AnyRAM, «ARD Trade»
Production
«ARD Trade», 1, 2-Priberezhnaya St, Vitebsk, Belarus
Tel: +375-29-5968565
Fax: +375-21-2586250
Website: http://www.anyram.net
E-Mail: client@anyram.net

Описание функций API V1.7 ридера бесконтактных карт RD-03AB для ОС Windows и Linux

V1.7$_1$

Особенности аппаратного интерфейса

Для организации обмена по USB, ридер использует протокол обмена USB HID устройства. Подобный подход позволяет организовать программный интерфейс с ридером без установки специальных драйверов, т.е. программный интерфейс организуется с помощью встроенных в систему функций. Результатом подобного подхода является то, что ридер поддерживается почти платформами и версиями ОС, которые поддерживают работу с USB HID устройствами, например, все версии Windows (XP, W8, W10), все версии Windows CE/RE, Linux (включая embedded-версии и Android), FreeBSD, MacOS и т.п.. В POSIX системах взаимодействие с ридером происходит через сервис, предоставляемый стандартным драйвером ***hidraw*** или библиотекой ***libusb-1.0***, в Windows ридер обслуживается встроенными функциями, как USB HID устройство.

В режиме USB ридер поддерживает автоматическое переключение спецификаций HID устройства: **HID_Interrupt** и **HID_Control**.

Режим **HID_Interrupt** включен в ридере по умолчанию и позволяет организовать непрерывный приём данных через приёмный тред или процесс путём считывания данных из внешнего блокового устройства: для Windows данные в режиме **HID_Interrupt** считываются функцией *ReadFile*, для Linux - *read*. Запись данных в ридер осуществляется путём записи данных во внешнее блоковое устройство: для Windows данные в режиме **HID_Interrupt** записываются функцией *WriteFile*, для Linux - *write*. Если процесс приёма/передачи данных прерывается (например, используется не непрерывный, а периодический вызов *ReadFile*), то данные могут быть утеряны. Для организации обмена данными с ридером путём периодического опроса следует использовать режим **HID_Control**.

Режим **HID_Control** включается после обмена стандартными пакетами **GET_REPORT** или **SET_REPORT:** для Windows это функции *HidD_GetInputReport* и *HidD_SetOutputReport*, для Linux – ***ioctl(…, HIDIOCGFEATURE, …)*** и ***ioctl(…, HIDIOCSFEATURE, …)***.

После активации режима **HID_Control**, ридер накапливает данные для передачи хосту в буфере и выдаёт очередную пачку данных только по стандартному запросу **GET_REPORT**: этим обеспечивается непрерывность потока данных при периодическом опросе ридера. Если периодичность опроса ридера будет недостаточно высокой, то внутренний кэш ридера может переполниться. Объёмы входного и выходного буфера ридера одинаковы - 64 байта.

В случае отсутствия в течение 5с запросов **GET_REPORT** или **SET_REPORT** ридер из режима **HID_Control** перейдёт в режим **HID_Interrupt**. Досрочно перевести ридер в режим **HID_Interrupt**, можно подав команду блоковой записи (для Windows функция *WriteFile*, для Linux - *write*). Низкоуровневые функции ввода-вывода в файле ***rd0xAB.cpp*** взаимодействуют с ридером только в режиме **HID_Control**: режим **HID_Interrupt** в текущей версии API не используется.

Использование описываемого API необязательно: на базе команд ридера может быть разработано своё уникальное API, учитывающее особенности эксплуатации ридера. Полное описание, формат команд и тайминги протокола обмена приводится в описании ридера RD-03AB.

Принципы построения API

Программное взаимодействие с ридером через API представляет собой типовой обмен данными с символьным устройством: открытие хэндла устройства, многократная посылка команды - ожидание ответа, закрытие хэндла устройства. При работе через COM-порт, парадигма символьного обмена реализуется простым вызовом соответствующих функций ОС. Однако, в связи с тем, что USB подключение ридера является блоковым, а задача API – унификация интерфейса, при открытии USB подключения приходится хранить дополнительные параметры обмена, чтобы сделать блочное подключение похожим на символьное: поэтому при открытии USB-хэндла ридера, приходится заводить структуру псевдохэндла, в которой хранятся дополнительные параметры обмена.

Если от программы не требуется развитая работа с данными на бесконтактных картах, то можно не подключать *mf0xAB.cpp* и уменьшить размер получаемого кода, что важно для компактных систем.

Подключение, закрытие подключения

Подключение к ридеру начинается с заполнения структуры псевдохэндла. Для создания псевдохэндла (*typedef CCR*) ридера, подключенного по USB, следует вызывать функцию *link_hidopen*; открытие псевдо-хэндла ридера, подключенного по COM-порту, производится функцией *link_comopen*.

Если открытие ридера было успешным, то заполняется структура CCR.

```
typedef struct {
    HANDLE  hL;     // хэндл устройства, полученный от ОС
    DWORD   txq;    // время следующей передачи данных по USB
    DWORD   rxout;  // таймаут приёма данных, TIMEOUT_RX = 600млс
    DWORD   txout;  // таймаут передачи данных, TIMEOUT_TX = 600млс
    DWORD   wtout;  // таймаут ожидания, TIMEOUT_WT = 2000млс
    PBYTE   inbuf;  // указатель на динамически выделенный буфер входных данных
    BYTE    inlen;  // длина данных во входном буфере, не более HIDBUF_SZ = 32
    PBYTE   outbuf; // указатель на динамически выделенный буфер выходных данных
    BYTE    outlen; // длина данных в выходном буфере, не более HIDBUF_SZ = 32
} CCR, *PCCR;
```

Поле **hL** - это хэндл устройства, полученный от системы. В принципе, им бы можно было и ограничиться, если бы не требовались дополнительные переменные для организации интерфейса с ридером. Хэндл закрытого/неактивного устройства заполняется в ОС Windows значением **INVALID_HANDLE_VALUE** (0xFFFFFFFF), в ОС Linux это значение 0.

Поле **txq** используется только при обслуживании USB подключения. Передача данных по USB начинается при заполнении выходного буфера *outbuf*, или при достижении времени *txq*, в зависимости от того, что наступит раньше. Сделано это для увеличения производительности USB интерфейса, т.к. заниматься передачей отдельных байтов по USB крайне накладно. Значение *txq* в ОС Windows заполняется перед передачей

очередной порции данных, как *txq = GetTickCount() + TXQ_CN*, где **TXQ_CN** = 5, т.е. в качестве опорного используется время от старта системы в миллисекундах. В ОС Linux это значение рассчитывается из данных ф-ции *gettimeofday.*

Поля **rxout**, **txout**, **wtout** - времена таймаутов при приёме, передаче и ожидании по запросу от устройства в миллисекундах. Если по какой-то причине таймауты не устраивают, то после открытия устройства их значения можно изменить.

Поля **inbuf** и **outbuf** - указатели на динамически выделенную область памяти, в которой организованы приёмный буфер и буфер для передачи данных в ридер. Длина каждого буфера **HIDBUF_SZ** = *32* байта. Память выделяется одновременно для входного и выходного буфера функцией *inbuf = GlobalAlloc(GPTR, HIDBUF_SZ * 2)* в ОС Windows, в ОС Linux выделение памяти производится функцией *inbuf = malloc(HIDBUF_SZ * 2)*. Значение указателя на выходной буфер равно половине выделенного блока, т.е. *outbuf = &inbuf[HIDBUF_SZ]*. В освобождении выделенной памяти при закрытии соединения, участвует только поле *inbuf*. Используется только при USB подключении ридера! При подключении ридера по COM-порту или tty-устройству *inbuf = outbuf = NULL*. Значение *inbuf* может использоваться для определения типа подключения: *inbuf != NULL* для USB подключения, *inbuf = NULL* для COM или tty подключения.

В полях *inlen* и *outlen* хранится длина данных, находящихся в приёмном буфере и буфере передачи соответственно. Используется только при USB подключении ридера!

