Программный интерфейс для работы с кабелем связи АБИТ USB – CAN/K-Line

Оглавление

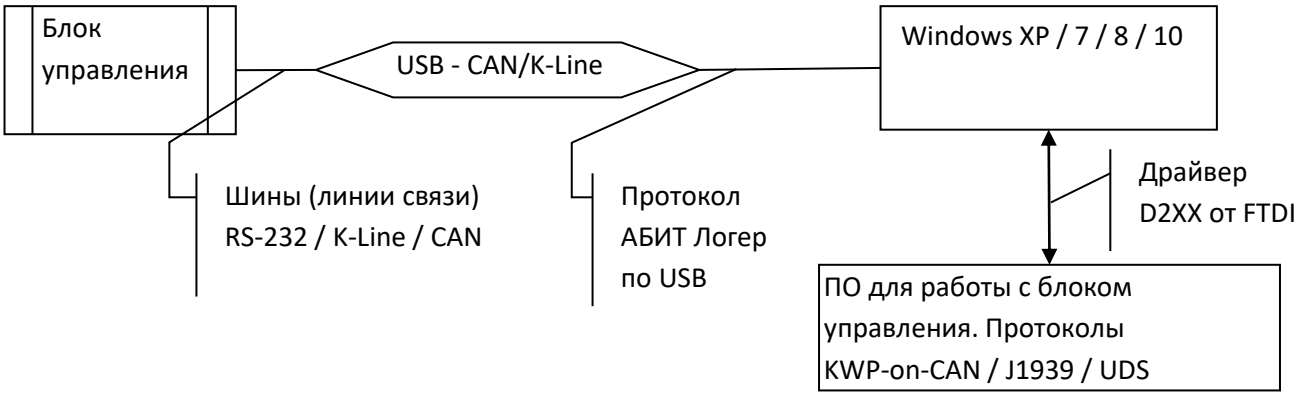
Общие сведения

Кабель связи АБИТ USB – CAN/K-Line (другое название - USB-адаптер CAN/K-Line) предназначен для программного взаимодействия между ПО, выполняемом на ПК под управлением операционной системы Windows и блоками управления (АБИТ, Bosch, Cummins). Кабель связи подключается к стандартному порту USB на ПК, для подключения к блоку управления кабель имеет стандартный разъём диагностической колодки OBD-II.

Выпускаемые кабели связи имеют две модификации:

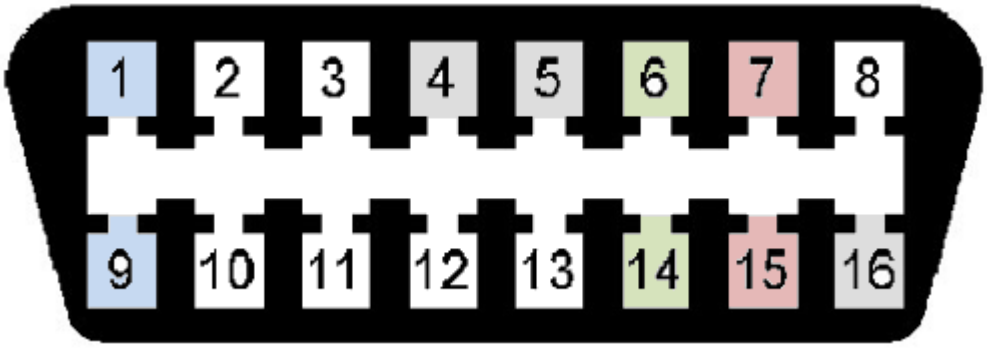
- **Адаптер LNK-34**. Данная модификация позволяет устанавливать связь с блоком управления по диагностической линии связи K-Line (ISO 9141, протоколы UDS (ISO 14229) и KWP2000 (ISO 14230)) и по интерфейсу CAN (ISO 15765, протоколы SAE J1939 и Bosch KWP-on-CAN). Поддерживается одновременная работа по шинам K-Line и CAN.
- **Адаптер LNK-2**. Данная модификация позволяет устанавливать связь с блоком управления только по диагностической линии связи K-Line.

Физическое соединение с блоком управления



Взаимодействие Адаптера и ПК по интерфейсу USB осуществлятся с использованием микросхемы FT232 (USB to serial UART interface) фирмы FTDI. Для программной работы с адаптером на ПК необходимо установить драйвера D2XX FTDI (Future Technology Devices International Ltd.) для операционной системы Windows.

