULE_DESCRIPTION_E440*, в которую заносится вся служебная информация модуля.	
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

1.5.10.2. Запись служебной информации

Формат:	**bool** ***SAVE_MODULE_DESCR***(*MODULE_DESCR_E440 *md*) (версия 1.0) **BOOL** ***SAVE_MODULE_DESCRIPTION***(*MODULE_DESCRIPTION_E440 * const ModuleDescription*) (с версии 3.0)
Назначение: Данная интерфейсная функция позволяет сохранять в модуле всю служебную информацию из структуры типа *MODULE_DESCRIPTION_E440*. **!!!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_E440*, из которой служебная информация переносится в модуль.	
Возвращаемое значение:	*TRUE* – функция успешно выполнена; *FALSE* – функция выполнена с ошибкой.

2. НИЗКОУРОВНЕВОЕ ОПИСАНИЕ МОДУЛЯ E14-440

2.1. Структурная схема модуля E14-440



Модуль E-440

Как видно из приведенной выше структурной схемы, функционально модуль *E14-440* как бы состоит из двух частей. Первая из них полностью отвечает за **USB**-интерфейс модуля с компьютером PC, т.е. с хост-компьютером. Она включает в себя микросхему **USB** интерфейса *PDIUSBD12* от фирмы *NXP Semiconductors* и микроконтроллер *AVR AT90S8515* или *ATmega8515* от фирмы *Atmel Corporation*. В ППЗУ микроконтроллера при наладке модуля в **ЗАО "Л-Кард"** зашивается специальный программный драйвер, всецело отвечающий за корректную работу *E14-440* с шиной **USB** и взаимодействие с цифровым сигнальным процессором. Другая часть находится под полным функциональным контролем цифрового сигнального процессора. Именно с помощью него можно достаточно гибко, чисто программном образом, эффективно управлять фактически всей периферией на модуле, т.е. обеспечивать надлежащее функционирование цепей коммутации и усиления входных аналоговых сигналов, управлять работой АЦП *LTC1416* от фирмы *Linear Technology Corporation* и ЦАП *AD7249* от фирмы *Analog Devices, Inc.*, контролировать состояния входных и выходных цифровых линий и т.д. Как уже сказано, на данном модуле установлен современный высокопроизводительный цифровой сигнальный процессор *ADSP-2185M* от фирмы от фирмы *Analog Devices, Inc*, работающий на тактовой частоте $\mathbf{F}_{clockout}$ = 48000.0 кГц. Внутренняя архитектура данного сигнального процессора специальным образом оптимизирована для реализации таких алгоритмов обработки информации, как цифровая фильтрация, спектральный анализ и т.д. Сам процессор имеет достаточно большую внутреннюю оперативную память программ (16 КСлов) и внутреннюю память данных (16 КСлов). Наличие на модуле *E14-440* такого сигнального процессора обеспечивает Вам, при необходимости и определенном усердии, возможность самостоятельного применения чрезвычайно гибких специализированных методов (алгоритмов) управления всей доступной периферией модуля. Также можно пе-

реносить часть достаточно сложных операций по обработке информации на сам модуль. Например, выполнять Быстрое Преобразование Фурье.

На модуле *E14-440* также устанавливается последовательное электрически стираемое перепрограммируемое запоминающее устройство, т.е. так называемое ППЗУ или Serial EEPROM. В качестве основы такого ППЗУ была выбрана микросхемы типа *93C46* с организацией памяти 64 Слов по 16 бит. Подробнее о формате данных в ППЗУ см. *§ 1.3.3 "Формат пользовательского ППЗУ"*. В ППЗУ при наладке на **ЗАО "Л-Кард"** прописывается вся служебная информация, корректировочные коэффициенты для АЦП и ЦАП, а также предусматривается область для пользовательских нужд.

Применение на модуле *E14-440* микросхемы программируемой логической матрицы (ПЛМ) позволяет разместить в ней практически всю логическую часть устройства, обеспечивая надёжное функциональное взаимодействие основных составных сегментов модуля. В качестве ПЛМ была выбрана микросхема типа *EPM3128A* от фирмы *Altera Corporation*. Кроме всего прочего, использование ПЛМ существенно упрощает процедуру разработки и монтажа изделия, а также максимально способствует достижению минимальных габаритных размеров модуля.

2.2. Организация USB интерфейса

2.2.1. Общие сведения о USB

USB (**U*niversal* Serial Bus**) — универсальная последовательная шина, которая является на сегодняшний день широко распространенным промышленным стандартом расширения архитектуры персонального компьютера. В своё время разработка данного стандарта была инициирована весьма авторитетными компьютерными фирмами – Intel, DEC, IBM, NorthenTelecom и Compaq. Версия первого утвержденного варианта стандарта **USB** (спецификация 1.0) появилась по компьютерным мерка довольно давно – 15 января 1996 года. Следующая *спецификация 1.1* была принята 23 сентября 1998 года. И, наконец, действующая на настоящий момент *спецификация 2.0* – 27 апреля 2000 года. Собственно, сами эти спецификации можно совершенно свободно скачать с сайта *www.usb.org*. Таким образом, согласно спецификации шина **USB** может с легкостью использоваться для подключения к компьютеру самых различных устройств, а именно: клавиатур, мышей, джойстиков, модемов, принтеров, сканеров, приводов CD-ROM, аудиоустройств (например, микрофонов и колонок), цифровых фото- и видеокамер, а также множества другой мультимедийной периферии. Т.о. мы видим, что для взаимодействия с компьютером данный стандарт может с успехом применяться для самого широкого и разнообразного спектра периферийных устройств.

Все преимущества применения именно шины **USB** можно описывать довольно долго. Очень много интересного можно найти в Интернете. Немного осветим всего лишь один важный аспект применения рассматриваемой шины **USB**. При подключении не **USB**-устройств к компьютеру (например, в процессе его модернизации) пользователям, как правило, все ещё приходится 'сражаться' с малопонятными настройками и элементами архитектуры PC, включая запросы прерываний (IRQ), каналы прямого доступа к памяти DMA и адреса портов ввода-вывода. Да и с подсоединением кабелей к разъемам, которых на задней стенке системного блока полным-полно, у многих возникают затруднения. Все эти и другие сложности с подключением к персональному компьютеру необходимых пользователю дополнительных устройств не радуют и самих производителей, поскольку противоречат самым современным требованиям к простоте использования и инсталляции внешней периферии. В том числе, исходя и из этих требований, группа лидирующих в компьютерной индустрии компаний создала и активно продвигает на рынок стандарт, получивший название универсальной последовательной шины, т.е. **USB**. Также одно из главных преимуществ шины **USB** состоит как раз в поддержке так называемого "горячего" подключения и отключения к PC периферийных устройств. Это означает, что пользователи могут подсоединить к компьютеру новое устройство и начать с ним работать, не выключая и не перезагружая систему.

Интерфейс **USB** позволяет осуществлять обмен информацией между хост-компьютером и множеством различных одновременно доступных периферийных устройств, обеспечивая при этом возможность работы на трёх различных скоростях :

- низкая скорость (*Low Speed* – ***LS***) – 1,5 Мбит/с (спецификация *USB 1.1*);
- полная скорость (*Full Speed* – ***FS***) – 12 Мбит/с (спецификация *USB 1.1*);
- высокая скорость (*High Speed* – ***HS***) – 480 Мбит/с (спецификация *USB 2.0*).

Фактически интерфейс соединяет между собой ***хост-контроллер*** **USB** и периферийные устройства. Хост-контроллер **USB** находится внутри персонального компьютера (в сущности, он является программно-аппаратной подсистемой персонального компьютера) и полностью конролирует работу всего интерфейса, т.е. все передачи данных по интерфейсу инициируются именно хост-контроллером. Для того чтобы к одному порту **USB** можно было подключать более одного устройства, применяются ***хабы*** (*hub* – устройство, обеспечивающее подключение к интерфейсу других устройств). ***Корневой хаб*** (*root hub*) находится внутри компьютера и подключен непосредственно к хосту. В интерфейсе **USB** используется специальный термин ***"функция"*** – это логически законченное устройство, выполняющее какую-либо специфическую функцию. В нашем случае это и есть модуль *E14-440*.

Всего в интерфейсе **USB** может быть использовано четыре типа пересылок информации, а именно:

- ***управляющая пересылка*** (*control transfer*). Используется для конфигурации устройства, а также для других специфических для конкретного устройства целей.
- ***потоковая пересылка*** (*bulk transfer*). Используется для передачи относительно большого объема информации. Характеризуется гарантированной безошибочной передачей данных между хост-контроллером и устройством посредством обнаружения ошибок в процессе передачи и повторного запроса информации.
- ***пересылка с прерыванием*** (*interrupt transfer*). Используется для передачи относительно небольшого объема информации, для которого особенно важна своевременная его пересылка. Имеет ограниченную длительность и повышенный приоритет относительно других типов пересылок.
- ***изохронная пересылка*** (*isochronous transfer*), также называется потоковой пересылкой *реального* времени. Информация, передаваемая в такой пересылке, требует реального масштаба времени при ее создании, пересылке и приеме.

В модуле *E14-440* из всего этого набора используются только ***управляющие*** **и** ***потоковые*** пересылки.

Спецификация **USB** определяет набор стандартных операций, которые должны поддерживать абсолютно все **USB**-устройства. Эти стандартные операции гарантируют некоторую согласованность в базовом поведении устройств, когда они подключаются к шине. Кроме того, разработчики **USB**-периферии при желании могут определять дополнительные операции **USB**-устройств, чтобы адекватно реализовывать необходимые алгоритмы их функционирования.

2.2.2. Интерфейс AVR с USB шиной

В связи с тем, что в интерфейсе **USB** реализован довольно сложный протокол обмена информацией, в устройстве сопряжения с интерфейсом **USB** необходимо применять микропроцессорный блок, обеспечивающий полную поддержку протокола. В качестве такового на модуле *E14-440* используется согласованная связка микросхем **USB** интерфейса *PDIUSBD12* от фирмы *NXP Semiconductors* и микроконтроллера *AVR AT90S8515* или *ATmega8515* от фирмы *Atmel Corporation*. При наладке модуля *E14-440* в **ЗАО "Л-Кард"** в ППЗУ микроконтроллера AVR зашивается специально разработанный драйвер, одна из основных задач которого — обеспечить корректный интерфейс модуля с хост-компьютером. В данном описании мы не будем глубоко вдаваться в технические подробности реализации **USB** интерфейса, а только в общем виде обрисуем заложенные в драйвер особенности, которые совершенно необходимы для надлежащего программирования модуля при создании своего приложения в PC.

Итак, помимо набора *стандартных запросов*, которые поступают с хост-компьютера и должны пониматься всеми без исключения **USB**-устройствами, в модуле *E14-440* организован целый ряд специализированных запросов '*Vendor Request*' для целей реализации требуемых алгоритмов функционирования модуля. Также как и для всех стандартных запросов, для посылки запроса такого рода используется ***управляющая*** пересылка. Каждый специализированный запрос имеет свой уникальный номер и по мере поступления в AVR обрабатывается соответствующим образом, т.е. AVR выполняет действия, однозначно предопределенные номером запроса. Например, в ответ на поступление V_RESET_MODULE запроса, микроконтроллер AVR просто осуществляет сброс модуля. С программной точки зрения посылка из хост-компьютера требуемого запроса типа '*Vendor Request*' в модуль *E14-440* осуществляется с помощью обычной *Windows API* функции ***DeviceIoControl()***. Подробнее об этом смотри документацию на эту функцию и исходные тексты библиотеки `Lusbapi` в директории `\DLL\Source\Lusbapi`. Далее мы попробуем чуть-чуть подробнее затронуть описание форматов функции ***DeviceIoControl()*** и запросов типа '*Vendor Request*' применительно к конкретной реализации модуля *E14-440*.