Пример заполнения структуры псевдохэндла CCR при различных подключениях:

при COM-подключении (*link_comopen*)

```
-----------------------------------------------
hL =     0x130;      // хэндл
txq =    0x01DE0064; // не используется
rxout =  600;        // TIMEOUT_RX
txout =  600;        // TIMEOUT_TX
wtout =  2000;       // TIMEOUT_WT
inbuf =  NULL;       // пусто
inlen =  0;          // пусто
outbuf = NULL;       // пусто
outlen = 0;          // пусто
```

при USBподключении (*link_hidopen*)

```
-----------------------------------------------
hL =     0x180;      // хэндл
txq =    0x01DE0123; // время след.передачи
rxout =  600;        // TIMEOUT_RX
txout =  600;        // TIMEOUT_TX
wtout =  2000;       // TIMEOUT_WT
inbuf =  0x00776010; // указатель на буфер
inlen =  0;          // длина данных
outbuf = 0x00776030; // указатель на буфер
outlen = 0;          // длина данных
```

Для API, использующего libusb, хэндл устройства задаётся с помощью union: при работе через tty-устройство, используется хэндл tty-устройства **hL.t**; при работе через USB-интерфейс, используется хэндл, предоставленный libusb **hL.u**. Объявление структуры псевдохэндла для libusb:

```
typedef struct {
    union
    {
        HANDLE                  t;  // tty-хэндл, полученный от ОС
        libusb_device_handle    *u; // USB-хэндл, полученный от libusb
    } hL;                           // объединение хэндлов
    DWORD   txq;    // время следующей передачи данных по USB
    DWORD   rxout;  // таймаут приёма данных, TIMEOUT_RX = 600млс
    DWORD   txout;  // таймаут передачи данных, TIMEOUT_RX = 600млс
    DWORD   wtout;  // таймаут ожидания, TIMEOUT_WT = 2000млс
    PBYTE   inbuf;  // указатель на динамически выделенный буфер входных данных
    BYTE    inlen;  // длина данных во входном буфере, не более HIDBUF_SZ = 32
    PBYTE   outbuf; // указатель на динамически выделенный буфер выходных данных
    BYTE    outlen; // длина данных в выходном буфере, не более HIDBUF_SZ = 32
} CCR, *PCCR;
```

Основное отличие COM-подключения от USB-подключения в том, что *link_comopen* **не проверяет** наличие подключенного ридера к открываемому порту: функция *link_comopen* просто открывает COM-порт. Основным методом проверки подключения ридера является посылка команды **CMG_VER** для получения аппаратной версии ридера и установки протокола обмена. Сразу после получения версии, рекомендуется перевести его в режим **DIR** без записи режима в энергонезависимую память, чтобы избежать рассинхронизации обмена, если ридер настроен на выдачу запросов при прикладывании карты. После этого можно безопасно работать с ридером, не боясь рассинхронизации соединения.

Закрытие подключения к ридеру производится вызовом функции *link_close* с указателем на структуру псевдохэндла: выделенная для буферов память освобождается, хэндл устройства закрывается, при этом в поле хэндла в структуре псевдохэндла устанавливается значение **INVALID_HANDLE_VALUE.**

Взаимодействие с ридером

Взаимодействие с ридером строится на базе всего одной функции - *link_packet*, однако, для удобства реализации “пропускных” режимов, введена ещё одна фунция *link_rqwait*: с помощью этой функции за один вызов можно получить UID, SAK, ATQA и данные, записанные на карте в режимах, отличных от режима прямого управления DIR. Все функции являются блокирующими, т.е. при организации параллельных процессов функции должны быть вынесены в отдельный тред.

link_packet – реализует все возможные типы передачи данных, параметры зависят от команды, описание команд приводится в руководстве. На базе *link_packet* строятся **все** функции API, однако, для того, чтобы взаимодействие с ридером происходило без ошибок, надо учитывать некоторые особенности функционирования ридера.

Например, возможна ситуация, когда ридер находится в режиме SRD, пользователь поднёс карту, и ридер послал запрос на разрешение прохода удалённому компьютеру. Если в этот момент попробовать связаться с ридером, попытка связи завершится неудачей, т.к. ридер ожидает подтверждение прохода по карте (генерируется командой *link_rqack*) и будет игнорировать другие команды. Для того чтобы ридер гарантированно отвечал на все запросы, его следует перевести в режим прямого управления DIR хотя бы на время управления. Следует помнить, что ридер поддерживает 2 установки текущего режима (и бипера): установку режима в энергонезависимой памяти (ROM-режим), и установку временного режима (RAM-режим), который будет сброшен в ROM-режим при выключении и повторной подаче питания.

Вообще, следует стараться как можно реже переписывать ROM-настройки, хранящиеся в энергонезависимой памяти, т.к. ресурс памяти хоть и очень большой, но не безграничный. Типовое начало сессии с ридером выглядит так:

```
// read device description: HW, NVM size, model, SN
if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto main_err;
if (buf[0] < 0x20)
{
    printf(" ERROR: HW version must be >= 2.0!\n");
    goto main_err;
}
// temporary set DIR mode, buzzer on
if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto main_err;
```

Вызов команды CMG_VER позволяет узнать версию и дополнительные данные о ридере: версия оборудования ниже 2.0 не поддерживается. Следующая команда

CMS_MOD переводит ридер в режим прямого управления DIR с включением бипера, но режим этот – временный, и при повторной подаче питания будет установлен ROM-режим, записанный в энергонезависимой памяти. Хорошим тоном является установка временного RAM-режима DIR на время управления ридером с последующим возвратом в ROM-режим перед закрытием подключения.

Если используется библиотека *mf0xAB,* то сессию можно начать, вызвав функцию *mfInit*, которая выполняет вышеописанные операции: считывает версию аппаратного обеспечения, объём памяти и серийный номер ридера, а затем переводит ридер в режим прямого управления DIR с включенным бипером.

```
// DIR-mode initialization
if ((err = mfInit(&ccr, &r_model, &r_hw, &r_sn)) != LERR_SUCCESS) goto main_err;
```

link_rqwait – позволяет организовать простой протокол обмена с внешним решателем/регистратором. Обычно используется в паре с функцией *link_rqack*, которая посылает запрет/подтверждение прохода по карте, но использование *link_rqack* не является обязательным.

Перед использованием *link_rqwait* ридер настраивают на необходимый режим работы (например, с помощью утилиты **ccr2**), режим обмена данными, ключи авторизации, таймаут ответа. Программа, обслуживающая ридер, вызывает функцию *link_rqwait* с установленной длиной принимаемого пакета. Крайне важно правильно установить длину принимаемого пакета, иначе приём данных будет невозможен!

Вызов *link_rqwait* заблокирует текущий поток на время таймаута приёма данных, которое задано в поле rxout в структуре псевдохэндла. После открытия значение поля rxout равно константе TIMEOUT_RX, но это значение может быть изменено при необходимости. Если во время ожидания будут приняты данные, выход из функции будет произведён раньше истечения времени таймаута.

После разбора принятых данных, программа может послать подтверждение с помощью *link_rqack,* в которой указывается один из 3х вариантов реакции: 0 - запрет прохода по карте, 1 – разрешение прохода по карте, 2 – решение о проходе карты принимает ридер. Важно чтобы ответ решателя дошёл до ридера за время таймаута ожидания ответа (устанавливается в ридере), иначе ридер примет решение о пропуске карты самостоятельно. Если программа просто регистрирует пропускаемые карты, то посылать подтверждение не обязательно.