Подключение адаптера к контактам колодки OBD-II



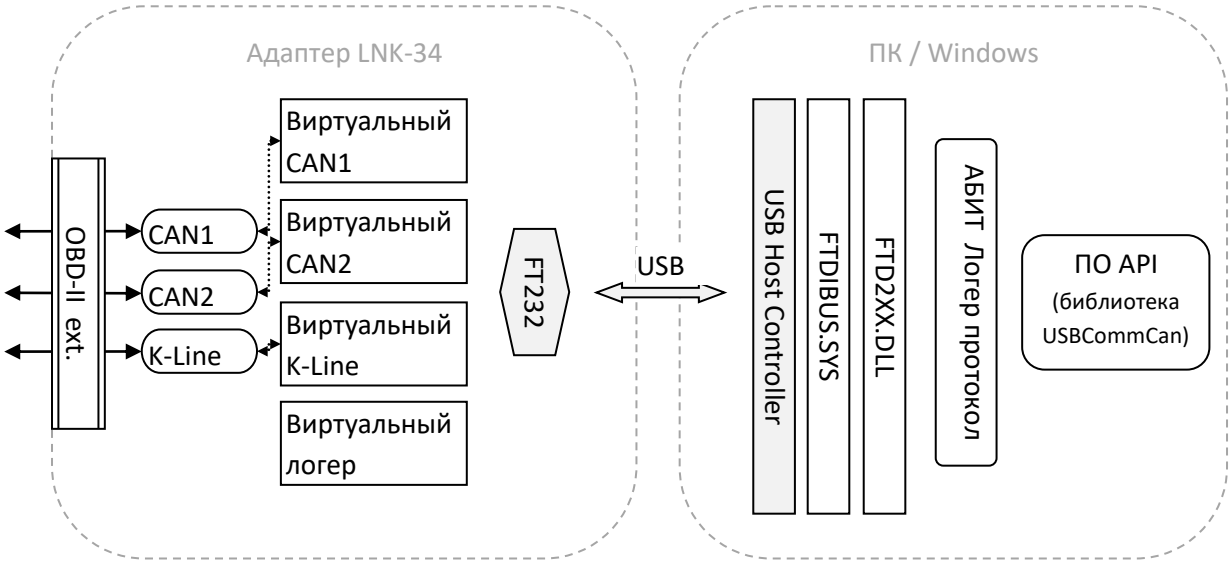
Номер контакта	Описание
1	Прямой CAN2 (CAN2 High, SAE J2284)
4*	Заземление (заземление шасси, chassis ground)
5*	Заземление (сигнальное заземление, signal ground)
6	Прямой CAN1 (CAN1 High, SAE J2284)
7	K линия (K-Line, ISO 9141-2, ISO 14230)
9	Инвертированный CAN2 (CAN2 Low, SAE J2284)
14	Инвертированный CAN1 (CAN1 Low, SAE J2284)
15	L линия (Low, ISO 9141-2, ISO 14230)
16	Питание (+12В)

* Контакты 4 и 5 должны быть соединены между собой.

Протокол АБИТ Логер

Независимо от того, какая именно шина (K-Line или CAN) используется для связи между адаптером и блоком управления, передача данных между ПК и адаптером выполняется по протоколу АБИТ Логер, который, в свою очередь, работает поверх программного интерфейса D2XX (предоставляемого в виде системных драйвером для Windows и реализующего асинхронную последовательную передачу данных через USB).

Программное обеспечение, запрограммированное в адаптер, виртуализирует шины передачи данных к блоку управления, то есть каждая из шин (K-Line или CAN) имеет представление в виде отдельного виртуального устройства. Для каждого из типов шин адаптер поддерживает произвольное количество виртуальных устройств. Например, в случае, если используется расширенная версия колодки OBD-II с двумя линиями CAN, со стороны адаптера для ПК представляются три виртуальных устройства: K-Line, CAN1, CAN2.



Все виртуальные устройства в адаптере пронумерованы, нумерация начинается с 2-х, номер 1 выделен самому Адаптеру (то есть он существует всегда).
Команды делятся по типам виртуальных устройств, каждому типу выделен собственный диапазон номеров команд

Тип устройства	Описание	Список команд	Номер команды
1	Команды взаимодействия с Адаптером (установка связи, периодическая проверка связи). Номер виртуального устройства = 1.	LOGGER_CMD_CONNECT LOGGER_CMD_PING	1 2

2	Команды взаимодействия с K-Line и RS-232. Номера виртуальных устройств = 2, 3.	COMM_CMD_OPEN COMM_CMD_CLOSE COMM_CMD_BAUDRATE COMM_CMD_SEND COMM_CMD_RECEIVE COMM_CMD_BUF_SIZE COMM_CMD_BUF_CLEAR COMM_CMD_SET_PIN COMM_CMD_GET_PIN COMM_CMD_SEND_WUP	51 52 53 54 55 56 57 58 59 60
3	Команды взаимодействия с CAN1 и CAN2. Номера виртуальных устройств = 4, 5.	CAN_CMD_OPEN CAN_CMD_CLOSE CAN_CMD_BUS CAN_CMD_SEND CAN_CMD_SEND_EXT CAN_CMD_RECV_DATA CAN_CMD_RECV_STATUS CAN_CMD_SET_MSG_DELAY	101 102 103 104 109 105 106 107

Таким образом, для адаптера с двумя шинами CAN и одной шиной K-Line, конфигурация виртуальных устройств будет следующая:

Номер	Тип	Описание
1	1	Адаптер
2	2	K-Line
3	2	*<отсутствует>*
4	3	CAN1
5	3	CAN2

Формат команд

Для взаимодействия с адаптером используется командно-ориентированный интерфейс. Обмен командами происходит в двух направлениях:

1. ‘ПК => Адаптер’.
   Так как буфер команд у адаптера ограничен, в каждой команде, пришедшей от адаптера, помимо прочей информации, содержится текущий размер приемного буфера команд (для каждого логического устройства используется отдельный приемный буфер).

2. ‘Адаптер => ПК’.
   Все данные, пришедшие от внешних интерфейсов (K-Line / RS-232 / CAN), сразу же передаются в выходной буфер, буферизации внутри адаптера не выполняется.

Формат команд ‘ПК => Адаптер’

Номер байта	Описание
1	Номер виртуального устройства
2	Номер команды
3-4	Длина данных команды (в байтах)
5..x	Данные

Формат команд ‘Адаптер => ПК’

Номер байта	Описание
1	Номер виртуального устройства
2	Номер команды
3-4	Длина данных команды (в байтах)
5-6	Размер приемного буфера адаптера (в байтах)
7..x	Данные

Для передачи значений, состоящих из 2-х и более байт, используется формат “little-endian”, то есть самый младший байт значения передаётся первым.

Команды виртуального устройства “Адаптер”

Номер данного виртуального устройства всегда равен 1 (нумерация остальных виртуальных устройств начинается с 2).