2.2.2.1. Функция DeviceIoControl()

Стандартная *Windows API* функция ***DeviceIoControl()*** является одной из самых ключевых функций в деле организации взаимодействия приложения в PC с модулем *E14-440*. Вот как эта функция объявлена в системе *Windows* (см. документацию на эту функцию):

```
BOOL DeviceIoControl
(
    HANDLE hDevice,                 // handle to device of interest
    DWORD dwIoControlCode,          // control code of operation to perform
    LPVOID lpInBuffer,              // pointer to buffer to supply input data
    DWORD nInBufferSize,            // size of input buffer (in bytes)
    LPVOID lpOutBuffer,             // pointer to buffer to receive output data
    DWORD nOutBufferSize,           // size of output buffer (in bytes)
    LPDWORD lpBytesReturned,        // pointer to variable to receive output byte count
    LPOVERLAPPED lpOverlapped       // pointer to overlapped structure for asynchronous
                                    // operation
);
```

Из описания данной функции следует, что её следует применять для передачи управляющих кодов `dwIoControlCode` непосредственно в драйвер с дескриптором (идентификатором) `hDevice`. Управляющие коды `dwIoControlCode` определяют те действия драйвера, которые он должен выполнить. Для модуля *E14-440* в драйвере предусмотрены три управляющих кода (см. файл `DLL\Source\Bulkioct.h`), мнемоники которых приведены ниже:

1. DIOC_SEND_COMMAND. Данный код предписывает драйверу выполнить соответствующую ***управляющую*** пересылку, т.е. посылку в модуль запроса типа 'V*endor Request*' с требуемым номером.

2. DIOC_RESET_PIPE1. Данный код предписывает драйверу выполнить сброс канала, обеспечивающего ***потоковые пересылки*** для записи данных в модуль.
3. DIOC_RESET_PIPE3. Данный код предписывает драйверу выполнить сброс канала, обеспечивающего ***потоковые пересылки*** для чтения данных из модуля.

При этом параметры `lpInBuffer`, `nInBufferSize`, `lpOutBuffer` и `nOutBufferSize`, задающие параметры требуемой ***управляющей*** пересылки, имеют смысл только при управляющем коде равном *DIOC_SEND_COMMAND*. При остальных же кодах они должны быть равны нулю. Собственно сам запрос '*Vendor Request*' может состоять из двух различных фаз:

1. сначала организуется пересылка типа *Setup packet*,
2. а затем, если в этом есть необходимость, организуются пересылки типа *Data stage*.

Весьма подробное изложение формата ***управляющих*** пересылок можно найти в официальной документации на шину **USB**, а именно в *§ 9.3 "USB Device Requests" спецификации 1.1*. Т.о. фаза *Setup packet* всегда присутствует в запросе, в то время как *Data stage* фаза вполне законно может и отсутствовать, в зависимости от требований. Стадия *Setup packet* состоит из пересылки по шине **USB** строго структуризированных восьми байтов информации, в которых передаются такие важные параметры как номер запроса, тип запроса (в рассматриваемом случае это всегда *Vendor Request*), направление передачи данных в фазе *Data stage*, количество байт необходимых переслать в фазе *Data stage* (если этот параметр равен нулю, то считается, что фаза *Data stage* полностью отсутствует, т.е. нет потребности в дополнительной передачи данных) и т.д. Массив `lpInBuffer` состоящий из четырех элементов типа *WORD* (параметр `nInBufferSize`) как раз предназначен для организации посылки *Setup packet* с требуемыми параметрами. Массив `lpOutBuffer` длиной `nOutBufferSize` байт, если это необходимо, используется для организации передачи данных в фазе *Data stage*. Подробности этих параметров будут рассмотрены ниже при описании доступных штатной DLL библиотеке номеров запросов типа '*Vendor Request*'.

Переменная `lpBytesReturned` задает обычный счетчик полученных байтов, если оные передаются из драйвера.

Так как функция ***DeviceIoControl()*** всегда выполняется в асинхронном режиме, в переменной `lpOverlapped` необходимо передавать указатель на стандартную *WinAPI* структуру типа *OVERLAPPED*, в которой должно быть проинициализировано событие для асинхронного запроса.

2.2.2.2. Запрос V_RESET_MODULE

Запрос типа '*Vendor Request*' с номером V_RESET_MODULE (см. файл `DLL\Source\E440.h`) предназначен для выполнения сброса DSP модуля. Т.е. при поступлении запроса с данным номером микроконтроллер AVR осуществляет аппаратный сброс DSP. С программной точки зрения пересылка указанного запроса выполняется очень просто и выглядит это примерно так:

```
WORD InBuf[4];      // буфер под параметры Setup packet
DWORD cbRet=0;      // байтовый счетчик полученных байтов
OVERLAPPED Ov;      // WinAPI структура
InBuf[0] = 0;       // направление передачи данных в фазе Data stage
                    // в данном запросе не используется, т.к. этой стадии не будет
InBuf[1] = V_RESET_MODULE;        // номер запроса
InBuf[2] = 0x0;     // дополнительный параметр (в данном запросе не используется)
InBuf[3] = 0x0;     // дополнительный параметр (в данном запросе не используется)
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                NULL,       // в данном запросе фазы Data stage не будет
                0,          // в данном запросе фазы Data stage не будет
                &cbRet,
                &Ov);
```

2.2.2.3. Запрос V_PUT_ARRAY

Запрос типа 'V*endor Request*' с номером V_PUT_ARRAY (см. файл DLL\Source\E440.h) предназначен для выполнения записи массива слов в память цифрового сигнального процессора, установленного на модуле. Т.е. при поступлении запроса с данным номером микроконтроллер AVR осуществляет процедуру записи поступающей информации по каналу *IDMA DSP* в память сигнального процессора. Немного подробнее о IDMA DSP смотри *§ 2.4.1 "Общие сведения"*. С программной точки зрения пересылка указанного запроса выполняется достаточно просто и выглядит следующим образом:

```
WORD InBuf[4];          // буфер под параметры Setup packet
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура

InBuf[0] = 0;           // будет передача данных в модуль в фазе Data stage
InBuf[1] = V_PUT_ARRAY;       // номер запроса
InBuf[2] = BaseAddress;       // базовый адрес в памяти DSP, начиная с которого
                              // будет осуществляться запись массива данных
InBuf[3] = 0x0;         // дополнительный параметр (в данном запросе не используется)
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                Data,       // массив с передаваемыми в фазе Data stage данными
                NBytes,     // в фазе Data stage передается NBytes байтов данных
                &cbRet,
                &Ov);
```

2.2.2.4. Запрос V_GET_ARRAY

Запрос типа 'V*endor Request*' с номером V_GET_ARRAY (см. файл DLL\Source\E440.h) предназначен для выполнения чтения массива слов из памяти цифрового сигнального процессора, установленного на модуле. Т.е. при поступлении запроса с данным номером микроконтроллер AVR осуществляет процедуру чтения информации по каналу *IDMA DSP* из памяти сигнального процессора. Немного подробнее о IDMA DSP смотри *§ 2.4.1 "Общие сведения"*. С программной точки зрения пересылка указанного запроса выполняется достаточно просто и выглядит следующим образом:

```
WORD InBuf[4];          // буфер под параметры Setup packet
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура

InBuf[0] = 1;           // будет приём данных из модуля в фазе Data stage
InBuf[1] = V_GET_ARRAY;       // номер запроса
InBuf[2] = BaseAddress;       // базовый адрес в памяти DSP, начиная с которого
                              // будет осуществляться чтение массива данных
InBuf[3] = 0x0;         // дополнительный параметр (в данном запросе не используется)
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                Data,       // массив для получаемых в фазе Data stage данных
                NBytes,     // в фазе Data stage передается NBytes байтов данных
                &cbRet,
```

```
                    &Ov);
```

2.2.2.5. Запрос V_START_ADC

Запрос типа 'V*endor Request*' с номером V_START_ADC (см. файл DLL\Source\E440.h) предназначен для надлежащей инициализации внутренних переменных и структур драйвера микроконтроллера. Это делается с целью подготовить программу AVR для дальнейшей работы по каналу *IDMA DSP* с *FIFO буфером АЦП*. Немного подробнее о IDMA DSP см. *§ 2.4.1 "Общие сведения"*. С программной точки зрения пересылка указанного запроса выполняется достаточно просто и выглядит примерно так:

```
WORD InBuf[4];          // буфер под параметры Setup packet
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура

InBuf[0] = 0;           // направление передачи данных в фазе Data stage
                        // в данном запросе не используется, т.к. этой стадии не будет
InBuf[1] = V_START_ADC;             // номер запроса
InBuf[2] = 0x0 | DSP_DM;            // базовый адрес FIFO буфера АЦП
InBuf[3] = AdcFifoLength/2.0;       // половина длины FIFO буфера АЦП
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                NULL,               // в данном запросе фазы Data stage не будет
                0,                  // в данном запросе фазы Data stage не будет
                &cbRet,
                &Ov);
```

Разумеется, что сама по себе посылка только данного запроса не обеспечивает собственно запуска АЦП. Для этого необходимо выполнить еще два дополнительных шага:

1. Осуществить посылку с помощью функции ***DeviceIoControl()*** в драйвер устройства ещё одного управляющего кода *DIOC_RESET_PIPE3*:
   ```
   DeviceIoControl(hDevice,
                   DIOC_RESET_PIPE3/* reset Read Pipe */,
                   NULL, NULL, NULL, NULL, &cbRet, NULL);
   ```
2. Послать в DSP модуля команду запуска АЦП, используя для этого штатную интерфейсную функцию *SEND_COMMAND()*:
   ```
   SEND_COMMAND(C_START_ADC);
   ```

После этих нехитрых манипуляций можно быть уверенным, что запуск АЦП модуля успешно осуществлен. Впоследствии, если в этом есть необходимость, можно организовать передачу из модуля, а точнее из *FIFO буфера АЦП*, в хост-компьютер уже собранной с АЦП аналоговой информации. В модуле *E14-440* за процедуру подобного рода отвечает *потоковая пересылка* данных из устройства в PC (подробнее см. *§ 2.2.2.11 "Потоковые пересылки"*).

2.2.2.6. Запрос V_START_DAC

Запрос типа 'V*endor Request*' с номером V_START_DAC (см. файл DLL\Source\E440.h) предназначен для надлежащей инициализации внутренних переменных и структур драйвера микроконтроллера. Это делается с целью подготовить программу AVR для дальнейшей работы по каналу *IDMA DSP* с *FIFO буфером ЦАП*. Немного подробнее о IDMA DSP смотри *§ 2.4.1 "Общие сведения"*. С программной точки зрения пересылка указанного запроса выполняется достаточно просто и выглядит примерно так:

```
WORD InBuf[4];          // буфер под параметры Setup packet
```

```
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура
InBuf[0] = 0;           // направление передачи данных в фазе Data stage
                        // в данном запросе не используется, т.к. этой стадии не будет
InBuf[1] = V_START_DAC;             // номер запроса
InBuf[2] = 0x3000 | DSP_DM;         // базовый адрес FIFO буфера ЦАП
InBuf[3] = DacFifoLength/2.0;       // половина длины FIFO буфера ЦАП
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                NULL,               // в данном запросе фазы Data stage не будет
                0,                  // в данном запросе фазы Data stage не будет
                &cbRet,
                &Ov);
```

Разумеется, что сама по себе посылка только данного запроса не обеспечивает собственно запуска ЦАП. Для этого необходимо выполнить еще два дополнительных шага:

1. Осуществить посылку с помощью функции ***DeviceIoControl()*** в драйвер устройства ещё одного управляющего кода *DIOC_RESET_PIPE1*:
   ```
   DeviceIoControl(hDevice,
                   DIOC_RESET_PIPE1/* reset Write Pipe */,
                   NULL, NULL, NULL, NULL, &cbRet, NULL);
   ```
2. Послать в DSP модуля команду запуска ЦАП, используя для этого штатную интерфейсную функцию *SEND_COMMAND()*:
   ```
   SEND_COMMAND(C_START_DAC);
   ```

После этих нехитрых манипуляций можно быть уверенным, что запуск ЦАП модуля успешно осуществлен. Впоследствии, если в этом есть необходимость, можно организовать передачу в модуль, а точнее в *FIFO буфер ЦАП*, дополнительных данных с целью их дальнейшего вывода на собственно сам ЦАП. В модуле *E14-440* за процедуру подобного рода отвечает *потоковая пересылка* данных в устройство (подробнее см. *§ 2.2.2.11 "Потоковые пересылки"*).