Ошибки

Почти все функции API возвращают код ошибки, описывающий результат работы функции. Код, равный нулю (LERR_SUCCESS), указывает на успешное завершение функции, остальные описывают ошибку, возникшую при выполнении.

```
// “low level” errors
#define LERR_SUCCESS            0  // успешное завершение функции
#define LERR_FAIL               1  // ошибка выполнения команды
#define LERR_HARDWARE           2  // ошибка интерфейса (COM-порта или USB)
#define LERR_TIMEOUT_TX         3  // истёк таймаут передачи
#define LERR_TIMEOUT_RX         4  // истёк таймаут приёма
#define LERR_NOTSYNC            5  // ошибка синхронизации
```

```
// “high level” errors
#define LERR_NOREADER           6  // ридер не найден
#define LERR_NOCARD             7  // нет карты
#define LERR_INVALIDCARD        8  // карта неправильного типа
#define LERR_INVALIDKEY         9  // неправильный ключ
#define LERR_IOERR              10 // ошибка обмена
#define LERR_DATAREAD           11 // ошибка чтения данных
#define LERR_DATAWRITE          12 // ошибка записи данных
#define LERR_BUSY               13 // устройство занято
#define LERR_USER               14 // пользовательский код ошибки
```

Ошибки с кодами от 1 (LERR_FAIL) до 5 (LERR_NOTSYNC) можно условно считать низкоуровневыми ошибками, ошибки с кодами от 6 (LERR_NOREADER) и до 12 (LERR_DATAWRITE) – высокоуровневыми ошибками. Низкоуровневые коды ошибок возвращают функции *rd0xAB*: в основном это ошибки, связанные с обменом данными. Высокоуровневые коды ошибок генерируются функциями *mf0xAB*: в основном это логические ошибки, связанные с обменом данными с картами. Коды LERR_BUSY и LERR_USER API не генерируются, но эти коды могут быть использованы в пользовательском программном обеспечении для указания занятости интерфейса ридера (LERR_BUSY) и формирования своих дополнительных кодов ошибок (LERR_USER).

При аварийном выходе **всегда** следует вызывать функцию закрытия псевдохэндла устройства *link_close* для освобождения хэндла и памяти, занятой под буферы ввода-вывода.

Константы, определения

При использовании API рекомендуется использовать предопределённые константы, а не числовые значения: в случае изменений в следующей версии API это позволит обойтись перекомпилированием кода без изменений в тексте программы. Назначение основных констант и их значения приводится ниже. Подробности использования констант приводятся в описании функций.

rd0xAB.h

```
#define UDEV_REV                0x20  // минимальная версия аппаратного обеспечения ридера

#define SZ_UID                  10    // максимальная длина UID, размер буфера для приёма UID
#define SZ_SCCD                 14    // длина дескриптора карты: uidlen(1)+UID(10)+SAK(1)+ATQA(2)

#define SZ_MFBLK                16    // длина блока при блоковом доступе
#define SZ_MFPAGE               4     // длина страницы

#define MFDM_BLOCK              0x00  // константа блокового режима (b.0 = 0), длина данных 16б
#define MFDM_PAGE               0x01  // константа страничного режима (b.0 = 1), длина данных 4б
#define MFDM_DM                 0x00  // константа режима чтения при чтении/авторизации (b.1 = 0)
#define MFDM_AM                 0x02  // константа режима авторизации при чтении/авторизации (b.1 = 1)
#define MFDM_NMC                0x00  // стандартная карта, операция в стандартном режиме (b.7 == 0)
#define MFDM_RWC                0x80  // RW-карта, операция с картой в RW-режиме (b.7 == 0)
```

mf0xAB.h

```
#define SZ_ATS                  30    // максимальная длина ATS, используется для выделения буфера
```

Типовые приёмы программирования

Структура программы

Полная реализация типовых программных циклов приводится в примерах, идущих в демонстрационном программном обеспечении, общая же концепция построения следующая.

Для режимов ACS, SRD, IBE:

1) перечисление подключенных ридеров (*link_hidopen*, *link_comopen*);
2) подключение к выбранным ридерам (*link_hidopen*, *link_comopen*);
3) установка RAM-режима в DIR (*link_packet*);
4) считывание параметров (*link_packet*);
5) изменение необходимых параметров (*link_packet*);
6) восстановление режима ROM-режима ACS, SRD или IBE (*link_packet*);
7) рабочий цикл, реализуемый с помощью функций *link_rqwait* и *link_rqack*;
8) выход (возможно с переустановкой параметров, (*link_close*)).

Для режима DIR:

1) перечисление подключенных ридеров (*link_hidopen*, *link_comopen*);
2) подключение к выбранному ридеру (*link_hidopen*, *link_comopen*);
3) установка RAM-режима в DIR (*link_packet*);
4) рабочий цикл, обычно реализуемый с помощью функций библиотеки *mf0xAB*;
5) выход с установкой RAM-режима, записанного в ROM (*link_packet*, *link_close*).

Демонстрационный пример ***acs-ibe*** показывает типовую реализацию цикла управления ридером в режимах ACS и IBE с помощью удалённого сервера с учётом обрывов связи.

Пример ***rqv*** показывает типовую реализацию программы, записывающей лог, при работе ридера автономно в режимах ACS, IBE, SRD без чтения данных.

Пример ***srd*** показывает типовые программные циклы управления ридером в режиме SRD с помощью удалённого сервера с учётом обрывов связи с различными типами карт, с авторизацией и без, с чтением и без чтения данных.

Исходники консоли настройки ридеров ***ccr*** показывают все возможные приёмы настройки ридеров.

Исходники ***mfoc*** на примере взлома карт Mifare Classic показывают реализацию *всех* функций режима DIR: лучший пример крайне трудно придумать, в исходниках есть почти все возможные комбинации вызовов высокоуровневых и низкоуровневых функций.

Пример ***scprobe*** демонстрирует начальный уровень низкоуровневого доступа к картам в режиме DIR, в котором пользователь избавлен от необходимости формировать контрольные суммы, биты паритета, декодировать трафик и т.п.. Такой режим работы подходит для большинства стандартных и нестандартный карт.

Пример ***scc*** показывает **типовые** приёмы работы с данными на картах семейства Mifare и Mifare Zero в режиме DIR, используя только высокоуровневые команды.

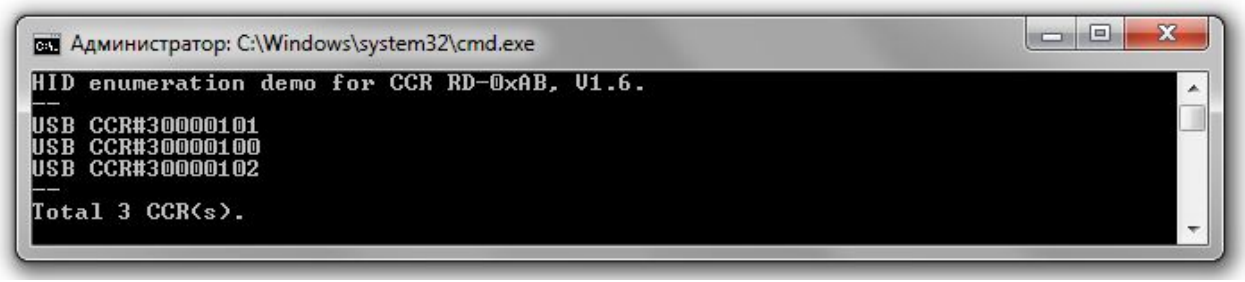
Пример ***c2rw*** демонстрирует **нетиповые** приёмы работы с данными на картах семейства Mifare и Mifare Zero в режиме DIR, используя только высокоуровневые команды.

Перечисление ридеров

При одновременной работе с несколькими ридерами, программе требуется знать, сколько ридеров присутствует в системе, и, в случае необходимости, подключаться к выбранному ридеру. Перечисление ридеров, подключенных по USB производится с помощью функции *link_hidopen*.

```
// *** перечисление ридеров, подключенных по USB ***
#include "rd0xAB.h"
// -- main --
int main(int argc, char *argv[])
{
    DWORD       cnt;        // счётчик ридеров

    printf("HID enumeration demo for CCR RD-0xAB, V1.6.\n--\n");

    for(cnt = 0;; cnt++)
    {
        DWORD   usn = 0;    // usn принимает серийный номер ридера, который выводится на экран
        if ((link_hidopen(cnt, &usn, NULL)) != LERR_SUCCESS) break;
        printf("USB CCR#%08X\n", usn);
    }
    printf("--\nTotal %u CCR(s).\n", cnt);

    return 0;
}
```

Приведённый код универсален: он одинаков как для Windows, так и для Linux. Результатом работы будет выведенный на экран список ридеров в системе.