Команда “Установка связи”

Номер команды равен 1.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Номер 1-го виртуального устройства	1..255
2	Тип 1-го виртуального устройства	1..3
3..11	Имя 1-го виртуального устройства	ASCII символы
[12]	Номер 2-го виртуального устройства	

[13]	Тип 2-го виртуального устройства	
[14..22]	Имя 2-го виртуального устройства	
…	<Следующие виртуальные устройства>	

Пример

Данные запроса

Значение байта (hex)	Описание
01	Запрос к 1-му устройству (к Адаптеру)
01	Номер команды = 1 (Установка связи)
00	Младший байт длины данных команды
00	Старший байт длины данных команды

Данные ответа

Значение байта (hex)	Описание
01	Ответ от 1-го устройства (от Адаптера)
01	Номер команды = 1 (Установка связи)
1e	Младший байт длины данных команды
00	Старший байт длины данных команды
68	Младший байт размера приёмного буфера адаптера
db	Старший байт размера приёмного буфера адаптера
01	Номер виртуального устройства *<1=Адаптер>*
01	Тип виртуального устройства *<1=Адаптер>*
55	1-й байт имени
53	
42	
43	
41	
4e	
5f	
5f	8-й байт имени *<USBCAN__>*
02	Номер виртуального устройства *<2>*
02	Тип виртуального устройства *<2=K-Line>*
4b	1-й байт имени
4c	
69	
6e	
65	
5f	
5f	
5f	8-й байт имени *<KLine___>*
04	Номер виртуального устройства *<4>*

03	Тип виртуального устройства *<3=CAN>*
43	1-й байт имени
41	
4e	
31	
5f	
5f	
5f	
5f	8-й байт имени *<CAN_____>*

Команда “Проверка связи”

Номер команды равен 2.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	До версии прошивки 40 отсутствует. Начиная с версии 40 – номер версии оборудования.	34..255
2	До версии прошивки 40 отсутствует. Начиная с версии 40 – номер версии прошивки адаптера.	40..255

Пример

Данные запроса

Значение байта (hex)	Описание
01	Запрос к 1-му устройству (к Адаптеру)
02	Номер команды = 2 (Проверка связи)
00	Младший байт длины данных команды
00	Старший байт длины данных команды

Данные ответа

Значение байта (hex)	Описание
01	Ответ от 1-го устройства (от Адаптера)

02	Номер команды = 2 (Проверка связи)
02	Младший байт длины данных команды
00	Старший байт длины данных команды
69	Младший байт размера приёмного буфера адаптера
db	Старший байт размера приёмного буфера адаптера
22	Номер версии оборудования <34>
2b	Номер версии прошивки адаптера <43>

Команды виртуальных устройств “RS-232 / K-Line”

Номера виртуальных устройств равны 2 и 3.

Команда “Открытие устройства”

Номер команды равен 51.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт скорости устройства	
2	Средний байт скорости устройства	
3	Старший байт скорости устройства	Baud rate: 110..625000
4	Чётность	Parity: 0..4
5	Биты данных	Data bits: 7..8
6	Стоп-биты	Stop bits: 0..2

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Закрытие устройства”

Номер команды равен 52.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Изменение скорости устройства”

Номер команды равен 53.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт скорости устройства	
2	Средний байт скорости устройства	
3	Старший байт скорости устройства	Baud rate: 110..625000

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Отсылка данных”

Номер команды равен 54.

Команда позволяет отослать данные по шине RS-232 / K-Line. Размер данных может варьироваться от 1 до 260 байт (максимальный размер буфера для команды UDS).

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Данные	0..255
..		
[260]		

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Получение данных”

Номер команды равен 55.

Команда не является ответом на команду ПК, она посылается адаптером сразу после получения данных по физической шине. Размер данных может варьироваться от 1 до 260 байт (максимальный размер буфера для команды UDS).

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Данные	0..255
..		
[260]		

Команда “Получение размера буфера данных”

Номер команды равен 56.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Тип буфера (Input Buffer = 1 / Output Buffer = 2)	1..2

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт размера буфера данных	
2	Старший байт размера буфера данных	0..65535

Команда “Очистка буфера данных”

Номер команды равен 57.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Тип буфера (Input Buffer = 1 / Output Buffer = 2 / All Buffers = 3)	1..3

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Установка состояния линии”

Номер команды равен 58.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Тип линии (RTS = 1 / DTR = 2 / CTS = 3)	1..3
2	Тип изменения (Clear = 0 / Set = 1 / Reset(XOR) = -1)	-1..1

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Чтение состояния линии”

Номер команды равен 59.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Тип линии (RTS = 1 / DTR = 2 / CTS = 3)	1..3

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Состояние линии (Clear = 0 / Set = 1)	0..1

Команда “Посылка Fast Initialization Wake Up Pattern (WuP)”

Номер команды равен 60.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт “T idle, ms”	
2	Старший байт “T idle, ms”	
3	Младший байт “T iniL, ms”	
4	Старший байт “T iniL, ms”	
5	Младший байт “T iniH, ms”	
6	Старший байт “T iniH, ms”	

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Успешное выполнение команды	1

Пример

Данные запроса

Значение байта (hex)	Описание
02	Запрос ко 2-му устройству (к K-Line)
3c	Номер команды = 60 (Посылка WuP)
06	Младший байт длины данных команды
00	Старший байт длины данных команды
0a	Младший байт “T idle, ms”
00	Старший байт “T idle, ms”
19	Младший байт “T iniL, ms”
00	Старший байт “T iniL, ms”
19	Младший байт “T iniH, ms”
00	Старший байт “T iniH, ms”

Данные ответа

Значение байта (hex)	Описание
02	Ответ от 2-го устройства (от K-Line)
3c	Номер команды = 60 (Посылка WuP)
01	Младший байт длины данных команды
00	Старший байт длины данных команды
69	Младший байт размера приёмного буфера адаптера
db	Старший байт размера приёмного буфера адаптера
01	Успешное выполнение команды

Команды виртуальных устройств “CAN”

Номера виртуальных устройств равны 4 и 5.