2.2.2.7. Запрос V_COMMAND_IRQ

Запрос типа 'V*endor Request*' с номером V_COMMAND_IRQ (см. файл `DLL\Source\E440.h`) предназначен для инициирования так называемого *'командного' прерывания IRQE* в цифровом сигнальном процессоре, установленном на модуле. Т.е. при поступлении запроса с данным номером микроконтроллер AVR осуществляет аппаратное генерирование прерывания *IRQE* в DSP. С программной точки зрения пересылка указанного запроса выполняется очень просто и выглядит следующим образом:

```
WORD InBuf[4];          // буфер под параметры Setup packet
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура
InBuf[0] = 0;           // направление передачи данных в фазе Data stage
                        // в данном запросе не используется, т.к. этой стадии не будет
InBuf[1] = V_COMMAND_IRQ;           // номер запроса
InBuf[2] = 0x0;     // дополнительный параметр (в данном запросе не используется)
InBuf[3] = 0x0;     // дополнительный параметр (в данном запросе не используется)
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                NULL,       // в данном запросе фазы Data stage не будет
```

```
                0,              // в данном запросе фазы Data stage не будет
                &cbRet, &Ov);
```

2.2.2.8. Запрос V_GET_MODULE_NAME

Запрос типа 'V*endor Request*' с номером V_GET_MODULE_NAME (см. файл DLL\Source\E440.h) предназначен для получения названия опрашиваемого устройства. Т.е. при поступлении запроса с данным номером микроконтроллер AVR модуля осуществляет в фазе *Data stage* передачу по шине **USB** семибайтового массива данных, содержащего строку 'E440'. С программной точки зрения пересылка указанного запроса выполняется очень просто и выглядит следующим образом:

```
WORD InBuf[4];          // буфер под параметры Setup packet
DWORD cbRet=0;          // байтовый счетчик полученных байтов
OVERLAPPED Ov;          // WinAPI структура

InBuf[0] = 1;           // будет приём данных из модуля в фазе Data stage
InBuf[1] = V_GET_MODULE_NAME;           // номер запроса
InBuf[2] = 0x0;         // дополнительный параметр (в данном запросе не используется)
InBuf[3] = 0x0;         // дополнительный параметр (в данном запросе не используется)
DeviceIoControl(hDevice,
                DIOC_SEND_COMMAND,
                &InBuf,
                sizeof(InBuf),
                ModuleName,     // массив для получаемых в фазе Data stage данных
                7,              // в фазе Data stage передается 7 байтов данных
                &cbRet,
                &Ov);
```

2.2.2.9. Потоковые пересылки

Пересылки такого рода на модуле *E14-440* используются исключительно для целей передачи массивов данных из *FIFO буфера АЦП* в хост-компьютер, а также из хост-компьютера в *FIFO буфер ЦАП*.

В общем виде принципы работы здесь достаточно проста. Допустим, пользователь осуществил запуск АЦП. Драйвер сигнального процессора начинает располагать поступающие отсчёты с АЦП в так называемом *FIFO буфере АЦП*. По мере накопления определенной порции аналоговых данных DSP через посредство соответствующего прерывания (см. *§ 2.3. "Интерфейс AVR с DSP"*), даёт знать микроконтроллеру AVR, что пора бы что-то делать с накопившейся информацией. В этой ситуации AVR проверяет, поступил ли запрос из хост-компьютера на ***потоковую*** передачу аналоговой информации в PC. Если да – AVR осуществляет такую операцию пересылки данных в PC из соответствующей половинки *FIFO буфера АЦП*, если нет – AVR терпеливо ждет поступления требуемого запроса. Пользователь из своего приложения в PC может легко организовать так необходимый ему для нормального сбора данных с АЦП запрос в модуль на ***потоковую*** пересылку. Но только предварительно он должен обеспечить запуск АЦП с помощью интерфейсной функции *START_ADC()*. Для этих целей и служит штатная интерфейсная функция *ReadData()*, которая является по сути своего рода слепком с *Windows API* функции ***RadFile()***. Т.е. данная функция обеспечивает формирование запроса на ***потоковую*** пересылку, передачу его в модуль и приём из модуля требуемого количества данных с аналоговых каналов.

Всё то же самое верно и для случая работы ЦАП. За тем исключением, что за формирование ***потоковой*** пересылки на передачу данных в модуль для *FIFO буфер ЦАП* предназначена штатная интерфейсная функция *WriteData()*. По своей сути эта функция является своего рода слепком с *Windows API* функции ***WriteFile()***.

2.3. Интерфейс AVR с DSP

Помимо ответственности за организацию и поддержание в надлежащем виде **USB**-интерфейса модуля *E14-440* с хост-компьютером, на микроконтроллер AVR ложится довольно непростая задача по корректному взаимодействию с цифровым сигнальным процессором, расположенным на модуле.

Драйвер микроконтроллера AVR написан таким образом, что разрешает пользователю из приложения в PC путём посылки в модуль соответствующих запросов **USB** (см. *§ 2.2. "Организация USB интерфейса"*) задавать функциональное состояние модуля *E14-440*. При этом аппаратно модуль реализован так, что вся периферия (АЦП, ЦАП, ТТЛ линии и .т.д.) находится исключительно под управлением цифрового сигнального процессора. Поэтому от AVR настоятельно требуется обладать возможностью оперативного вмешательства в любой момент времени в текущую работу драйвера DSP без остановки в его функционировании (кроме сброса DSP). С этой целью весь интерфейс микроконтроллера AVR с DSP реализован двояким образом:

1. Набор выделенных цифровых линий, которые в штатном программном обеспечении (как в драйвере AVR, так и в *LBIOS'*е) используются следующим образом:
   - ✓ По поступлении из хост-компьютера запроса *V_RESET_MODULE* AVR выполняет полный аппаратный сброс DSP.
   - ✓ По поступлении из хост-компьютера запроса *V_COMMAND_IRQ* AVR инициирует в DSP так называемое *'командное' прерывания IRQE*, которое должно надлежащим образом отрабатываться соответствующим обработчиком прерывания в *LBIOS'*е.
   - ✓ Две цифровые линии заведены с флагов *PF1* и *PF3* DSP (см. *§ 2.4.5. "Конфигурирование флагов DSP"*) на внешние прерывания AVR, а именно на входные линии *INT0* и, в зависимости от ревизии модуля, *ANA_COMP* или *INT2* (см. описание *AVR AT90S8515* или *ATmega8515*). Штатное программное обеспечение DSP использует флаги *PF1* и *PF3* с целью извещения AVR о настоятельной необходимости обеспечить передачу требуемых данных либо в хост-компьютер (уже собранные данные из *FIFO буфера АЦП*), либо из хост-компьютера (новые данные для *FIFO буфера ЦАП*).
2. Возможность доступа микроконтроллера AVR к любой ячейке памяти DSP обеспечивается через посредство такой полезной архитектурной особенности сигнального процессора, как наличие в микросхеме DSP канала *IDMA*. Чуть подробнее о канале IDMA DSP смотри *§ 2.4.1 "Общие сведения"*. Именно эта отличительная черта архитектуры сигнального процессора даёт возможность пользователю из своего приложения при помощи организации посылок запросов типа *V_PUT_ARRAY* и *V_GET_ARRAY* считать или изменить по своему усмотрению содержимое любой области памяти DSP. Также именно благодаря этой архитектурной особенности сигнального процессора микроконтроллер AVR в произвольные моменты времени имеет возможность обращаться к содержимому любой области как в *FIFO буфере АЦП*, так и в *FIFO буфере ЦАП*, обеспечивая, таким образом, необходимые *потоковые пересылки* информации в/из хост-компьютера.

2.4. Интерфейс DSP с периферией модуля

В данном разделе хотелось бы достаточно подробно осветить различные аспекты взаимодействия установленного на модуле *E14-440* цифрового сигнального процессора с периферией, а именно АЦП, ЦАП, цифровые линии, ППЗУ и т.д.

2.4.1. Общие сведения

Основными функциональными обязанностями цифрового сигнального процессора *ADSP-2185M*, установленного на модуле *E14-440*, являются управление и полный контроль над всей периферией устройства, а также, при необходимости, первичная обработка информации. DSP функционирует достаточно автономно, в том смысле, что даже и не "подозревает" о существовании микроконтроллера AVR, не говоря уже о наличии **USB** шины, все взаимодействие с которой взял на себя AVR (см. *§ 2.2. "Организация USB интерфейса"*). Именно и только AVR, в соответствии с поступающими по **USB** с хост-компьютера запросами (командами), обеспечивает всё необходимое функциональное согласование Вашего приложения с сигнальным процессором модуля, заставляя его выполнять те или иные алгоритмы работы.

Можно сказать, что микроконтроллер AVR оперативно связан с DSP двумя различными способами (методами). Первый из них заключается во взаимном использовании трех отдельных цифровых линий. Так, AVR имеет возможность инициировать в DSP прерывание *IRQE*. В штатном *LBIOS'*е данный цифровой сигнал используется в качестве так называемого "командного прерывания" (см. *ниже*). DSP же, со своей стороны, может генерировать в AVR целых два прерывания *INT0* и, в зависимости от ревизии модуля, *ANA_COMP* или *INT2* (см. описание *AVR AT90S8515* или *ATmega8515*). В штатном *LBIOS'*е данные прерывания используются для управления потоками данных с АЦП и на ЦАП соответственно.

Второй способ взаимодействия AVR с сигнальным процессором основывается на такой архитектурной особенности DSP, как свой собственный контроллер доступа к внутренней памяти (ПДП) или так называемый канал IDMA (Internal Direct Memory Access). Подробнее о работе канала IDMA см. книгу *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 11.3 "IDMA Port", Analog Devices, Inc., Third Edition September 1995*, которую можно без проблем скачать с нашего *ftp*-сайта: *ftp.lcard.ru/pub/users/adsp/adsp21xxmanualeng.exe* (английский язык) или *ftp.lcard.ru/pub/users/adsp/adsp21xxmanualrus.exe* (русский язык). Микроконтроллер AVR способен по запросам (командам) из хост-компьютера осуществлять те или иные операции прямого доступа к любой ячейки памяти DSP. Таким образом, благодаря указанной возможности приложение в PC через посредство AVR может обращаться к любой ячейке памяти сигнального процессора, не прерывая при этом работу самого DSP. Это исключительно удобно при построении алгоритмов, работающих в реальном масштабе времени. Кроме возможности собственно прямого доступа к памяти, протокол работы с каналом IDMA DSP предусматривает также такую важную процедуру как начальная загрузка сигнального процессора управляющей программой (драйвером), которая и будет осуществлять требуемые алгоритмы обработки и ввода-вывода информации (подробности загрузки DSP см. *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 11.3.5 "Boot Loading Through the IDMA Port", стр. 11-24, Analog Devices, Inc., Third Edition September 1995*). Т.е. микроконтроллер AVR по командам из хост-компьютера способен осуществлять загрузку всей памяти сигнального процессора требуемым программным кодом.