Очевидно, что при перечислении можно выяснить серийные номера подключенных ридеров, и затем подключиться к ридеру с заданным серийным номером, а не подключаться в порядке, возвращаемом системой. Функция *link_hidopen* позволяет подключаться к ридеру с заданным серийным номером, что может быть использовано при привязке ридера к конкретному объекту: например, когда известно, что ридер #30000101 установлен на входном турникете, а ридер #30000102 установлен на выходном турникете, а ридер #30000100 в системе пропуска не используется.

Произвести перечисление ридеров, подключенных по COM-порту (UART или RS-485), значительно сложнее: в начале надо перечислить COM-порты, а затем проверить, подключены ли к этим портам ридеры, подав какую-либо команду с заранее известным ответом.

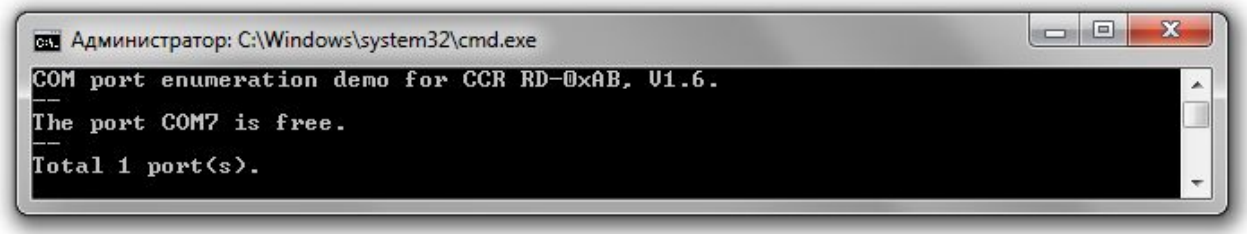
В ОС Windows перечисление последовательных портов производится попыткой открыть порт, исходя из того, что стандартное имя последовательного порта **\\.\COM**n, где n – номер порта от 1 до 255. Универсального способа перечисления портов под ОС Linux нет, однако, можно попробовать подключаться к символьным устройствам в каталоге **/dev**, если в имени устройства присутствует аббревиатура **tty**, например, *ttyS0, ttyS1, ttyUSB0, ttyACM* и т.д. В силу больших затрат времени на выяснение подключен ли к последовательному порту ридер или нет (таймаут ответа 0.6с), обычно ограничиваются перечислением портов, а наличие/отсутствие подключенного к нему ридера выясняют в

процессе установки связи с ридером. Специфика реализации COM-портов также подразумевает, что заданный ридер, например, ридер входного турникета, всегда подключен к одному и тому же COM-порту, а не “гуляет” по разным портам, поэтому при подключении ридера по последовательному порту можно ограничиться указанием номера COM-порта (Windows) или имени последовательного порта (Linux), не привязываясь к серийному номеру ридера.

Перечисление ридеров, подключенных к COM-порту под ОС Windows, может быть выполнено с помощью простого цикла перебора портов.

```
// *** перечисление COM-портов Windows ***
#include "rd0xAB.h"
// -- main --
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    DWORD       cnt;    // счётчик количества портов в системе
    DWORD       idx;    // номер порта при переборе

    _tprintf(TEXT("COM port enumeration demo for CCR RD-0xAB, V1.6.\n--\n"));

    for(cnt = 0, idx = 1; idx < 256; idx++)
    {
        // следует указывать реальную скорость подключения, т.к.
        // порт может не поддерживать другие скорости
        if ((link_comopen(idx, CBR_9600, NULL)) != LERR_SUCCESS) continue;
        _tprintf(TEXT("The port COM%u is free.\n"), idx);
        cnt++;
    }
    _tprintf(TEXT("--\nTotal %u port(s).\n"), cnt);

    return 0;
}
```

Результатом работы будет выведенный на экран список свободных COM-портов.



Установка скорости последовательного порта

Сама по себе установка скорости обмена BR (Baud Rate) не вызывает сложности, если ридер подключен к USB-интерфейсу. Соответствие параметра команды **CMS_UBR** реальной скорости обмена определяется таблично: 0 – 1200бод, 1 – 2400бод, 2 – 4800 – бод, 3 – 9600бод, 4 – 14400бод, 5 – 19200бод, 6 – 38400бод, 7 – 56000бод, 8 – 57600бод, 9 – 115200бод.

```
// установка скорости BR = 9600бод
if ((err = link_packet(&ccr, CMS_UBR, 3, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
```

Однако, если ридер подключен к последовательному порту, то после выполнения команды **CMS_UBR** (команда отработает успешно, т.к. подтверждение выполнения **специально** даётся на старой скорости), ридер станет недоступным, т.к. ридер перешёл на новую скорость, а интерфейс остался на старой. Проблема может быть решена 2мя способами: 1) переустановкой скорости последовательного порта средствами ОС без закрытия хэндла устройства, 2) переподключением, с указанием новой скорости обмена.

Для изменения скорости 1м способом достаточно знать только хэндл последовательного порта, для изменения скорости 2м способом помимо псевдохэндла требуется знать номер порта (в Windows) или имя устройства (в Linux). Несмотря на наличие дополнительных параметров и изменение хэндла в процессе работы, 2ой способ изменения скорости часто оказывается удобнее 1го.

1) Изменение скорости последовательного порта “на ходу” в ОС Windows.

```
BOOL change_baudrate(HANDLE hCOM, DWORD cbr)
    DCB         dcb;
    if (!GetCommState(hCOM, &dcb)) return FALSE;    // выход при фатальной ошибке
    dcb.BaudRate = cbr;                             // установка нового BR
    if (!SetCommState(hCOM, &dcb)) return FALSE;    // установка нового режима порта
    return TRUE;
}
```

Изменение скорости последовательного порта “на ходу” в ОС Linux.

```
bool change_baudrate(int hTTY, uint32_t cbr)
{
    uint32_t    defBR[8] = { 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 };
    uint32_t    defBCN[8] = { B1200, B2400, B4800, B9600, B19200, B38400, B57600, B115200 };

    struct  termios         dcb;
    int                     i;

    // конвертируем пользовательскую скорость в “стандартную” для Linux
    for (i = 0; i < sizeof(defBR) / sizeof(defBR[0]); i++)
        if (defBR[i] == cbr) break;
    if (i < sizeof(defBR) / sizeof(defBR[0])) i = defBCN[i]; // ура, стандартный BR найден
    else i = 0;

    memset(&dcb, 0, sizeof(dcb));   // очистка структуры
    if (tcgetattr(hTTY, &dcb) < 0) return false;
    cfsetospeed(&dcb, (i) ? i : B38400);    // установка скорости передачи
    cfsetispeed(&dcb, (i) ? i : B38400);    // установка скорости приёма
    if (tcsetattr(hTTY, TCSAFLUSH, &dcb) < 0) return false;
    if (i == 0) // если выбрана нестандартная скорость, производим дополнительные действия
    {
        // установка пользовательской скорости обмена
        struct  termios2    tio;

        if (ioctl(hTTY, TCGETS2, &tio) < 0) return false;
        tio.c_cflag &= ~CBAUD;
        tio.c_cflag |= BOTHER;
        tio.c_ispeed = cbr;
        tio.c_ospeed = cbr;
        if (ioctl(hTTY, TCSETS2, &tio) < 0) return false;
    }
    return true;
}
```

2) Изменение скорости последовательного порта с переподключением в ОС Windows.

```
BOOL change_baudrate(DWORD port, DWORD cbr, PCCR pccr)
{
    link_close(pccr);       // закрытие старого подключения
    // переподключение по COM-порту с новым BR
    if ((err = link_comopen(port, cbr, pccr)) != LERR_SUCCESS) return FALSE;
    return true;
}
```

Изменение скорости последовательного порта с переподключением в ОС Linux.

```
bool change_baudrate(char *portnm, uint32_t cbr, PCCR pccr)
{
    link_close(pccr);       // закрытие старого подключения
    // переподключение по COM-порту с новым BR
    if ((err = link_comopen(portnm, (DWORD)cbr, pccr)) != LERR_SUCCESS) return false;
    return true;
}
```

Что необходимо знать об установке режима ридера

Мест хранения рабочих режимов устройства у ридера два: сохранённый в энергонезависимой памяти (ROM) и текущий режим (RAM). **Ридер всегда находится в рабочем режиме, который указан в RAM.** При подаче питания (после сброса), ROM-режим ридера копируется в RAM, т.е. ROM-режим – это режим, в который устанавливается ридер при включении; сохранение рабочего режима в ROM необходимо для автономной работы ридера.