Команда “Открытие устройства”

Номер команды равен 101.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт скорости устройства	
2	Средний байт скорости устройства	
3	Средний байт скорости устройства	
4	Старший байт скорости устройства	Baud rate: 10000..1000000
5	Формат фреймов CAN (Base = 0 / Extended = 1)	0..1
6	Состояние терминального резистора 120 Ом (OFF = 0 / ON = 1)	0..1
7	Зарезервировано	0
8	Зарезервировано	0
9	Зарезервировано	0

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Закрытие устройства”

Номер команды равен 102.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Установка состояния линии CAN”

Номер команды равен 103.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Состояние линии (BUS_OFF = 0 / BUS_ON = 1)	0..1

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Отсылка сообщения CAN”

Номер команды равен 104.

Для ускорения передачи данных возможно объединение до 4-х сообщений в один буфер данных для отсылки (число 4 возникает из-за ограничения размера внутреннего буфера LNK-34, равного 64 байтам). Если номер версии прошивки (см. команду “Проверка связи”) меньше 43, тогда нулевой размер данных не допускается, то есть минимальный размер данных может быть 1 байт.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт CAN ID сообщения	
2	Средний байт CAN ID сообщения	
3	Средний байт CAN ID сообщения	
4	Старший байт CAN ID сообщения	CAN ID: 0..FFFFFFFF
5	Длина данных CAN сообщения	0..8 До 43-й версии прошивки допустимый диапазон 1..8
[6..13]	Данные сообщения	

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Расширенная отсылка сообщения CAN”

Номер команды равен 109.

Команда поддерживается начиная с 47-й версии прошивки (см. команду “Проверка связи”). Исполняется аналогично команде 104 за исключением того, что есть возможность указать формат CAN-фрейма конкретно для данной команды. Также возможный максимальный размер поля данных увеличен до 64-х байт (с учётом будущего применения стандарта CAN-FD).

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт CAN ID сообщения	
2	Средний байт CAN ID сообщения	
3	Средний байт CAN ID сообщения	
4	Старший байт CAN ID сообщения	CAN ID: 0..FFFFFFFF
5	Формат фрейма CAN	BIT7=1: Extended Frame
6	Длина данных CAN сообщения	0..64
[7..70]	Данные сообщения	

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание
	Нет данных

Команда “Получение сообщения CAN”

Номер команды равен 105.

Команда не является ответом на команду ПК, она посылается адаптером сразу после получения данных по физической шине.

Данные запроса ‘ПК => Адаптер’

Номер байта	Описание
	Нет данных

Данные ответа ‘Адаптер => ПК’

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт CAN ID сообщения	
2	Средний байт CAN ID сообщения	
3	Средний байт CAN ID сообщения	
4	Старший байт CAN ID сообщения	CAN ID: 0..FFFFFFFF

5	Формат фрейма CAN (BIT7=1: Extended Frame / BIT6=1: RTR-Frame)	
6	Длина данных CAN сообщения	0..8
7	Младший байт CAN Time сообщения	
8	Средний байт CAN Time сообщения	
9	Средний байт CAN Time сообщения	
10	Старший байт CAN Time сообщения	CAN Time: 0..FFFFFFFF
[11..18]	Данные сообщения	

Команда "Получение статуса линии CAN"

Номер команды равен 106.

Данные запроса 'ПК => Адаптер'

Номер байта	Описание
	Нет данных

Данные ответа 'Адаптер => ПК'

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт состояния линии	
2	Старший байт состояния линии	Значение = 0x10 означает состояние линии BUS_OFF

Команда " Установка минимальной задержки между посылкой сообщений CAN "

Номер команды равен 107.

Данная команда поддерживается начиная с 40-й версии прошивки.

Данные запроса 'ПК => Адаптер'

Номер байта	Описание	Допустимый диапазон значений
1	Младший байт значения задержки, в микросекундах	
2	Старший байт значения задержки, в микросекундах	0..65535 микросекунд (0.000001 сек)

Данные ответа 'Адаптер => ПК'

Номер байта	Описание

	Нет данных

Подключение нескольких адаптеров одновременно

К ПК можно одновременно подключить несколько кабелей связи (к разным USB портам). По-умолчанию, все кабели связи имеют одинаковую EEPROM идентификацию, что не позволяет одновременно работать с нескольким кабелями (программно определяться будет только один кабель). Для активизации возможности одновременной работы, необходимо выполнить дополнительное программирование EEPROM с изменённой идентификацией.

Настройки параметров передачи данных по USB

Настройки выполняются при инициализации протокола обмена с кабелем связи и выполняются через функции интерфейса библиотеки FTD2XX DLL.

1. Таймауты по чтению и передаче данных установлены в 100 мс.
2. Таймер задержки установлен в 1 мс (минимальное значение). Такая задержка позволяет оптимизировать передачу данных для протокола АБИТ Логер, что позволяет получить максимально быстрое время отклика для коротких пакетов данных.
3. Размер пакетов передачи для USB запросов установлен в 1024 байта.

Библиотека USBCommCan

Библиотека поставляется в виде архива в формате ZIP. Архив содержит заголовочные файлы и файлы библиотек для “C”/”C++” компилятора. Настройки и ключевые слова компилятора, приводимые далее, специфичны для компилятора Microsoft Visual C++. Однако, использование именно Visual C++ компилятора не является обязательным.