Фирменный драйвер DSP работает по принципу команд, и для реализации такой возможности используется прерывание *IRQE* сигнального процессора, так называемое "*командное прерывание*". Сначала в соответствующую ячейку памяти программ DSP (предопределенная константа *L_COMMAND_E440*, см. *§ 1.5.3.1. "Переменные драйвера DSP"*) заносится номер команды (см. *§ 1.5.3.2 "Номера команд драйвера DSP"*), которую драйвер *LBIOS* впоследствии должен выполнить. Затем в DSP инициируется командное прерывание *IRQE*, в ответ на которое обработчик данного прерывания, содержащийся в самом *LBIOS,* выполняет соответствующие данной команде действия. Названия штатных команд для DSP говорят сами, для чего каждое из них предназначено, а более полное понимание их функционирования можно при желании проследить по исходным текстам *LBIOS* (см. директорию `\E14-440\DSP`).

Цифровой сигнальный процессор получает данных с АЦП, контролирует цепи коммутатора входных сигналов, коэффициенты усиления программируемого усилителя, частоту запуска АЦП и, при необходимости, синхронизирует ввод данных с АЦП либо по сигналу с цифровой линии **TRIG** внешнего разъёма *DRB-37M*, либо по выбранному аналоговому каналу. Также DSP обеспечивает управление микросхемой двухканального ЦАП через посредство своего последовательного порта (*SPORT0*). Функциональное состояние внешних цифровых линий на разъеме *DRB-37M* определяется DSP с помощью простых операций чтения/записи нулевой ячейки своего пространства ввода/вывода (I/O Memory Space), а также с помощью флага *PF0* управляет доступностью выходных линий. И, наконец, сигнальный процессор обеспечивает взаимодействие с микросхемой ППЗУ, предоставляя полный доступ к любой его ячейке.

В штатном пакете программного обеспечения, прилагаемого к модулю *E14-440*, поставляются все исходные тексты драйвера *LBIOS*, написанные на языке ассемблер цифрового сигнального процессора. Их можно найти в директории `\E14-440\DSP`. Исходные тексты достаточно подробно прокомментированы и могут быть с успехом использованы в качестве законченного примера программирования данного модуля при модификации штатного или создании собственного драйвера устройства.

2.4.2. Создание управляющей программы

У пользователя, как правило, не появляется необходимость в написании своих собственных управляющих программ для данного модуля, т.к. все наиболее часто требуемые алгоритмы работы уже реализованы в фирменном драйвере, находящимся в файле `E440.bio`. Если у Вас все-таки возникла необходимость в создании собственной управляющей программы (например, для реализации какого-либо специализированного алгоритма действия DSP), то для этого придется освоить достаточно несложный язык ассемблера для сигнального процессора. В качестве готового примера программирования модуля на таком языке можно использовать исходные тексты фирменного драйвера, хранящиеся в файлах `DSP\E440.DSP` и `DSP\*.H`.

Процесс формирования собственной управляющей программы для DSP потребует от Вас приложения определенных усилий:

1. Вам необходимо будет изучить достаточно понятную архитектуру процессоров семейства ADSP-218x, а также освоить довольно несложный язык программирования – ассемблер DSP. Всю подробную информацию об этом можно найти в оригинальной книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", Analog Devices, Inc., Third Edition, September 1995*, которую можно без проблем скачать с нашего *ftp*-сайта *ftp.lcard.ru/pub/users/adsp/adsp21xxmanualeng.exe*. Также существует книга с русским переводом *"Руководство пользователя по сигнальным микропроцессорам семейства ADSP-2100", под редакцией А.Д.Викторова, Санкт-Петербург, 1997*, которую опять же можно скачать с нашего *ftp*-сайта *ftp.lcard.ru/pub/users/adsp/adsp21xxmanualrus.exe*. Очень подробное описание многочисленных примеров и алгоритмов программирования DSP с исходными текстами приводятся в двухтомном справочнике *"Digital Signal Processing Applications Using the ADSP-2100 Family", Analog Devices, Inc.*, который можно скачать с нашего *ftp*-сайта *ftp.lcard.ru/pub/users/adsp/adsp21xxapplicationseng.exe*. Также много полезной информации в дополнение к указанной выше документации можно обнаружить на сайте *www.analog.com*.
2. Процессоры семейства ADSP-218x поддерживаются полным набором программных средств отладки. Этот набор включает в себя несколько программ: построитель системы (`bld21.exe`), ассемблер (`asm21.exe`), линкер или редактор связей (`ld21.exe`) и т.д. Весь пакет разработчика программ для сигнальных процессоров семейства ADSP-21xx, содержащий все вышеуказанные средства кроме `bld21.exe`, можно скачать с нашего *ftp*-сайта *ftp.lcard.ru/pub/users/adsp/dsptools.exe*. В качестве архитектурного файла следует использовать `E440.ACH`. Все эти программы достаточно подробно описываются в оригинальной книге *"ADSP-2100 Family Assembler Tools & Simulator Manual", Analog Devices,*

Inc., Second Edition, November 1994, которую можно скачать с нашего *ftp*-сайта *ftp.lcard.ru/pub/users/adsp/adsp21xxassemblertool.exe*.

3. Итак, если у Вас не возникло проблем с первыми двумя этапами, то теперь можно приступать к собственно написанию Вашей управляющей программы. Для начала надо создать соответствующие файлы с исходным кодами Вашей программы на языке ассемблер DSP. Затем эти файлы необходимо оттранслировать (`asm21.exe`) и скомпоновать с помощью редактора связей (`ld21.exe`), формируя, таким образом, выполняемую программу типа `.EXE`, так называемый файл отображения в памяти (memory image file). **!!! Не путать с обычной исполняемой DOS/Windows программой типа `.EXE`!!!** Формат сформированного файла отображения в памяти очень подробно описан в *"ADSP-2100 Family Assembler Tools & Simulator Manual", Appendix **B** "File Format", **B.2** "Memory Image File (.EXE)", Analog Devices, Inc., Second Edition, November 1994*. Именно в этом файле содержатся все коды инструкций Вашей программы с соответствующими адресами их расположения в памяти программ DSP, а также инициализирующие значения Ваших переменных и адреса их нахождения в памяти данных. Зная всю эту, информацию нужно просто аккуратно загрузить ее в память DSP по надлежащим адресам. Для упрощения процедуры загрузки полученный файл отображения в память `*.EXE` преобразуется с помощью утилиты `DSP\BIN3PCI.EXE` в файл `*.BIO` (подробности см. *Приложение В*).

Вообще-то, всю эту последовательность создания файла `*.BIO` можно проследить по содержанию файлу `DSP\E440.BAT`. Созданный таким образом файл с управляющей программой `*.BIO` затем используется, например, в штатной функции загрузки сигнального процессора ***LOAD_MODULE()***. Подробности об этой функции смотри *§ 1.5.4.6. "Загрузка драйвера DSP"*.

При желании, для создания собственного драйвера сигнального процессора для модуля *E14-440* можно воспользоваться такой продвинутой интегрированной средой разработки как *VisualDSP++*. На сегодняшний момент для работы с *ADSP-2185M* доступна версия 3.5. Подробности смотри на сайте *www.analog.com*.

2.4.3. Загрузка управляющей программы в DSP

Перед началом работы с устройством *E14-440* необходимо его "оживить", т.е. загрузить в DSP модуля либо фирменный драйвер, находящийся в файле `E440.bio`, либо Вашу собственную управляющую программу. Только после выполнения такой процедуры модуль будет корректно работать с функциями штатной или Вашей, если создадите, библиотеки. Формат файла `E440.bio`, содержащего все коды инструкций управляющей программы, а также переменных *LBIOS*, подробно описан в *Приложении В*. Эти коды инструкций драйвера *LBIOS* и начальные значения его переменных необходимо расположить по соответствующим адресам памяти программ сигнального процессора и запустить управляющую программу на выполнение (в штатном драйвере память данных DSP изначально пуста и, следовательно, в неё ничего загружать не надо). Все это и называется загрузка управляющей программы в DSP или просто загрузка DSP.

Сама процедура начальной загрузки управляющей программы во внутреннюю память DSP модуля достаточно проста и хорошо описана в *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 11.3.5 "Boot Loading Through the IDMA Port", стр. 11-24, Analog Devices, Inc., Third Edition September 1995*. Она предполагает выполнение следующих достаточно несложных шагов:

- произведите сброс (*RESET*) DSP на данном модуле, например, с помощью интерфейсной функции ***DSP_RESET()*** (см. *§ 1.5.4.10. "Сброс модуля"*);
- заполните требуемой информацией память программ и/или данных DSP, кроме ячейки памяти программ с адресом 0x0000 (или короче PM(0x0000)), для чего можно, например, воспользоваться соответствующими интерфейсными функциями ***PUT_DM_ARRAY()*** (см. *§ 1.5.5.7. "Запись массива слов в память данных DSP"*) или ***PUT_PM_ARRAY()*** (см. *§ 1.5.5.8. "Запись массива слов в память программ DSP"*).

– запишите данные в ячейку памяти программ по адресу PM(0x0000) для старта Вашей управляющей программы (при этом ее выполнение начнется с инструкции, находящейся в PM(0x0000)).

Внимание!!! Необходимо обязательно убедиться, что Вы загрузили всю нужную информацию в память DSP до записи данных по адресу PM(0x0000).

В общем-то, ВСЁ! Теперь можно спокойно приступать к управлению модулем с помощью интерфейсных функций штатной или Вашей библиотеки.

2.4.4. Порты управления

В качестве интерфейса цифрового сигнального процессора с некоторой частью периферии модуля (а конкретнее АЦП и цифровые линии) используются так называемые порты управления, которые отображены на пространство ввода-вывода DSP (I/O Memory Space). Подробное описание I/O Memory Space можно найти, например, в оригинальной книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 10.6.4 "ADSP-2181 I/O Memory Space", стр. 10-32, Analog Devices, Inc., Third Edition September 1995*.

На модуле аппаратно организованы четыре порта управления (см. *Таблицу 9*). При взаимодействии с ними дешифрируется только младший бит адреса при обращении к ячейке пространства ввода-вывода DSP (следовательно, адреса портов можно устанавливать кратным двум). Цифровой сигнальный процессор рассматривает все свое пространство ввода-вывода (всего 2048 ячеек), как 4 области по 512 ячеек и обладает возможностью устанавливать для каждого такого сегмента индивидуальное количество тактов ожидания (*waitstates*) для обращения к нему. За конфигурацию тактов ожидания для I/O Memory Space отвечает служебный регистр сигнального процессора Waitstate Control Register, расположенный по адресу DM(0x3FFE). Таким образом, часть портов управления можно расположить в одной области пространства ввода-вывода, а часть – в другой и т.д. Это позволяет, установив задержку на одну область, не замедлить доступ к портам в другой. В штатном *LBIOS'*е четыре порта управления расположены в первой области пространства ввода-вывода и для любой операции обращения к ним устанавливается один дополнительный такт ожидания. В штатном *LBIOS'*е это выглядит так:

```
{ Константа для обозначения адреса регистра Waitstate Control Register      }
     const Dm_Wait_Reg = 0x3FFE;

     . . . . .

{ Set 1 Waitstate for I/O Memory Space                                        }
    AR = 0x0001;              { 0000 0000 0000 0001                           }
    DM(Dm_Wait_Reg) = AR;     { 0x3FFE - Waitstate Control Register           }
```

Краткое описание используемых портов управления приведено в нижеследующей таблице.

Таблица 9. Порты управления DSP

Название порта	Доступ	Назначение	Адрес
TTL_IN	Чтение	Чтение состояния 16[ти] входных цифровых линий на внешнем разъёме *DRB-37F*.	xxx0H
TTL_OUT	Запись	Установка 16[ти] выходных цифровых линий на внешнем разъёме *DRB-37F*.	xxx0H
READ_ADC	Чтение	Чтение данных из АЦП	xxx1H
SET_ADC_CHANNEL	Запись	Управление входным коммутатором и коэффициентом усиления канала АЦП	xxx1H

2.4.5. Конфигурирование флагов DSP

Микросхемы цифровых сигнальных процессоров типа *ADSP-2185M* имеют в своём составе восемь дополнительных неспециализированных выводов, так называемых флагов, обозначаемых обычно **PF0÷PF7**. Данные флаги, при желании, могут быть индивидуально запрограммированы как на вход, так и на выход. На микросхеме указанного типа сигнального процессора выводы этих флагов обладают совмещенными функциями. Так, флаг **PF6** совмещен с входом прерывания *IRQL1*, а флаг **PF7** – с *IRQ2* и т.д. (все необходимые подробности можно найти в техническом описании (*datasheet*) данного DSP на сайте *www.analog.com*). Все флаги и краткое описание их назначения для модуля *E14-440* приведены в следующей таблице:

Таблица 10. Флаги сигнального процессора.