При настольном использовании ридера, в ROM можно записать режим прямого управления ридером **DIR**, чтобы ридер не “попискивал” при поднесении карт без запущенного управляющего софта.

В целом, следует руководствоваться правилом: чем реже меняем настройки в энергонезависимой памяти – тем лучше, т.к. несмотря на большой ресурс flash-памяти, он всё-таки ограничен. Второй причиной является уязвимость ридера к помехам по цепям питания в момент записи энергонезависимой памяти: несмотря на высокую скорость записи (~50млс), теоретически возможна ситуация, когда помеха может повредить данные. Сбой при записи не повреждает ридер, но может потребоваться повторная перезапись параметров.

В процессе работы обычно возникает ситуация, когда значения режимов устройства в ROM и RAM различны. Например, при настройке ридера требуется, чтобы ридер находился в режиме DIR, но при этом в ROM находился основной режим ридера. Установка различных режимов достигается последовательной подачей двух команд CMS_MOD, в начале для ROM, чтобы установить основной режим работы (обычно ACS, SRD или IBE), а затем для RAM, чтобы восстановить текущий режим ридера (обычно DIR).

```
// установка ROM-режима ACS с включенным бипером: 0 – код ACS, бит 3 (+8) управляет звуковым сигналом
if ((err = link_packet(&ccr, CMS_MOD, (0 + 8) | 0x80, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
// восстанавливаем в RAM режим DIR (код 3), чтобы можно было управлять ридером
if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
```

При завершении работы программы принято восстанавливать режим работы ридера на режим, записанный в ROM без перезаписи энергонезависимой памяти.

```
if ((err = link_packet(&ccr, CMG_MOD, 0x80, NULL, 0, &dm, 1)) != LERR_SUCCESS) goto ferr_exit;
if ((err = link_packet(&ccr, CMS_MOD, dm & 0x7F, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
```

Управление выводом запорного устройства (замка)

Для правильного функционирования запорного устройства следует правильно установить флаг “polar” в командах **CMG_OPT**/**CMS_OPT**. Назначение битов 0 – прямая полярность, а 1 – инверсная – достаточно условно, т.к. в различных исполнениях ридера и конструкциях запорного устройства понятие “прямой” полярности может меняться на противоположное. Поэтому при установке полярности, правильным следует считать то значение флага, при котором запорное устройство открыто при светящемся жёлтом индикаторе.

В режимах ACS и SRD ридер самостоятельно управляет запорным устройством, но при необходимости, можно заблокировать состояние запорного устройства в открытом или закрытом положении с помощью команды **CMF_SW**, при этом ридер игнорирует работу с картами (за исключением мастер-карты в режиме ACS). При выключении и повторном включении ридера или переходе в другой режим, принудительная блокировка запорного устройства сбрасывается: ридер продолжает работать в нормальном режиме. В режимах IBE и DIR запорное устройство по умолчанию находится в закрытом состоянии – возможно только принудительное управление.

Несмотря на то, что с битовой точки зрения параметры команды управления запорным устройством вполне логичны (2 управляющих бита), при программировании реальных функций возникает два разных значения с одинаковой реакцией ридера: значения 0 и 1 разблокируют принудительное управление запорным устройством, значение 2 принудительно закрывает запорное устройство, значение 3 принудительно открывает запорное устройство (светится жёлтый индикатор). Маленькая хитрость заключается в переводе управляющих значений 0(неактивно), 1(закрыто), 2(открыто) в диапазон параметров команды **CMF_SW**: для этого достаточно к управляющему значению прибавить 1.

```
lock = 2; // 0 – управление замком неактивно, 1 – дверь закрыта, 2 – дверь открыта
if ((err = link_packet(&ccr, CMF_SW, lock + 1, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
```

Операции с ACS-ключами

ACS-ключ представляет собой запись длиной 8 байт, идентифицирующую UID карты. Запись состоит из байта длины UID и 7ми байт UID. Возможны следующие длины UID: 4 байта, 7 байт и, теоретически, 10 байт (предусмотрено стандартом, но в реальных картах такой UID не встречается). Если длина UID недостаточна для заполнения записи ACS-ключа (4 байта), запись дополняется нулями. Если длина UID превышает длину ACS-ключа (10 байт), последние байты UID отбрасываются. ACS-ключ, у которого все байты равны 0xFF, считается пустым (нет записи о ключе, ключ удалён).

Формат ACS-ключа

байт 0	**байт 1**	**байт 2**	**байт 3**	**байт 4**	**байт 5**	**байт 6**	**байт 7**
длина UID	UID0	UID1	UID2	UID3	UID4	UID5	UID6

Пример ACS-ключа для UID длиной 4 байта AC18E431

байт 0	**байт 1**	**байт 2**	**байт 3**	**байт 4**	**байт 5**	**байт 6**	**байт 7**
0x04	0xAC	0x18	0xE4	0x31	0x00	0x00	0x00

Пример ACS-ключа для UID длиной 7 байт 043A01171ECA80

байт 0	**байт 1**	**байт 2**	**байт 3**	**байт 4**	**байт 5**	**байт 6**	**байт 7**
0x07	0x04	0x3A	0x01	0x17	0x1E	0xCA	0x80

Пример ACS-ключа для UID длиной 10 байт 041FA50294D3675B1843

байт 0	**байт 1**	**байт 2**	**байт 3**	**байт 4**	**байт 5**	**байт 6**	**байт 7**
0x0A	0x04	0x1F	0xA5	0x02	0x94	0xD3	0x67

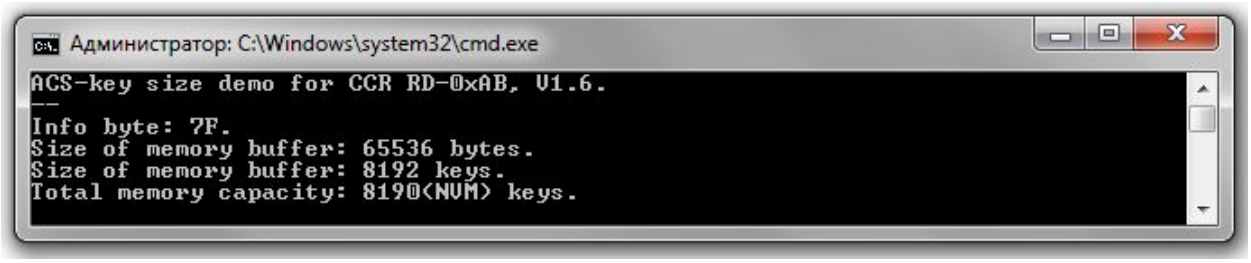
В зависимости от исполнения ридера, ACS-ключи могут храниться как во внутренней памяти контроллера - ROM (Read Only Memory), так и во внешней энергонезависимой памяти – NVM (Non-Volatile Memory). В ROM помещается около 700 ключей, количество ключей в NVM зависит от типа установленной микросхемы: в NVM может храниться от 511 (24C32) до 8191(24C512) ключей, включая мастер-ключ. Вообще-то, в NVM количество записей о ключах кратно 64, т.е. реальный объём микросхемы 24C512 в ключах равен 8192 записям, но последняя запись NVM содержит сигнатуру: по сигнатуре контроллер определяет, что NVM отформатирована и готова к использованию. Объём памяти ACS-ключей содержится во втором байте (смещение +1), возвращаемом командой **CMG_VER**. Мастер-ключ хранится по смещению 0 на странице 0.