Для использования программного интерфейса необходимо в проект подключить библиотечный файл USBCommCan.lib. В архиве представлены три версии библиотеки USBCommCan.dll - с ведением текстового лог-файла выполняемых действий, без вывода и версия с отладочной информацией. Для активации режима ведения текстового лог-файла выполняемых действий, необходимо скопировать файл USBCommCan.cfg в каталог с исполняемыми файлами (в нем можно задать имя лог-файла), иначе лог сохраняться не будет.

Так же в архиве находятся примеры использования библиотеки для взаимодействия с блоками управления по шинам K-Line и CAN (проекты для компилятора Microsoft Visual C++).

Содержимое архива

- ***include*** – содержит заголовочные файлы библиотеки (.h)
- ***src*** - примеры использования библиотеки USBCommCan для связи с блоками управления (для Microsoft Visual C++)
    - **Comm_Test** – пример использования библиотеки для передачи данных по шине K-Line
    - **CAN_Test** – пример использования библиотеки для передачи данных по шине CAN
    - **AdaptLogger** – пример использования библиотеки для отображения и записи в лог-файл трафика сообщений, передаваемых по шине CAN
    - **examples.sln** – файл проекта для Microsoft Visual Studio, содержит все три примера
- ***.\..\output*** - откомпилированные примеры с соответствующими версиями библиотеки USBCommCan.dll
    - **Debug** – версия библиотеки с отладочной информацией
    - **Release** - версия библиотеки без отладочной информации
    - **ReleaseLog** - версия библиотеки с ведением текстового лог-файла выполняемых действий
- ***readme.txt*** – файл с кратким описанием содержимого архива и последовательности установки драйверов и подключения кабеля связи к ПК

Примечание: Актуальную версию драйверов FTDI можно загрузить с сайта АБИТ http://abit.spb.ru/akm/usb-adaptery/.

Интерфейс библиотеки USBCommCan

Класс LoggerProto

Предназначен для реализации протокола АБИТ Логер, работающего “поверх” интерфейса библиотеки FTD2XX DLL (низкоуровневый интерфейс для передачи данных между USB Host Controller ⇔ FT232 по шине USB). В классе реализовано кодирование и декодирование фреймов данных, контроль целостности, обработку и восстановление после ошибок.

```
/*****************************************************************************\
 FILE.........: LoggerProto.h
 VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef LOGGER_PROTO_H
#define LOGGER_PROTO_H
```

```
// default to 8 byte alignment
#pragma pack(push,8)

class LoggerProto
{
```

Описание интерфейса для приёма сообщений и обработки событий, возникающих при передаче данных. Реализация должна производиться в соответствии с шаблоном проектирования “наблюдатель”. Реализация поддерживает регистрацию произвольного количества “наблюдателей”.

```
enum ListenerType {
	LISTENER_NONE = 0,
	LISTENER_ORDINAL = 1,
	LISTENER_CANBUS = 2,
	LISTENER_TEMPORARY = 3,
};

// Received data callback structure
struct IListener {
	virtual void OnReceiveData(
			int deviceIndex,
			int command,
			const std::string & data
			) const = 0;

	virtual void ListenerEvent(
			ListenerType listenerType,
			int deviceIndex,
			bool isConnectedNotDisconnected
			) const = 0;
};
```

Список команд протокола АБИТ Логер, которые поддерживает данная реализация.

```
// Standart logger device
static const int LOGGER_DEVICE_INDEX = 1;

// Logger commands
static const int LOGGER_CMD_CONNECT = 1;
static const int LOGGER_CMD_PING = 2;

// RS-232 / K-LINE commands
static const int COMM_CMD_OPEN = 51;
static const int COMM_CMD_CLOSE = 52;
static const int COMM_CMD_BAUDRATE = 53;
static const int COMM_CMD_SEND = 54;
static const int COMM_CMD_RECEIVE = 55;
static const int COMM_CMD_BUF_SIZE = 56;
static const int COMM_CMD_BUF_CLEAR = 57;
static const int COMM_CMD_SET_PIN = 58;
static const int COMM_CMD_GET_PIN = 59;
static const int COMM_CMD_SEND_WUP = 60;

// CAN commands
static const int CAN_CMD_OPEN = 101;
static const int CAN_CMD_CLOSE = 102;
static const int CAN_CMD_BUS = 103;
```

```
static const int CAN_CMD_SEND = 104;
static const int CAN_CMD_RECV_DATA = 105;
static const int CAN_CMD_RECV_STATUS = 106;
static const int CAN_CMD_SET_MSG_DELAY = 107;
```

Методы для посылки сообщений. Параметры для первого варианта включают: индекс виртуального устройства, номер команды из списка, приведённого выше, и данные для команды. Второй вариант отличается тем, что позволяет одновременно (в рамках одного пакета данных) передать несколько команд протокола, что позволяет существенно ускорять пакетную передачу данных.

Индекс виртуального устройства является уникальным идентификатором выбранного устройства в рамках выбранного адаптера. Такая индексация связана с тем, что в системе может быть одновременно активными несколько разных кабелей связи, каждый из которых имеет несколько виртуальных устройств (одна шина K-Line и до двух шин CAN).

```
bool sendCommand(
        int deviceIndex,
        int command,
        const std::string & data
        );

bool sendCommands(
        int subDeviceIndex,
        int command,
        const std::vector<std::string> & poolData
        );
```

Метод для получения текущей версии ПО, запрограммированного в кабель связи.

```
int getSoftwareVersion(
        ) const;
```

Класс LoggerProtoFactory

Предназначен для создания экземпляров класса протокола АБИТ Логер. Реализован в соответствии с шаблоном проектирования “Фабрика”. Создаваемые экземпляры протоколов учитываются фабрикой, то есть при запросе повторного создания экземпляра протокола, новый объект не создаётся, а производится увеличения счётчика ссылок на данный созданный экземпляр. Экземпляр удаляется только после уменьшения счётчика ссылок до нуля.

```
/*****************************************************************************\
 FILE.........: LoggerProtoFactory.h
 VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef LOGGER_PROTO_FACTORY_H
#define LOGGER_PROTO_FACTORY_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Параметры для создания экземпляра класса протокола АБИТ Логер включают: индекс виртуального устройства, тип регистрируемого интерфейса для приёма сообщений и обработки событий, экземпляр интерфейса, флаг необходимости установки соединения.