Название флага	***Направ-ление***	***Назначение***
PF0	Выход	При **PF0**=1 переводятся в третье, *высокоимпедансное*, состояние все 16^{ть} выходных цифровых линий на внешнем разъёме *DRB-37F* модуля.
PF1	Выход	Прерывание в **AVR**. В штатном *LBIOS*'е используется для инициализации передачи данных, полученных с АЦП, в хост-компьютер. Т.е., когда DSP собрал соответствующую половину FIFO буфера АЦП, следует 'дернуть' этим флажком вверх-вниз. При этом сгенерится прерывание в **AVR**, обработчик которого начнет передачу этих готовых данных в хост-компьютер. Нормальное состояние – 0.
PF2	-------	Не используется.
PF3	Выход	Прерывание в **AVR**. В штатном *LBIOS*'е используется для инициализации передачи данных, полученных с хост-компьютера, в соответствующую половинку FIFO буфера ЦАП. Т.е., когда DSP отослал очередную половинку FIFO буфера, следует 'дернуть' этим флажком вверх-вниз. При этом сгенерится прерывание в **AVR**, обработчик которого начнет передачу новых данных из хост-компьютера в память DSP. Нормальное состояние – 0.
PF4/IRQE	Вход	*Командное прерывание* для DSP.
PF5	Выход	Прямая запись в выходной регистр номера/усиления канала.
PF6	-------	Не используется.
PF7/IRQ2	Вход	Прерывание от АЦП (конфигурируется для работы по фронту).
FL0	Выход	**FL0**=1 выбор микросхемы пользовательского ППЗУ (*Chip Select*).
FL1	Выход	Клоки пользовательского ППЗУ.
FL2	Выход	Данные в пользовательское ППЗУ.
FI	Вход	Данные из пользовательского ППЗУ.
FO	Выход	Разрешение загрузки данных в ЦАП. Активный уровень – низкий

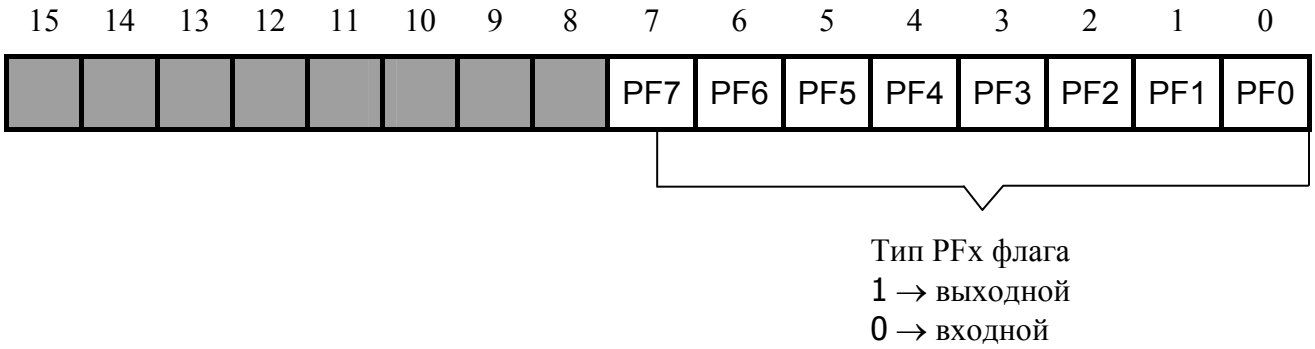
Таким образом, флаги PFx необходимо запрограммировать на вход-выход надлежащим образом, т.е. флаги PF2, PF4, PF6 и PF7 как входные, а флаги PF0, PF1, PF3 и PF5 как выходные.

Программирование флагов осуществляется с помощью двух управляющих регистров DSP:

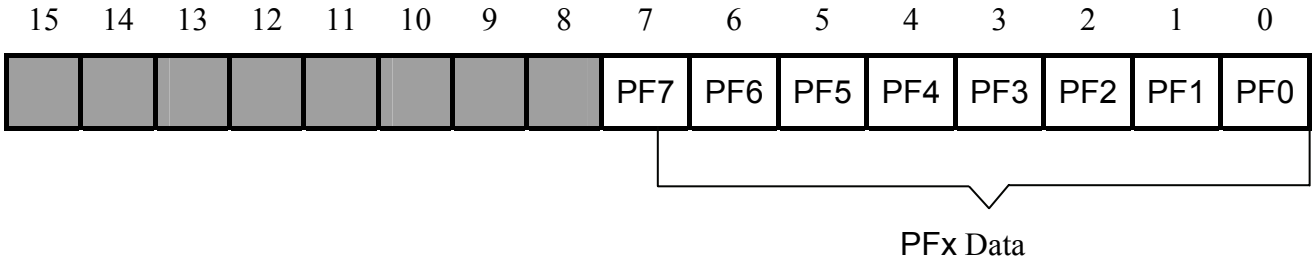
- ✓ регистр Programmable Flag & Composite Select Control, находящийся по адресу DM(0x3FE6) в памяти данных DSP, управляет направлением флага
  - 1 → выход,
  - 0 → вход;
- ✓ регистр Programmable Flag Data, расположенный по адресу DM(0x3FE5), используется для записи и считывания значений флагов.

Формат данных регистров приведен ниже и подробно описан в *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 9.5 "Flag Pins", cmp. 9-15, Analog Devices, Inc., Third Edition, September 1995.*

Programmable Flag & Composite Select Control (DM(0x3FE6))



Programmable Flag Data (DM(0x3FE5))



Например, в штатном *LBIOS*'е для надлежащего программирования флагов FO, FLx и PFx написан следующий код:

```
{ Введем константы для обозначения адресов регистров DSP,            }
{ отвечающих за управление флагами PF0÷PF7                           }
     const Prog_Flag_Data = 0x3FE5;
     const Prog_Flag_Comp_Sel_Ctrl = 0x3FE6;
     . . . . .
{ установим флаг FO в низкое состояние - разрешение загрузки ЦАП     }
     RESET FLAG_OUT;

{ установим флаги FLx в исходное низкое состояние                     }
     RESET FL0, RESET FL1, RESET FL2;

{ Флаг PF0 в 1 (переводим все линии TTL OUT на внешнем цифровом      }
{ разъёме в третье (высокоимпедансное) состояние)                     }
{ Флаг PF5 в 0 (запись буферный регистр адреса/усиления канала)      }
     AR=0x01;
     DM(Prog_Flag_Data)=AR;
```

```
{ Отконфигурируем все флаги PFx:                                    }
{                    PF0,PF1,PF3 и PF5 - выходные,                  }
{                    PF2,PF4,PF6 и PF7 - входные                    }
     AR=0x2B;                          { 0010 1011                  }
     DM(Prog_Flag_Comp_Sel_Ctrl)=AR;
```

2.4.6. АЦП, коммутатор и программируемый усилитель

На модуле *E14-440* установлен 14[ти] битный АЦП *LTC1416* с параллельным выходным портом производства фирмы *Linear Technology Corporation*. Все взаимодействие цифрового сигнального процессора с этим АЦП, а также коммутатором и программируемым усилителем осуществляется через посредство портов *READ_ADC* и *SET_ADC_CHANNEL*, прерывания *IRQ2* (предварительно данное прерывание должно быть обязательно сконфигурировано на работу по фронту) и тактовых синхроимпульсов (клоков) *SCLK1* последовательного порта *SPORT1*. Описание I/O Memory Space можно найти, например, в оригинальной книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 10.6.4 "ADSP-2181 I/O Memory Space", стр. 10-32, Analog Devices, Inc., Third Edition September 1995*. Конфигурирование внешних прерываний DSP подробно описано в той же книге в *§ 3.4.2 "Configuring Interrupt", стр. 3-14* и *§ 9.5 "EXTERNAL INTERRUPTS", стр. 9-14*.

Как упоминалось выше, для обслуживания взаимодействия с АЦП предлагается использовать внешнее прерывание DSP *IRQ2*. Запрос на это прерывание устанавливается по спадающему фронту сигнала –BUSY, а в качестве задатчика сигнала запуска преобразования АЦП (–CONV) используются тактовые клоки *SCLK1* последовательного порта *SPORT1*. Как правило, готовые данные с АЦП считываются в обработчике *IRQ2* путем выполнения операции чтения из порта *READ_ADC*, т.е. первой ячейки пространства ввода-вывода `IO(0x1)`. При заходе в данный обработчик прерывания первым делом необходимо записать в так называемый буферный регистр адреса/усиления очередное значение логического канала, потом считывать готовые данные с АЦП, а далее уже может следовать Ваш алгоритм обработки. Также очень важно, чтобы от момента прихода прерывания до чтения данных из порта *READ_ADC* проходило не более 1.4 мкс. Иначе могут быть получены уже не (N-1)[ые], а N[ые] данные с АЦП (подробнее см. диаграммы сигналов для работы с АЦП на *рисунке ниже*).

Управление адресом (номером) канала и коэффициентом усиления сигнала происходит с помощью двух управляющих регистров:

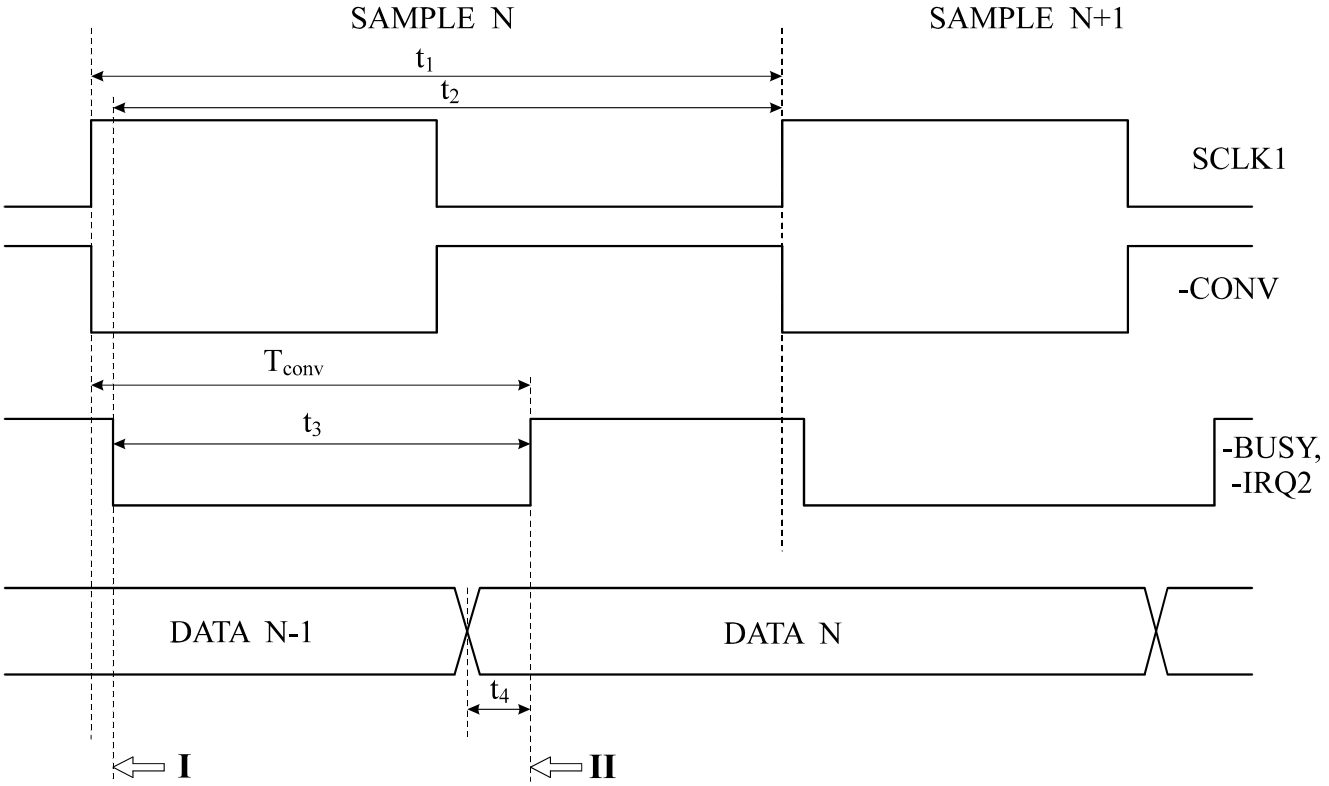
- ✓ *буферный (промежуточный)* регистр адреса/усиления канала;
- ✓ *выходной* регистр адреса/усиления канала.