Пример вычисления объёмов памяти, связанных с ACS-ключами.

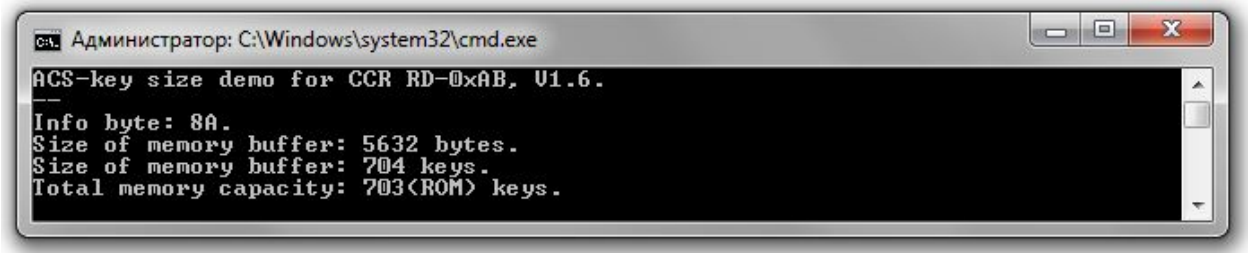
```
// ACS-key size demo, V1.6, Windows
#include "rd0xAB.h"

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        buf[6];

    BYTE        romi;   // ROM info byte: b.7 – тип памяти, b.6...b.0 – последняя страница памяти
    UINT        memsz;  // объём рабочего буфера для считывания всех страниц, в байтах
    UINT        memszk; // объём рабочего буфера для считывания всех страниц, в ключах
    UINT        total;  // количество ключей, которое можно хранить в ридере (без мастер-ключа)
```

```
    _tprintf(TEXT("ACS-key size demo for CCR RD-0xAB, V1.6.\n--\n"));

    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
    // чтение информации о ридере
    if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto ferr_exit;
    romi = buf[1];  // запоминаем информационный байт
    _tprintf(TEXT("Info byte: %02X.\n"), romi);

    // выделяем биты 6..0 с номером последней доступной страницы,
    // затем инкремент кол-ва страниц, т.к отсчёт начинается с 0,
    // затем умножение на 512, т.к. это размер страницы с ACS-ключами;
    // в memsz получаем объём буфера, который который надо выделить в
    // памяти при считывании всех ACS-ключей из ридера
    memsz = ((UINT)(romi & 0x7F) + 1) * 512;
    _tprintf(TEXT("Size of memory buffer: %u bytes.\n"), memsz);

    // в memszk полный объём буфера памяти в ключах (длина ACS-ключа 8 байт)
    memszk = memsz / 8;
    _tprintf(TEXT("Size of memory buffer: %u keys.\n"), memszk);

    // вычисляем количество ключей которое может поместиться в памяти,
    // мастер-ключ за ключ не считается
    total = memszk - 1;

    // при хранении ACS-ключей во внешней памяти (NVM) последняя запись
    // используется для хранения сигнатуры, т.е. ключом не является
    if ((romi & 0x80) == 0) total--;    // учитываем, что в NVM на один ключ меньше

    _tprintf(TEXT("Total memory capacity: %u(%s) keys.\n"),
        total,
        (romi & 0x80) ? TEXT("ROM") : TEXT("NVM"));
ferr_exit:
    link_close(&ccr);
    if (err) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Результат работы программы для ридера с ROM-памятью для хранения ключей.



Результат работы программы для ридера с NVM-памятью для хранения ключей.



Необходимость обслуживания ACS-ключей возникает достаточно редко, поэтому в API отсутствуют специальные функции для работы с ACS ключами: все потребности при операциях с ключами обычно удовлетворяются консолью для настройки ридеров ***ccr***. В случае необходимости, реализация таких функций разбивается на две задачи: чтение ACS-ключей и запись ACS-ключей.

Читать и записывать ACS-ключи можно только постранично с помощью команд **CMG_KM** / **CMS_KM**, т.е. по 64 ключа (8байт * 64ключа = 512байт); операции только с одним ключом невозможны. Для того чтобы поменять один ключ, надо считать страницу с 64 ключами целиком, поменять значение требуемого ключа и записать страницу обратно. Чтобы не заниматься динамическим выделением памяти, можно сразу статически выделить буфер объёмом 64Кбайта (65536 байт) – это максимальный объём памяти ACS-ключей, который может быть в ридере.

Считывать ACS-ключи можно классическим способом, считывая все страницы (время чтения всех страниц из внешней памяти объёмом 64Кбайта по USB ~9с), и быстрым способом, считав все страницы до первой пустой: в ридере есть функция, которая возвращает последнюю занятую страницу (**CMF_KME** с параметром 0), незанятые страницы можно не считывать, т.к. они заполнены байтом 0xFF.

Пример считывания данных классическим способом, чтением всех страниц.

```
// ACS-key slow read demo, V1.6, Windows
#include "rd0xAB.h"

#pragma pack(1)
typedef struct {
BYTE        b[8]; } ACSK;   // структура для хранения ACS ключа
#pragma pack()

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        buf[6];     // буфер для приёма конфигурации ридера и его SN
    ACSK        acsk[8192]; // буфер для приёма ACS-ключей
    ACSK        emptyk;     // пустой ключ
    BYTE        romi;       // ROM info byte: b.7 – тип памяти, b.6...b.0 – последняя страница памяти
    UINT        cnt, memszk;

    _tprintf(TEXT("ACS-key read demo for CCR RD-0xAB, V1.6.\n--\n"));

    memset(&emptyk, 0xFF, sizeof(emptyk));  // инициализация пустого ключа

    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;

    // чтение информации о ридере
    if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto ferr_exit;

    romi = buf[1];  // запоминаем информационный байт
    for (cnt = 0; cnt <= (UINT)(romi & 0x7F); cnt++) // цикл чтения страниц с ACS-ключами
        if ((err = link_packet(&ccr, CMG_KM, (BYTE)cnt,
            NULL, 0, acsk[cnt * 64].b, 512)) != LERR_SUCCESS) goto ferr_exit;