```
LoggerProto * createLoggerProto(
        int deviceIndex,
        LoggerProto::ListenerType listenerType = LoggerProto::LISTENER_NONE,
        const LoggerProto::IListener * pListener = 0,
        bool * isNeedConnect = 0
        );
```

Для удаления экземпляра протокола нужно передать: индекс виртуального устройства и тип интерфейса для приёма сообщений.

```
void removeLoggerProto(
        int deviceIndex,
        LoggerProto::ListenerType listenerType = LoggerProto::LISTENER_NONE
        );
```

Дополнительные сервисные методы позволяют получить конфигурацию активных виртуальных каналов.

Для получения текстового описания виртуального канала нужно передать: индекс виртуального устройства.

```
bool getDeviceLocationId(
        size_t orderIndex,
        std::string & id
        ) const;
```

Для получения описаний всех виртуальных каналов, присутствующих в системе, предназначен следующий метод.

```
void enumAllDevicesByType(
        std::map<int, std::string> & devices,
        LoggerDeviceInfo::DeviceType type,
        bool isGetOnlyNonUsed
        );
```

Для активизации режима “передача и приём пакетов данных с нулевой задержкой”, предназначен следующий метод. При активизации данного режима возможно существенное повышение загрузки CPU системы, поэтому данный режим не рекомендуется держать постоянно включённым.

```
void setBoostMode(
        bool isBoostMode
        );
```

Класс LoggerCommFactory

Предназначен для создания экземпляров виртуальной шины K-Line. Реализован в соответствии с шаблоном проектирования “Фабрика”. Создаваемые экземпляры виртуальной

шины учитываются фабрикой, то есть при запросе повторного создания экземпляра шины, новый объект не создаётся, а производится увеличения счётчика ссылок на данный созданный экземпляр. Экземпляр удаляется только после уменьшения счётчика ссылок до нуля.

```
/*****************************************************************************\
  FILE.........: LoggerCommFactory.h
  VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef LOGGER_COMM_FACTORY_H
#define LOGGER_COMM_FACTORY_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Медоты для создания и удаления экземпляров виртуальной шины K-Line / RS-232.

```
virtual IComm * createComm(
            );

virtual void removeComm(
            IComm * pComm
            );
```

Интерфейс IComm

Класс является реализацией виртуальной шины K-Line / RS-232. Он выполнен в виде интерфейса (содержит чисто виртуальные функции) и решение о выборе реализующего класса принимается классом фабрикой LoggerCommFactory.

```
/*****************************************************************************\
 FILE.........: IComm.h
 VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef ICOMM_H
#define ICOMM_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Для передачи данных по шине K-Line необходимо работать с COM-портом последовательной передачи данных. Для открытия COM-порта используется структура, поля которой определяют все настройки порта.

```
struct CCommSettings
{
	int			m_port;
	CRS232Config::EnumBaud		m_nBaud;
	CRS232Config::EnumParity	m_nParity;
	CRS232Config::EnumDataBit	m_nDataBits;
	CRS232Config::EnumStopBit	m_nStopBits;
	CRS232Config::EnumFlowControl	m_nFlowControl;
	bool			m_bMonoline;

	CCommSettings() : m_port(1), m_nBaud(CRS232Config::E_BAUD_10400),
		m_nParity(CRS232Config::E_PARITY_N), m_nDataBits(CRS232Config::E_DATABITS_8),
		m_nStopBits(CRS232Config::E_STOPBITS_1), m_nFlowControl(CRS232Config::E_FLOWCTRL_ABIT),
		m_bMonoline(true)
	{}
};
```

Методы для открытия и закрытия COM-порта. В случае повторного открытия внутри метода выполняется закрытие предыдущей сессии.

```
virtual bool Open(
	const CCommSettings & commSettings
	) = 0;

virtual void Close(
	) = 0;
```

Метод для переключения текущей скорости COM-порта. Скорость задаётся в бодах, стандартные значения скорости перечислены в заголовочном файле ‘rs232/rs232lib.h’ (110, 300, 600, 1200, 2400, 4800, 9600, 10400, 19200, 38400, 57600, 115200).

```
virtual bool ChangeBaud(
	int iBaud,
	int clearbufFlag = 0
	) = 0;
```

Методы для посылки данных (одного байта и массива байтов) в COM-порт. Если в установках порта было указано использование монолинии, передаваемы байты запоминаются во внутреннем буфере для контроля принимаемых данных.

```
virtual bool SendByte(
        BYTE byteToSend
        ) = 0;

virtual bool SendBuffer(
        const char * byteBuffer,
        size_t iSize,
        DWORD answerWaitTimeout = 0
        ) = 0;
```

Метод для чтения данных из COM-порта. При чтении данных, если в установках порта было указано использование монолинии, принимаемые данных контролируются на совпадение с данными, запомненными во внутреннем буфере.

```
virtual bool ReadByte(
        BYTE  & byteRead,
        DWORD   dwTimeout = SERIAL_READ_TOTAL_TIMEOUT_MS
        ) = 0;
```

Методы для получения информации о загруженности входного и выходного буферов данных для COM-порта.

```
virtual size_t GetInBufferSize(
        ) = 0;

virtual size_t GetOutBufferSize(
        ) = 0;
```

Метод для очистки буферов данных. Обычно используется для очистки данных в случае восстановления после ошибок приёма / передачи данных.