Информация, находящаяся в *выходном* регистре, определяет адрес канала и коэффициент усиления для текущего преобразования АЦП. Формат этой информации полностью соответствует формату логического канала, описанному в *§ 1.3.2.3 "Логический номер канала АЦП"*. Данные из *буферного* регистра переписываются в *выходной* регистр по спадающему фронту сигнала –BUSY. Таким образом, данные адреса/усиления, записанные в *буферный* регистр при обработке прерывания от выборки N (см. рисунок ниже), будут действовать при преобразовании АЦП для выборки N+2. Запись информации в *буферный* и *выходной* регистры производится через операцию записи в порт *SET_ADC_CHANNEL* (т.е. в первую ячейку пространства ввода-вывода `IO(0x1))`, при надлежащем использовании состояния флага *PF5*. Порядок записи в *выходной* регистр аппаратно определен следующим образом:

1. Сбросить флаг PF5 в ноль, т.е. PF5 = 0.
2. Произвести операцию записи в порт *SET_ADC_CHANNEL* требуемого логического номера канала АЦП. Это фактически означает запись в *буферный* регистр адреса/усиления.
3. Установить PF5 в единицу, т.е. PF5 = 1. При этом информация из *буферного* регистра будет скопирована в *выходной*.
4. Сбросить флаг PF5 в ноль, т.е. PF5 = 0.
5. Произвести операцию записи в порт *SET_ADC_CHANNEL* следующего логического номера канала АЦП. Это опять же означает запись в *буферный* регистр адреса/усиления.

Таким образом, мы прописали необходимой информацией сразу оба буфера и *буферный* и *выходной*. При выполнении этой процедуры следует помнить о двух важных моментах, а именно:

- ✓ Запись в *выходной* регистр должен производиться за время не менее 2 мкс до запуска преобразования АЦП (необходимо время на установления аналогового тракта).
- ✓ Запрещена запись в *выходной* регистр во время активного сигнала -BUSY, т.е. от момента **I** и вплоть до момента **II** (см. ниже *рисунок с диаграммами*).



Временные диаграммы основных сигналов, используемых для работы с АЦП, приведены на следующем рисунке:

На приведенном рисунке через **I** обозначен момент времени:

- ✓ прихода N^{ого} прерывания *IRQ2*, в обработчике которого необходимо считать готовые (N-1)^{ые} данные с АЦП и записать в *буферный* регистр (N+2)^{ой} логический канал;
- ✓ аппаратного копирования содержимого *буферного* регистра с (N+1)^{ым} логическим каналом в *выходной* регистр;

В момент времени **II** аппаратно защелкиваются N^{ые} данные с АЦП в порту *READ_ADC*.

Основные временные характеристики, показанные на рисунке с диаграммами, приведены в таблице ниже:

Таблица 11. Временные характеристики сигналов АЦП

Обозначение	Длительность			Единицы
	min	typ	max	
T_{conv}	1.5	1.9	2.2	мкс
t_1	2.5			мкс
t_2		2.475		мкс

t_3	1.475	1.875	2.175	мкс
t_4	75	100		нс

Частота запуска преобразования АЦП на данном модуле фактически определяется частотой следования внутренних тактовых синхроимпульсов *SCLK1* последовательного порта *SPORT1*. Для надлежащего конфигурирования порта *SPORT1* используются соответствующие регистры сигнального процессора: SPORT1 Control Register и SPORT1 SCLKDIV, адреса которых в памяти данных определены как DM(0x3FF2) и DM(0x3FF1) соответственно. С помощью данных регистров Вы можете разрешить либо запретить генерацию синхроимпульсов, а также напрямую задавать период их следования, однозначно определяя частоту работы АЦП. Исходя из архитектуры сигнального процессора, частота запуска преобразования АЦП определяется по следующей формуле:

$$\mathbf{AdcRate} = \mathbf{F}_{clockout}/(2\cdot(\mathbf{SCLKDIV1}+1)),$$

где $\mathbf{F}_{clockout}$ – тактовая частота DSP равная 48000 кГц, **SCLKDIV1** – коэффициент деления частоты $\mathbf{F}_{clockout}$, который определяется содержимым регистра SPORT1 SCLKDIV. Очень подробное описание работы сериальных портов DSP можно найти в книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", Chapter 5 "Serial Ports", cmp. 5-1, Analog Devices, Inc., Third Edition, September 1995*.

Теперь попробуем проиллюстрировать все выше сказанное примером:

```
{ Для начала отконфигурируем SPORT1                                    }
{ SPORT0 - disable, SPORT1 - disable, SPORT1 – not serial port         }
    AR=0x0000;
    DM(Sys_Ctrl_Reg)=AR;      { 0x3FFF - System Control Register       }
{ ************************************************************** }
{ Set SPORT1 for start of ADC chip                                     }
{ Serial Clock Divide Modulus                                          }
    AR = 99;                  { задаем частоту запуска АЦП (SCLK1)     }
    DM(Sport1_Sclkdiv) = AR;  { 0x3FF1 - Serial Clock Divide Modulus   }
{ Receive Frame Sync Divide Modulus                                    }
    AR = 0xF;              { may be any number: - not used             }
    DM(Sport1_Rfsdiv)=AR;  { 0x3FF0-Receive Frame Sync Divide Modulus }
{ Control word for SPORT1: SCLK1 – external (остановим АЦП)            }
{ high level, alternate external receive frame on each word(16 bit) - not used  }
{ high level, alternate external transmit frame on each word(16 bit) - not used }
    AR = 0x3C1F;              { 0111 1100 0001 1111                    }
    DM(Sport1_Ctrl_Reg) = AR; { 0x3FF2 – SPORT1 Control Register       }
{ ************************************************************** }
{ задержка на 2.0 мкс, чтобы оцифрился последний предыдущий отсчет    }
    cntr=95;
    DO DelayLoop UNTIL CE;
DelayLoop: NOP;
{ очистим все запросы на прерывания                                    }
    IFC = 0xFF; NOP;
{ разрешим прерывания IRQ2                                             }
    IMASK=0x200; NOP;
```

```
{ зададим усиление и номер канала для текущего и следующего отсчетов,}
{ предполагая наличия циклического буфера с управляющей таблицей     }
{ в памяти данных с указателем (I5, M5, L5)                           }

{ PF5 в ноль – промежуточный (буферный) регистр                       }
    AR=DM(Prog_Flag_Data);
    AR=CLRBIT 5 OF AR;
    DM(Prog_Flag_Data)=AR;

{ запись первого канала в буферный регистр                            }
    AR=PM(I5, M5);
    IO(SET_ADC_CHANNEL)=AR;

{ PF5 в единицу – копирование из буферного в выходной регистр         }
    AR=DM(Prog_Flag_Data);
    AR=SETBIT 5 OF AR;
    DM(Prog_Flag_Data)=AR;

{ PF5 в ноль – опять промежуточный (буферный) регистр                 }
AR=DM(Prog_Flag_Data);
AR=CLRBIT 5 OF AR;
DM(Prog_Flag_Data)=AR;

{ запись второго канала в буферный регистр                            }
AR=PM(I5, M5);
IO(SET_ADC_CHANNEL)=AR;

{ задержка на 2.0 мкс, чтобы установился аналоговый тракт             }
    cntr=95;
    DO DelayLoop1 UNTIL CE;
DelayLoop1: NOP;

{ включим клоки SCLK1, запустив АЦП, т.е. сделаем SCLK1 внутренним    }
    AR = 0x7C1F;                { 0111 1100 0001 1111                 }
    DM(Sport1_Ctrl_Reg) = AR;   { 0x3FF2 – SPORT1 Control Register    }

{ Все! Теперь можно ждать прихода прерываний IRQ2                     }
{ Первое пришедшее прерывание IRQ2 будет содержать 'левый' отсчет с   }
{ АЦП и его не надо принимать в расчёт, а вот при последующих         }
{ прерываниях мы будем получать уже то, что надо ☺                    }
{ Частота работы АЦП: AdcRate=48000.0/(2*(99+1))=240.0 кГц            }

    . . . . .

{ Обработчик прерывания IRQ2 может содержать следующие строчки        }
    AR=DM(I5, M5);              { зададим усиление и номер канала для  }
    IO(SET_ADC_CHANNEL)=AR;     { следующего отсчета                   }
    . . . . .
    AR=IO(READ_ADC);            { теперь в AR находится отсчет с АЦП   }
```

2.4.7. Организация сбора данных с АЦП в LBIOS

В штатном *LBIOS*'е для работы с АЦП программно организован циклический двойной FIFO буфер АЦП достаточно большого размера (до 12288 слов), расположенный в памяти данных DSP. При этом всё базисное взаимодействие с АЦП (чтение готовых данных, их корректировка, размещение в буфере, установление очередного логического канала и т.д.) сосредоточено в обработчике прерывания *IRQ2*. Основной же программе остается только периодически (в фоновом режиме) отслеживать степень заполнения текущей половинки FIFO буфера готовыми данными с АЦП. Как только это событие произошло (т.е. текущая половинка буфера полностью заполнилась) DSP сиг-

нализирует об этом факте в микроконтроллер AVR через посредство флага *PF1*. Это является признаком того, что данная половинка FIFO буфера полностью готова к передаче в хост-компьютер по шине **USB**. AVR, используя свою возможность работы с DSP по *каналу IDMA*, должен осуществить данную процедуру передачи, в то время как DSP продолжает непрерывный сбор данных с АЦП в другую половинку FIFO буфера. Для целей приёма данных посылаемых модулем в хост-компьютере можно, например, воспользоваться штатной интерфейсной функцией *ReadData()*.