    // вычисляем номер последнего ключа (это также объём памяти в ключах) для вывода ключей на экран
    memszk = (romi & 0x80) ? (UINT)(romi & 0x7F) * 64 - 1 : (UINT)(romi & 0x7F) * 64 - 2;
    for (cnt = 0; cnt <= memszk; cnt++)
    {
        UINT    i;
        // выводим на экран мастер-ключ и непустые ключи
        if ((cnt == 0) || (memcmp(acsk[cnt].b, emptyk.b, 8)))
        {
            if (cnt == 0) _tprintf(TEXT("MAST: "));
            else _tprintf(TEXT("%4u: "), cnt);
            for( i = 0; i != 8; i++)    // вывод значения ACS-ключа
                _tprintf(TEXT("%02X"), acsk[cnt].b[i]);
            _tprintf(TEXT("\n"));
        }
    }
ferr_exit:
    link_close(&ccr);
    if (err) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Пример считывания данных быстрым способом.

```
// ACS-key fast read demo, V1.6, Windows
#include "rd0xAB.h"

#pragma pack(1)
typedef struct {
BYTE        b[8]; } ACSK;   // структура для хранения ACS ключа
#pragma pack()

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        buf[6];     // буфер для приёма конфигурации ридера и его SN
    ACSK        acsk[8192]; // буфер для приёма ACS-ключей
    ACSK        emptyk;     // пустой ключ
    BYTE        romi;       // ROM info byte: b.7 – тип памяти, b.6...b.0 – последняя страница памяти
    BYTE        lstpg;      // последняя занятая ключами страница памяти
    UINT        cnt, memszk;

    _tprintf(TEXT("ACS-key read demo for CCR RD-0xAB, V1.6.\n--\n"));

    memset(&emptyk, 0xFF, sizeof(emptyk));  // инициализация пустого ключа

    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;

    // чтение информации о ридере
    if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto ferr_exit;
    romi = buf[1];  // запоминаем информационный байт

    // чтение последней занятой ключами страницы (не путать с последней страницей памяти!)
    if ((err = link_packet(&ccr, CMF_KME, 0, NULL, 0, &lstpg, 1)) != LERR_SUCCESS) goto ferr_exit;
    // обязательная(!) инициализация массива ключей, т.к. при неполном чтении заполняется не весь массив
    memset(&acsk, 0xFF, sizeof(acsk));

    for (cnt = 0; cnt <= (UINT)lstpg; cnt++)    // цикл чтения страниц с ACS-ключами
        if ((err = link_packet(&ccr, CMG_KM, (BYTE)cnt,
            NULL, 0, acsk[cnt * 64].b, 512)) != LERR_SUCCESS) goto ferr_exit;

    // вычисляем номер последнего ключа (это также объём памяти в ключах) для вывода ключей на экран
    memszk = (romi & 0x80) ? (UINT)(romi & 0x7F) * 64 - 1 : (UINT)(romi & 0x7F) * 64 - 2;
    for (cnt = 0; cnt <= memszk; cnt++)
    {
        UINT    i;
        // выводим на экран мастер-ключ и непустые ключи
        if ((cnt == 0) || (memcmp(acsk[cnt].b, emptyk.b, 8)))
        {
            if (cnt == 0) _tprintf(TEXT("MAST: "));
            else _tprintf(TEXT("%4u: "), cnt);
            for( i = 0; i != 8; i++)    // вывод значения ACS-ключа
                _tprintf(TEXT("%02X"), acsk[cnt].b[i]);
            _tprintf(TEXT("\n"));
        }
    }
ferr_exit:
    link_close(&ccr);
    if (err) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Цветом в обоих примерах выделена часть программы, отвечающая за чтение страниц с ACS-ключами. Видно, что при быстром чтении, чтение ведётся до последней занятой страницы, а не до конца памяти, что значительно сокращает время чтения.

Записывать ACS-ключи можно классическим способом, записывая все страницы подряд, и быстрым способом, выполнив полное стирание памяти, а затем записав только занятые страницы. Следует учитывать, что **при полном стирании памяти мастер-ключ не стирается**, т.е. при желании удалить мастер-ключ, чтобы запись новых ключей была возможна только с помощью компьютера, дать команду стирания или сброса настроек недостаточно: необходимо записать страницу 0 с пустым или новым мастер-ключом. **При записи последней страницы памяти NVM следует сохранить сигнатуру NVM**, иначе при следующем включении ридер отформатирует энергонезависимую память.

Стирание встроенной памяти ROM и NVM небольших объёмов можно производить, просто подав команду стирания **CMF_KME**. При стирании NVM больших объёмов, команда будет возвращать ошибки потери связи, т.к. полное стирание NVM может занимать до 10с. Поэтому универсальный программный код должен подавать команду стирания и, в случае возникновения ошибки, пинговать ридер, ожидая установления связи с ним. После установления связи, программа может продолжить работу.

Пример универсального кода стирания всех страниц ACS-ключей.

```
// посылка команды стирания памяти ACS-ключей
if ((err = link_packet(&ccr, CMF_KME, 0xA5, NULL, 0, NULL, 0)) == LERR_HARDWARE) goto ferr_exit;
if (err != LERR_SUCCESS)        // если была ошибка, то значит идёт "длинное" стирание
{
    // производим 32 попытки пинга ридера
    for (cnt = 0; cnt != 32; cnt++)
    {
        // цикл пинга: посылаем символ пинга IO_CHPING, ждём эха, затем меняем сивол, и так 8 раз;
        // пинг считается успешным, когда все 8 символов прошли без ошибок
        for (i = 0; i != PINGLEN_CN; i++)
        {
            if ((err = link_echobyte(&ccr, (BYTE)(IO_CHPING + i))) == LERR_HARDWARE) goto ferr_exit;
            if (err != LERR_SUCCESS) break;
        }
        if (err == LERR_SUCCESS) break;
        Sleep(150);
    }
}
if (err != LERR_SUCCESS) goto ferr_exit;
// здесь продолжаем выполнение кода программы
```

Сигнатуру NVM можно узнать при чтении ACS-ключей – это последняя запись на последней странице: “AnyRAM\x00\x0**N**”, где N – число, указывающее объём памяти (тип) микросхемы. Допустимы пять значений N: 1 – тип NVM 24C32, объём 4Кб; 2 – тип NVM 24C64, объём 8Кб; 3 – тип NVM 24C128, объём 16Кб; 4 – тип NVM 24C256, объём 32Кб; 5 – тип NVM 24C512, объём 64Кб.

Если программа обладает ограниченным количеством ресурсов, например, на платформе Arduino, где в контроллере каждый байт памяти на счету, сформировать сигнатуру можно “вручную”, зная последнюю страницу NVM.

Пример формирования сигнатуры для записи последней страницы NVM.

```
BYTE        nvmsign[8] = { 'A', 'n', 'y', 'R', 'A', 'M', 0, 0 };
BYTE        nsz, tb = romi & 0x7F; // romi – последняя страница памяти (смещение +1 из команды CMG_VER)
// сдвигаем значение последней страницы, пока в бите 3 не появится 0
for (nsz = 1; nsz != 5; nsz++, tb >>= 1)
    if ((tb & 8) == 0) break;
nvmsign[7] = nsz;
memcpy(acsk[(romi & 0x7F) << 6) | 63].b, nvmsign, 8);
```

Для ускорения работы ридера и процесса записи ключей, рекомендуется уплотнять ключи: удалять повторяющиеся ключи, убирать чередование рабочих ключей с пустыми ключами, смещать рабочие ключи к началу памяти, т.к. проверка на наличие ключа в памяти производится перебором ключей, начиная с младших адресов.

Пример записи данных классическим способом, записью всех страниц.

```
// ACS-key slow write demo, V1.6, Windows
#include "rd0xAB.h"

#pragma pack(1)
typedef struct {
BYTE        b[8]; } ACSK;   // структура для хранения ACS ключа
#pragma pack()

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        buf[6];     // буфер для приёма конфигурации ридера и его SN
    ACSK        acsk[8192]; // буфер ACS-ключей
    BYTE        romi;       // ROM info byte: b.7 – тип памяти, b.6...b.0 – последняя страница памяти
    BYTE        lstpg;      // последняя страница памяти
    UINT        cnt, memszk;

    _tprintf(TEXT("ACS-key slow write demo for CCR RD-0xAB, V1.6.\n--\n"));