```
virtual bool ClearBuffer(
        int clearbufFlag
        ) = 0;
```

Значение поля ‘clearbufFlag‘ может быть комбинацией битовых флагов

```
enum {
        CLEARBUF_IN = 0x01,
        CLEARBUF_OUT = 0x02,
};
```

Метод для определения текущего состояния COM-порта.

```
virtual bool IsGood(
        ) const  = 0;
```

Методы установки флагов управляющих сигналов для COM-порта (DTR - Готовность приемника данных (Data Terminal Ready), RTS - Запрос на передачу (Request to Send), BREAK – приостанавливает передачу символов для COM-порта и ставит линию передачи в состояние прерывания до тех пор, пока не будет вызвана функция очистки сигнала BREAK).

```
virtual bool SetRTS(
        int iMode
        ) = 0;

virtual bool SetDTR(
        int iMode
        ) = 0;

virtual bool SetBreak(
        int iMode
        ) = 0;
```

Метод для посылки “wake-up” сигнала по шине K-Line (это сигнал длительностью 50 мс используемый для начала коммуникационной сессии).

```
virtual bool sendWakeUpPattern(
        ) = 0;
```

Класс LoggerCanFactory

Предназначен для создания экземпляров виртуальной шины CAN. Реализован в соответствии с шаблоном проектирования “Фабрика”. Создаваемые экземпляры виртуальной шины учитываются фабрикой, то есть при запросе повторного создания экземпляра шины, новый объект не создаётся, а производится увеличения счётчика ссылок на данный созданный экземпляр. Экземпляр удаляется только после уменьшения счётчика ссылок до нуля.

```
/*****************************************************************************\
  FILE.........: LoggerCanFactory.h
  VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef LOGGER_CAN_FACTORY_H
#define LOGGER_CAN_FACTORY_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Медоты для создания и удаления экземпляров виртуальной шины CAN.

```
virtual ICan * createCan(
        );

virtual void removeCan(
        ICan * pCan
        );
```

Интерфейс ICan

Класс является реализацией виртуальной шины CAN. Он выполнен в виде интерфейса (содержит чисто виртуальные функции) и решение о выборе реализующего класса принимается классом фабрикой LoggerCanFactory.

```
/*****************************************************************************\
 FILE.........: ICan.h
 VERSION......: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef ICAN_H
#define ICAN_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Для передачи данных по шине CAN необходимо инициализировать настройки микросхемы CAN-контроллера. Для этого используется структура, поля которой определяют все настройки шины. Кроме настроек собственно CAN-контроллера, есть две настройки для протокола CAN, это ‘ExtendedCanId’ и ‘UseTerminalResistor’. Первая используется для идентификации того какая адресация CAN используется (стандартная 11 бит, либо расширенная 29 бит). Вторая используется для указания того, нужно ли виртуальномы устройству CAN эмулировать терминальный резистор.

```
struct AsapCanSettings
{
	// first type of settings
	int   m_iChannelIndex;
	BYTE  m_btBTR0;
	BYTE  m_btBTR1;

	// second type of settings
	DWORD m_dwBaudRate;
	BYTE  m_btSamplePoint;
	BYTE  m_btSampleRate;
	BYTE  m_btBTLCycles;
	BYTE  m_btSJW;
	BYTE  m_btSyncEdge; // 0 - single, 1 - dual

	bool  m_isExtendedCanId; // false - Normal 11 bit Identifiers, true - Extended 29 bit Identifiers
	bool  m_isUseTerminalResistor; // false - not used, true - use 120KoM terminal resistor
};
```

Описание интерфейса для приёма сообщений и обработки событий, возникающих при передаче данных по шине CAN. Реализация должна производиться в соответствии с шаблоном проектирования “наблюдатель”. Реализация поддерживает регистрацию произвольного количества “наблюдателей”.

```
struct IListener {
	virtual void OnReceiveMessage(
			DWORD dwId,        // CAN Identifier
			BYTE  frameFormat,   // CAN Frame format (BIT7=1: 29BitID / BIT6=1: RTR-Frame)
			BYTE  dataLenght,    // CAN Data Length Code
```

```
                BYTE  data[8],      // CAN Data
                DWORD dwTime        // Time in ms
                ) const = 0;

        virtual void OnReceiveStatus(
                ICanStatus canStatus
                ) const = 0;

        virtual void OnShutDown(
                ) const = 0;
};
```

Метод для открытия виртуального устройства CAN. Параметрами являются настройки шины данных CAN, а также экземпляр с типом интерфейса приёма сообщений и обработки событий. В случае повторного открытия внутри метода выполняется закрытие предыдущей сессии.

```
virtual bool Open(
        const ICan::AsapCanSettings & canSettings,
        const ICan::IListener * pListener
        ) = 0;
```

Метод для закрытия виртуального устройства CAN.

```
virtual void Close(
        const ICan::IListener * pListener
        ) = 0;
```

Метод для очистки буферов данных входящих и исходящих сообщений. Обычно используется для очистки данных в случае восстановления после ошибок приёма / передачи данных.

```
virtual bool ClearBuffers(
        ) = 0;
```

Методы для посылки сообщений (одного сообщения и группы сообщений) по шине данных CAN. Параметрами метода являются: идентификатор сообщения CAN, флаг с указанием стандартной / расширенной идентификации и данные сообщения для передачи.

```
virtual bool SendMsg(
        DWORD           dwId,
        bool            isExtendedCanId,
        const std::string & sData
        ) = 0;

virtual bool SendAllMsg(
        DWORD           dwId,
        bool            isExtendedCanId,
        const std::vector<std::string> & poolMsg
        ) = 0;
```

Метод для установки задержки между отдельными сообщениями при пакетной передаче сообщений. Наличие данного метода обсловлено тем, что некоторые блоки управления требуют наличия обязательной задержки между отдельными сообщениями.