2.4.8. Корректировка данных с АЦП

Помимо собственно сбора данных *LBIOS* может осуществлять (по желанию пользователя) первичную обработку получаемой информации, заключающуюся в корректировке отсчетов с АЦП. Управление корректировкой в штатном *LBIOS'*е осуществляется через посредство предопределенной переменной *L_CORRECTION_ENABLED_E440*. В качестве корректировочных коэффициентов можно использовать либо собственные, либо штатные, которые хранятся в ППЗУ модуля (см. *§ 1.3.3. "Формат пользовательского ППЗУ"*). Штатные коэффициенты прошиваются в ППЗУ при наладке модуля в **ЗАО "Л-Кард"**. Вся процедура первичной обработки располагается в обработчике прерывания *IRQ2* и обеспечивает корректировку смещения и масштаба получаемых с АЦП данных. Обычная формула для корректировки выглядит следующим образом:

$$U = (V + A)\cdot B,$$

где **V** – полученный отсчет с АЦП, **A** – кор. коэффициент смещения, **B** – кор. коэффициент масштаба, **U** – откорректированное значение отсчета с АЦП. В нашем случае коэффициент **A** это обычное *знаковое* целое $16^{ти}$ битное число. Коэффициент **B** это *беззнаковое* дробное число по величине близкое к 1.0 (например, 0.956 или 1.17). Чтобы произвести процедуру умножения в DSP, коэффициент **B** приходится немного '*причесать*' к виду, удобному для работы сигнального процессора. Для этого необходимо привести его к известному в DSP дробному формату 1.15 (подробнее о форматах см. книгу *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", Appendix C "Numeric Format", стр. C-1, Analog Devices, Inc., Third Edition, September 1995.*), используя следующую формулу:

$$B' = 32768\cdot B + 0.5,$$

где **B'** – $16^{ти}$ битное представление коэффициента **B** в дробном формате 1.15. Именно коэффициенты **A** и **B'** хранятся в служебной области ППЗУ модуля и применяются в процедуре корректировки данных с АЦП на уровне драйвера DSP. Однако следует помнить, что в структуре *MODULE_DESCRIPTION_E440* корректировочные коэффициенты находятся в более привычном виде типа *double*. Также необходимо учитывать, что для каждого *коэффициента усиления* в ППЗУ модуля хранится соответствующая пара коэффициентов **A** и **B'** (см. *§ 1.3.3. "Формат пользовательского ППЗУ"*).

На языке DSP ассемблер операция корректировки выглядит примерно так:

```
{ чтение отсчета с АЦП                                              }
    AR=IO(READ_ADC);
{ корректировка смещения; в регистре AY1 коэффициент A              }
    AY1 = -4; AR=AR+AY1;
{ корректировка масштаба; в регистре MY1 коэффициент B'             }
    MY1 = 32832; MR=AR*MY1(SU);
{ округление результата                                             }
    MR=MR(RND);
{ теперь в регистре MR1 находится откорректированный отсчет с АЦП  }
```

2.4.9. ЦАП

Взаимодействие с микросхемой двухканального 12[ти] битного ЦАП *AD7249* (техническое описание (*datasheet*) можно найти на сайте *www.analog.com*), которая по Вашему желанию может быть установлена на модуле *E14-440*, цифровой сигнальный процессор осуществляет с помощью своего последовательного порта *SPORT0*. Для этого *LBIOS* программирует *SPORT0* надлежащим образом, а именно:

- ✓ длина слова (*word length*) – 16 бит;
- ✓ тактовые синхроимпульсы (*serial clocks*) – внутренние с периодом не менее 0.2 мкс;
- ✓ кадровая синхронизация приема каждого слова;
- ✓ внутренняя кадровая синхронизация приема;
- ✓ альтернативная кадровая синхронизация приема;
- ✓ кадровый сигнал приема активен по низкому уровню;
- ✓ кадровая синхронизация передачи каждого слова;
- ✓ внешняя кадровая синхронизация передачи;
- ✓ альтернативная кадровая синхронизация передачи;
- ✓ кадровый сигнал передачи активен по низкому уровню.

Выводы порта *SPORT0* сигнального процессора для кадровой синхронизации приема и передачи (выводы *RFS0* и *TFS0* соответственно) аппаратно объединены. Таким образом, получается, что внутренне генерируемые сигналы кадровой синхронизации приема являются сигналами внешней кадровой синхронизации для передачи. Частоту же генерирования сигналов на выводе *RFS0* можно программным способом изменять в достаточно широких пределах, устанавливая, таким образом, частоту вывода данных на ЦАП. Очень подробное описание конфигурирования сериальных портов DSP можно найти в *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", Chapter 5 "Serial Ports", стр. 5-1, Analog Devices, Inc., Third Edition, September 1995*.

В штатном *LBIOS'*е период тактовых синхроимпульсов *SCLK0* установлен равным 250 нс с использованием для этого регистр делителя тактовой частоты SPORT0 SCLKDIV, который расположен в памяти данных по адресу DM(0x3FF5). Именно этот период определяет скорость последовательной передачи 16[ти] битного слова данных в ЦАП. Собственно частоту передачи этих слов данных в ЦАП (т.е. частоту работы ЦАП) задаёт другой регистр – SPORT0 RFSDIV, расположенный в памяти данных по адресу DM(0x3FF4). Итого для определения частоты вывода информации на ЦАП можно написать следующую формулу:

$$\mathbf{DacRate} = \mathbf{F}_{clockout}/(2\cdot(\mathbf{SCLKDIV0}+1)\cdot(\mathbf{RFSDIV0}+1)),$$

где $\mathbf{F}_{clockout}$ – тактовая частота DSP равная 48000 кГц, **SCLKDIV0** – коэффициент деления частоты $\mathbf{F}_{clockout}$, который определяется содержимым регистра SPORT0 SCLKDIV (штатно равен 5), **RFSDIV0** –задает периодичность следования внутренне генерируемых сигнальным процессором сигналов *RFS0*, и следовательно *TFS0* (определяется содержимым регистра SPORT0 RFSDIV).

Формат 16[ти] битного слова данных, передаваемого по последовательному порту *SPORT0* в микросхему ЦАП, приведен в следующей таблице:

Номер бита	Назначение
0÷11	12[ти] битный код ЦАП
12	Выбор номера канала ЦАП: ✓ '0' – первый канал; ✓ '1' – второй канал.
13÷15	Не используются

Теперь, научившись корректно организовывать передачу требуемой информации в ЦАП, необходимо сделать еще один шаг и заставить его обновлять напряжение на своих выходах в нужный момент времени в соответствии с уже поступившими данными. За это отвечает флаг *FO* (линия *LDAC* микросхемы ЦАП). Т.е. последовательно переданная информация только защелкивается в соответствующих регистрах микросхемы ЦАП, а реальное обновление напряжения на его выходах наступает только при определенных состояниях флага *FO*. При этом возможны два варианта:

1. Если в момент прихода сигнала *TFS0* (линия *SYNC* микросхемы ЦАП) уровень флага *FO* низкий, то происходит *автоматическое* обновление выходных напряжений ЦАП сразу по получении последнего бита данных.
2. Если в момент прихода сигнала *TFS0* (линия *SYNC* микросхемы ЦАП) уровень флага *FO* высокий, то обновление выходных напряжений ЦАП происходит при переводе флага *FO* в низкое состояние в любой момент времени после окончания передачи данных.

В штатном *LBIOS'*е флаг *FO* раз и навсегда устанавливается в *низкое состояние*, т.е. реализуется первый вариант *автоматической* загрузки в ЦАП получаемой информации.

Приведем пример подготовки порта *SPORT0* для управления ЦАП'ами:

```
{ Для начала отконфигурируем SPORT0 на передачу данных                }
{ SPORT0 - disable, SPORT1 - disable, SPORT1 - not serial port        }
    AR=0x0;
    DM(Sys_Ctrl_Reg)=AR;      { 0x3FFF - System Control Register      }
{ ************************************************************** }
{   Set SPORT0 for transmit digital codes in DAC                      }
{   SCLK0 and Receive Frame - internal, word = 16 bits                }
{   Transmit Frame - external                                         }
{   Serial Clock Divide Modulus                                       }
    AR=5;                     { SCLK0 - internal, T=250 ns            }
    DM(Sport0_Sclkdiv) = AR;  { 0x3FF5 - Serial Clock Divide Modulus  }
{Receive Frame Sync Divide Modulus                                    }
    AR = 49;           { Определяет частоту вывода отсчетов с ЦАП'а   }
    DM(Sport0_Rfsdiv)=AR; {0x3FF4-Receive Frame Sync Divide Modulus   }
{ Control word for SPORT0: SCLK0 - internal                           }
{ low level, alternate internal receive frame on each word(16 bit)    }
{ low level, alternate external transmit frame on each word(16 bit)   }
    AR = 0x7DCF;              { 0111 1101 1100 1111                   }
    DM(Sport0_Ctrl_Reg) = AR; { 0x3FF6 - SPORT1 Control Register      }
{ ************************************************************** }
{ SPORT0 - enable, SPORT1 - disable, SPORT1 - not serial port         }
    AR = 0x1000;              { 0001 1100 0000 0000                   }
    DM(Sys_Ctrl_Reg) = AR;    { 0x3FFF - System Control Register      }
{ Мы установили частоту вывода данных на ЦАП равной                   }
{ 48000/(2*(5+1)*(49+1))=80 Кгц                                       }
{ Теперь с этой частотой будут приходить прерывания SPORT0 Transmit,  }
{ обработчик которого каждый раз должен записывать в регистр         }
{ передачи TX0 очередное значение, посылаемое в микросхему ЦАП       }

    . . . . .
```

```
{ Мы же сейчас просто установим нулевой уровень на первом ЦАП'е      }
    TX0=0x0;

    . . . . .

{ А сейчас установим уровень 5.0 В (код 2047) на втором ЦАП'е        }
    TX0=0x17FF;
```

2.4.10. Организация работы с ЦАП в LBIOS

В штатном *LBIOS*'е работа порта *SPORT0* при *потоковом* выводе на ЦАП организована с использованием так называемого режима *autobuffering* (подробнее о данном режиме можно прочитать в книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", Chapter 5 "Serial Ports", § 5.11 "AutoBuffering", стр. 5-26, Analog Devices, Inc., Third Edition, September 1995*.). Для этого в памяти данных сигнального процессора программно организован циклический двойной FIFO буфер ЦАП достаточно большого размера (до 4032 слов). Параметры этого буфера и разрешение использовать режим *autobuffering* передаются последовательному порту при его инициализации.

Таким образом, сразу после запуска ЦАП (фактически при разрешении последовательного порта) *SPORT0* будет сам, в фоновом режиме для основной программы и без прерывания её работы, передавать всю требуемую информацию из FIFO буфера на ЦАП с предварительно заданной частотой. Тогда задача основной программы резко упрощается, и будет заключаться только в периодическом отслеживании количества переданных в ЦАП слов. По факту завершения вывода данных из очередной половинки FIFO буфера ЦАП DSP сигнализирует об этом в микроконтроллер AVR через посредство флага *PF3*. Это является признаком того, что вся информация на ЦАП из текущей половинки буфера уже успешно выведена, и её можно перезаписывать новыми данными. Из хост–компьютера по шине **USB** должны поступать требуемые порции новых данных для последующего их вывода на ЦАП (для этого, например, можно воспользоваться штатной интерфейсной функцией *WriteData()*), и именно AVR (отвечающий за **USB** интерфейс) по мере необходимости осуществляет подгрузку вновь поступивших данных в надлежащую половинку FIFO буфера ЦАП, используя при этом свою возможность работы с DSP по *каналу IDMA*. При этом в процессе этой подгрузки собственно сам вывод на ЦАП продолжается из другой половинки FIFO буфера.

2.4.11. Управление ТТЛ линиями

Управление цифровыми линиями внешнего разъёма ***DRB-37F*** на модуле *E14-440* осуществляется достаточно просто. Флаг *PF0* позволяет контролировать нахождение всех $16^{ти}$ **выходных** линий разъёма в третьем (высокоимпедансном) или рабочем состоянии. Любая операция записи какого-либо числа в порт *TTL_OUT*, т.е. по адресу 0x0 в пространстве ввода-вывода (I/O Memory Space) сигнального процессора, приводит к установке всех $16^{ти}$ выходных линий на разъёме в состояние, соответствующее битам записываемого значения. Аналогично любая операция чтения из порта *TTL_IN*, т.е. по адресу 0x0 из пространства ввода-вывода (I/O Memory Space) сигнального процессора, позволяет получить состояние всех $16^{ти}$ входных линий на цифровом разъёме ***DRB-37F***. Подробности расположения выводов разъема и краткое описание их назначения см. *"E14-440. Руководство пользователя", § 3.3.2. "Внешний разъём для подключения цифровых сигналов"*. Подробное описание I/O Memory Space можно найти, например, в оригинальной книге *"ADSP-2100 Family User's Manual (Includes ADSP-2171, ADSP-2181)", § 10.6.4 "ADSP-2181 I/O Memory Space", стр. 10-32, Analog Devices, Inc., Third Edition September 1995*.

Теперь можно рассмотреть тривиальный пример работы с цифровыми линиями:

```
{ считаем входные цифровые линии                                     }
    AR=IO(TTL_IN);
{ установим полученные состояния на выходных линиях                  }
    IO(TTL_OUT)=AR;
```

2.4.12. ППЗУ

Как уже отмечалось во *введении к данной главе*, на модуле ***E14-440*** устанавливается микросхема электрически стираемого программируемого ПЗУ (Serial EEPROM) типа ***93C46***. Организация памяти данного устройства представлена следующим форматом: 64 Слова x 16 бит. Очень подробные технические описания (*datasheet*) данного устройства можно найти в Интернете, например, по нижеприведенным ссылкам:

- *www.microchip.com/1000/pline/memory/memdvice/micro/devices/93c46b/index.htm*;
- *www.atmel.com/atmel/products/prod162.htm*.

На модуле ***E14-440*** все программное взаимодействие цифрового сигнального процессора и микросхемы ППЗУ ***93C46*** организовано по следующим линиям:

1. Флаг *FL0* DSP позволяет осуществлять выбор микросхемы ППЗУ. Данный флаг заведен на ножку *Chip Select* (*CS*) микросхемы ***93C46***.
2. Флаг *FL1* DSP используется в качестве тактовых синхроимпульсов (клоков, clocks) при обмене информацией между сигнальным процессором и ППЗУ. Данный флаг заведен на ножку *Serial Data Clock* (*CLK*) микросхемы ***93C46***.
3. Флаг *FL2* используется для последовательной передачи информации (команд, адресов ячеек и собственно данных) из сигнального процессора в ППЗУ. Данный флаг заведен на ножку *Serial Data In* (*DI*) микросхемы ***93C46***.
4. Флаг *FI* используется для последовательного приёма данных из ППЗУ. Данный флаг заведен на ножку *Serial Data Out* (*DO*) микросхемы ***93C46***.

Всю дополнительную информацию, касающуюся работы с микросхемой ППЗУ типа ***93C46***, т.е. временны**е** диаграммы, набор и формат команд управления, инструкции по работе с устройством и многое, многое другое, можно с лихвой обнаружить в упомянутых выше технических описаниях. А практическое воплощение взаимодействия DSP и микросхемы ППЗУ ***93C46*** при необходимости нетрудно найти в исходных текстах драйвера *LBIOS* (см. файл `\E14-440\DSP\flash.h`)

2.4.13. Внешняя синхронизация сигнального процессора

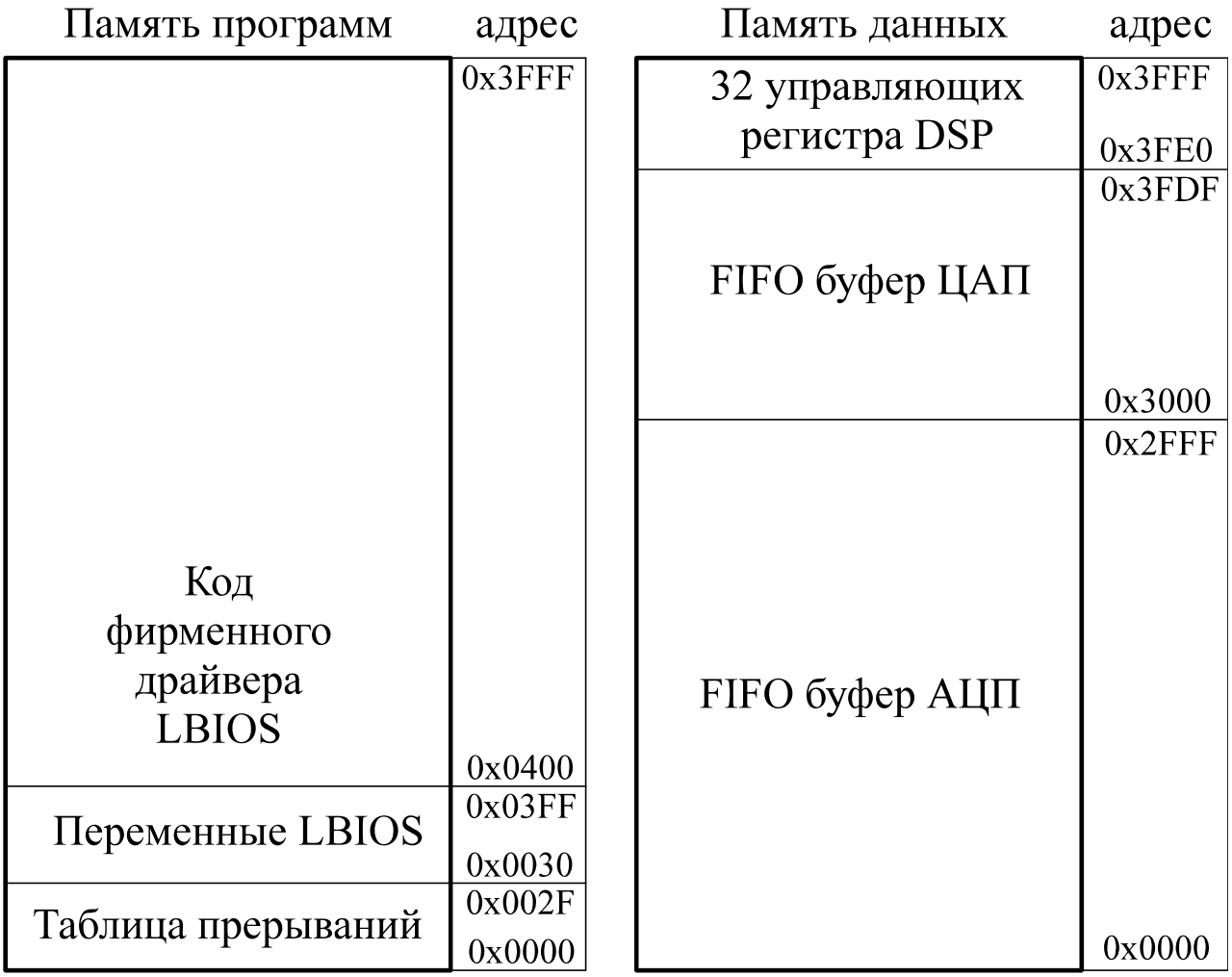
Для задач, которые требуют дополнительной синхронизации работы DSP с внешними устройствами, предусмотрено использование следующих ТТЛ совместимых выводов на внешних разъёмах модуля (см. *"E14-440. Руководство пользователя"*):

- ✓ вывод **TRIG** на внешнем аналоговом разъёме ***DRB-37M***, который подключен к линии прерывания *IRQ1* цифрового сигнального процессора. В фирменном драйвере *LBIOS* это прерывание отконфигурировано для работы по фронту и используется для различных режимов цифровой синхронизации ввода данных с АЦП.
- ✓ вывод **INT** на цифровом разъёме ***DRB-37F***, который подключен к линии прерывания *IRQ0* цифрового сигнального процессора (прерывание может быть отконфигурировано для работы по уровню или по фронту). В фирменном драйвере *LBIOS* эта линия не используется.

Каждая из указанных линий прерывания может быть индивидуально сконфигурированы на работу, как по фронту, так и по уровню (за это отвечает системный регистр DSP под названием *ICNTL*). Если прерывание работает по фронту, то оно генерируется при отрицательном перепаде импульса (⎍) длительностью не менее 50 нс. В случае же конфигурации по уровню, соответствующая линия прерывания должна оставаться в активном низком уровне до тех пор, пока сигнальный процессор не начнет обслуживание данного прерывания. В обработчике прерывания соответствующую ему линию обязательно надо сбрасывать в высокое исходное состояние, чтобы это прерывание не обрабатывалось повторно.

Приложение А.
Структура памяти драйвера DSP

Карта распределения памяти программ и памяти данных для цифрового сигнального процессора *ADSP-2185M* и расположение в ней программных блоков драйвера *LBIOS* приведена на следующем рисунке:



Карта распределения памяти для ADSP-2185

Приложение B.
Утилита BIN3PCI.EXE и формат файла .BIO

Досовская утилита BIN3PCI.EXE предназначена исключительно для целей преобразования стандартного формата файла отображения в памяти (memory image file), сформированного редактором связей DSP ld21.exe, в формат .BIO, применяемый в **ЗАО "Л-Кард"** и являющийся более удобным для процесса загрузки цифрового сигнального процессора. Стандартный формат файла отображения в памяти (memory image file) в деталях описан в оригинальной книге *"ADSP-2100 Family Assembler Tools & Simulator Manual", Appendix B "File Format", B.2 "Memory Image File (.EXE)", Analog Devices, Inc., Second Edition, November 1994*. Для выполнения процедуры преобразования формата, например файла отображения в памяти E440.exe в файл E440.bio, в командной строке необходимо набрать следующую строчку:

```
bin3pci E440.exe
```

Файл .BIO содержит в обычном бинарном виде массив 16ти слов типа **WORD** (в C++), формат которого представлен в нижеследующей таблице:

Таблица 12. Формат файла BIO

Индекс	Назначение
0	Общее количество слов (**NPM**) типа **WORD** (в C++), которые надо грузить в память программ DSP, начиная с адреса PM(0x0)
1	Старшие 16 бит из 24ого битного слова памяти программ, загружаемого по адресу PM(0x0)
2	Младшие 8 бит из 24ого битного слова памяти программ, загружаемого по адресу PM(0x0)
3	Старшие 16 бит из 24ого битного слова памяти программ, загружаемого по адресу PM(0x1)
4	Младшие 8 бит из 24ого битного слова памяти программ, загружаемого по адресу PM(0x1)
.....	
NPM-1	Старшие 16 бит из 24ого битного слова памяти программ, загружаемого по адресу PM((**NPM**–2)/2)
NPM	Младшие 8 бит из 24ого битного слова памяти программ, загружаемого по адресу PM((**NPM**–2)/2)
NPM+1	Общее количество слов (**NDM**) типа **WORD** (в C++), которые надо грузить в память данных DSP, начиная с адреса DM(0x0)
NPM+2	Первое 16ти битное слово, загружаемое по адресу DM(0x0)
NPM+3	Второе 16ти битное слово, загружаемое по адресу DM(0x01)
.....	
NPM+NDM+1	Последнее 16ти битное слово, загружаемое по адресу

	DM(0x0+(**NDM**–1))

Следует помнить, что штатно загрузка памяти данных для штатного *LBIOS'*а не происходит поскольку, он написан таким образом, что изначально память данных DSP не инициализирована.

Приложение C.
Вспомогательные константы и типы

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

C.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.

C.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
};
```

C.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 Comment[COMMENT_LINE_LENGTH_LUSBAPI];   // строка комментария
};
```

C.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
};
```

C.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
};
```

C.6. Структура DSP_INFO_LUSBAPI

Вспомогательная структура *DSP_INFO_LUSBAPI* содержит информацию об используемом исполнительном устройстве типа DSP. Данная структура описывается следующим образом:

```
struct DSP_INFO_LUSBAPI
{
    BOOL   Active;                      // флаг достоверности остальных полей структуры
    BYTE   Name[NAME_LINE_LENGTH_LUSBAPI]; // название DSP
    double ClockRate;                   // тактовая частота работы DSP в кГц
    VERSION_INFO_LUSBAPI Version;       // информация о драйвере DSP
    BYTE   Comment[COMMENT_LINE_LENGTH_LUSBAPI];    // строка комментария
};
```

C.7. 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
};
```

C.8. 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_LUSBAPI];
    // корректировочные коэффициенты масштаба ЦАП
    BYTE   Comment[COMMENT_LINE_LENGTH_LUSBAPI];     // строка комментария
};
```

C.9. 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
};
```