```
virtual bool SetMsgDelay(
        size_t microsec
        ) = 0;
```

Метод для определения текущего состояния шины данных CAN.

```
virtual bool IsInitialized(
        ) const = 0;
```

Интерфейс LoggerCanBus

Класс предоставляет сервисные функции, облегчающие использование виртуальной шины CAN. Для передачи и приёма сообщений используется экземпляр интерфейса ICan.

```
/*****************************************************************************\
  FILE.....: LoggerCanBus.h
  VERSION..: 1.00
\*****************************************************************************/

/*=========================================[ REDEFINITION DEFENCE ]=======*/
#ifndef LoggerCanBus_H
#define LoggerCanBus_H

// default to 8 byte alignment
#pragma pack(push,8)
```

Метод для установления связи и запуска получения статистики передаваемых сообщений по шине CAN.

```
bool startListen(
        const ICan::AsapCanSettings & canSettings
        );
```

Метод для отключения от шины данных CAN.

```
void stopListen(
        );
```

Метод для определения текущего состояния шины данных CAN.

```
bool isListen(
        ) const;
```

Метод для получения текущей статистической информации о сообщениях (учитываются сообщения, передаваемые в обоих направлениях ‘ПК => Кабель связи’ и ‘Кабель связи => ПК’), переданных по шине данных CAN.

```
void getMsgData(
        std::vector<CanMsgInfo> & canMsgInfos
        ) const;
```

Формат описания статистической информации об одном сообщении

```
struct ADAPTDLL_API CanMsgInfo {
	DWORD dwId_;
	DWORD count_;
	DWORD timeDeltaMs_;
};
```

Метод для посылки сообщения по шине данных CAN. Параметрами метода являются: индекс виртуального устройства, идентификатор сообщения CAN и данные сообщения для передачи. Подразумевается использование расширенной адресации CAN.

```
bool sendCanMsg(
	int deviceIndex,
	DWORD dwCanId,
	const std::string & sData
	) ;
```

Примеры использования интерфейса

Ниже представлены описания примеров, которые поставляются вместе с библиотекой и показывают примеры использования интерфейса USBCommCan.

Настройки Visual C++

Для компиляции описываемых примеров используются следующие настройки компиляции

```
cl.exe /nologo /MD /W3 /GX /O2 /I "../../include" /D "WIN32" /D "NDEBUG" /D "_WINDOWS" /D "_AFXDLL" /D "_MBCS"
/Yu"stdafx.h"  /FD /c
```

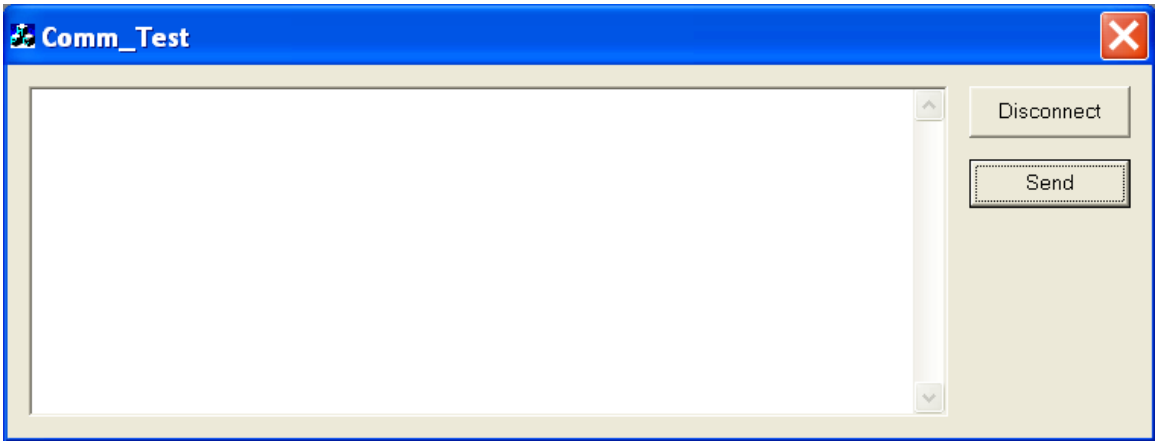
Использование шины K-Line

Исходный код примера, а так же файл проекта находятся в каталоге ‘examples/src_vc60/Comm_Test’. В данном примере при запуске выполняется последовательное создание экземпляров классов LoggerProtoFactory (используется для последующего создания фабрики виртуальных устройств K-Line) , LoggerCommFactory (используется для последующего создания виртуального устройства K-Line) и собственно IComm.

При нажатии на кнопку “Connect” настройки для шины данных K-Line устанавливаются в начальные (скорость 10400 бод) и производится подключение к шине данных. После этого все сообщения, которые приходят по шине данных, выводятся в окно.

При нажатии на кнопку “Test” выполняется посылка данных

```
char buf[] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0};
```



Использование шины CAN

Исходный код примера, а так же файл проекта находятся в каталоге ‘examples/src_vc60/AdaptLogger’. В данном примере при запуске выполняется последовательное создание экземпляров классов LoggerProtoFactory (используется для последующего создания фабрики виртуальных устройств CAN) и LoggerCanBus (используется для получения статистики о сообщениях, передаваемых по шине данных CAN).

Интерфейс примера позволяет выбрать скорость передачи данных по шине CAN (по-умолчанию используется 250 кБит), эмуляцию терминального резистора и использование расширенной адресации CAN.

После нажатия на кнопку “Запустить” выполняется установка связи с шиной данных CAN и затем в окно выводится информация о передаваемых сообщениях.