    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
    // чтение информации о ридере
    if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto ferr_exit;
    romi = buf[1];  // запоминаем информационный байт
    // стираем все ключи в памяти
    memset(acsk, 0xFF, sizeof(acsk));
    // устанавливаем новый мастер-ключ: UID=11223344, ACSK=0411223344000000
    memcpy(acsk, "\x04\x11\x22\x33\x44\x00\x00\x00", 8);
    // вычисляем номер последнего ключа
    memszk = ((romi & 0x7F) + 1) * 64 - 1;
    if ((romi & 0x80) == 0) memszk--;       // - 1 для внешней NVM
    // вычисляем номер последней страницы
    lstpg = (BYTE)(memszk >> 6);
    // запись всех страниц памяти
    for (cnt = 0; cnt <= lstpg; cnt++)
    {
        // проверка записи последней страницы NVM: формируем сигнатуру, если условие выполнено
        if (((romi & 0x80) == 0) && (cnt == (UINT)(romi & 0x7F)))
        {
            BYTE        nvmsign[8] = { 'A', 'n', 'y', 'R', 'A', 'M', 0, 0 };
            BYTE        nsz, tb = romi & 0x7F;
            // сдвигаем значение последней страницы, пока в бите 3 не появится 0
            for (nsz = 1; nsz != 5; nsz++, tb >>= 1) if ((tb & 8) == 0) break;
            nvmsign[7] = nsz;
            memcpy(acsk[(cnt << 6) | 63].b, nvmsign, 8);
        }
        _tprintf(TEXT("Page %u writing... "), cnt);
        // write ACS page
        if ((err = link_packet(&ccr, CMS_KM, (BYTE)cnt,
            acsk[cnt << 6].b, 512, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
        _tprintf(TEXT("Ok.\n"));
    }
ferr_exit:
    link_close(&ccr);
    if (err) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Пример записи данных быстрым способом.

```
// ACS-key fast write demo, V1.6, Windows
#include "rd0xAB.h"
#pragma pack(1)
typedef struct {
BYTE        b[8]; } ACSK;   // структура для хранения ACS ключа
#pragma pack()

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        buf[6];     // буфер для приёма конфигурации ридера и его SN
    ACSK        acsk[8192]; // буфер для приёма ACS-ключей
    ACSK        emptyk;     // пустой ключ
    BYTE        romi;       // ROM info byte: b.7 – тип памяти, b.6...b.0 – последняя страница памяти
    BYTE        lstpg;      // последняя занятая ключами страница памяти
    UINT        cnt, memszk;

    _tprintf(TEXT("ACS-key fast write demo for CCR RD-0xAB, V1.6.\n--\n"));
    memset(&emptyk, 0xFF, sizeof(emptyk));  // инициализация пустого ключа

    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = link_packet(&ccr, CMS_MOD, 3 + 8, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
    // чтение информации о ридере
    if ((err = link_packet(&ccr, CMG_VER, 0, NULL, 0, buf, 6)) != LERR_SUCCESS) goto ferr_exit;
    romi = buf[1];  // запоминаем информационный байт
    // стираем все ключи в памяти
    memset(acsk, 0xFF, sizeof(acsk));
    // устанавливаем новый мастер-ключ: UID=11223344, ACSK=0411223344000000
    memcpy(acsk, "\x04\x11\x22\x33\x44\x00\x00\x00", 8);

    // вычисляем номер последнего ключа в памяти ридера (объём памяти в ключах)
    memszk = ((romi & 0x7F) + 1) * 64 - 1;
    if ((romi & 0x80) == 0) memszk--;       // - 1 для внешней NVM
    // ищем последнюю занятую страницу по первому непустому ключу, не включая мастер-ключ
    for (cnt = memszk; cnt != 0; cnt--)     // обратный поиск непустого ключа
        if (memcmp(acsk[cnt].b, emptyk.b, 8)) break;
    lstpg = (BYTE)(cnt >> 6);               // вычисление страницы, на которой хранится ключ
    // * стирание памяти ACS-ключей *
    if ((err = link_packet(&ccr, CMF_KME, 0xA5, NULL, 0, NULL, 0)) == LERR_HARDWARE) goto ferr_exit;
    if (err != LERR_SUCCESS)    // если была ошибка, то значит идёт "длинное" стирание
    {
        // производим 32 попытки пинга ридера
        for (cnt = 0; cnt != 32; cnt++)
        {
            UINT        i;
            for (i = 0; i != PINGLEN_CN; i++)
            {
                if ((err = link_echobyte(&ccr, (BYTE)(IO_CHPING + i))) == LERR_HARDWARE) goto ferr_exit;
                if (err != LERR_SUCCESS) break;
            }
            if (err == LERR_SUCCESS) break;
            Sleep(150);
        }
    }
    if (err != LERR_SUCCESS) goto ferr_exit;
    // * запись только занятых страниц памяти до первой свободной *
    for (cnt = 0; cnt <= lstpg; cnt++)
    {
        if (((romi & 0x80) == 0) && (cnt == (UINT)(romi & 0x7F)))
        {
            BYTE        nvmsign[8] = { 'A', 'n', 'y', 'R', 'A', 'M', 0, 0 };
            BYTE        nsz, tb = romi & 0x7F;
            // сдвигаем значение последней страницы, пока в бите 3 не появится 0
            for (nsz = 1; nsz != 5; nsz++, tb >>= 1) if ((tb & 8) == 0) break;
            nvmsign[7] = nsz;
            memcpy(acsk[(cnt << 6) | 63].b, nvmsign, 8);
        }
        _tprintf(TEXT("Page %u writing... "), cnt);
        // write ACS page
        if ((err = link_packet(&ccr, CMS_KM, (BYTE)cnt,
            acsk[cnt << 6].b, 512, NULL, 0)) != LERR_SUCCESS) goto ferr_exit;
        _tprintf(TEXT("Ok.\n"));
    }
```

```
ferr_exit:
    link_close(&ccr);
    if (err) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Цветом в обоих примерах выделена часть программы, отвечающая за запись страниц с ACS-ключами. Видно, что при быстрой записи, запись ведётся до первой свободной страницы, а не до конца памяти, что значительно сокращает время. Несмотря на то, что размер кода для записи ACS-ключей быстрым способом больше, запись в большинстве случаев происходит значительно быстрее. В приведённых примерах запись классическим способом 128 страниц занимает около 3х минут, запись тех же данных быстрым способом занимает 16с, т.е. преимущество быстрого алгоритма записи налицо.

Разумеется, когда память ридера близка к заполнению, быстрые алгоритмы не будут давать преимущества в скорости работы с памятью, однако, при неполной занятости памяти, т.е. в большинстве случаев, быстрые алгоритмы очень эффективны.

Чтение и запись данных на карту

Основными операциями с бесконтактными картами являются чтение и запись данных, и, несмотря на то, что в спецификации Mifare Classic дополнительно описаны команды PICC_TRANSFER, PICC_DECREMENT, PICC_INCREMENT, PICC_RESTORE, найти реальные приложения, использующие эти команды, крайне сложно. Ридер позволяет использовать все описанные и неописанные в спецификации команды, но для дополнительного удобства есть специальные команды ридера для чтения и записи данных: **CMS_CBLK**, **CMG_CBLK** для блоковых операций (длина блока 16 байт), и их дублёры **CMS_CPGE**, **CMG_CPGE** для страничных операций (длина страницы 4 байта). Карты, использующие авторизацию, обычно, работают в блоковом режиме (Security Level 1, Security Level 2, сокращённо SL1, SL2); карты, не использующие авторизацию (SL0), работают в страничном режиме, хотя это “правило” довольно часто нарушается. Например, блоковыми функциями следует пользоваться при работе с картами Mifare Classic, Mifare Mini, Mifare Plus, а страничными функциями следует пользоваться при работе с картами Mifare Ultralight, Mifare Ultralight C.

Для того, чтобы не плодить четыре отдельных функции, в API реализованы две функции обёртки: *mfDataRead* и *mfDataWrite*. Они имеют большое число параметров, которые в простых приложениях можно не использовать, однако, при использовании параметров, функции *mfDataRead* и *mfDataWrite* позволяют реализовать сложные алгоритмы путём простой подстановки значений без написания дополнительного кода. Функции являются блокирующими, т.е. выход из них происходит по истечении таймаута или при успешном чтении/записи.

Пример чтения данных из карты Mifare Classic/Plus/Mini.

```
// CL/PL read demo, V1.6, Windows
#include "mf0xAB.h"

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        ak[6] = { 0xA0, 0xA1, 0xA2, 0xA3, 0xA4, 0xA5 };
    BYTE        acm;    // команда авторизации, обычно PICC_AUTHENT1A или PICC_AUTHENT1B
    BYTE        blk = 0;// номер начального блока для чтения
    BYTE        blklen = 4; // длина в блоках
    BYTE        buf[64];// буфер для 4х блоков: 64 = 4 * 16

    _tprintf(TEXT("CL/PL read demo for CCR RD-0xAB, V1.6.\n--\n"));
    // открытие первого попавшегося ридера по USB
    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = mfInit(&ccr, NULL, NULL, NULL)) != LERR_SUCCESS) goto ferr_exit;
    // установка в RAM значения ключа авторизации (AK0)
    if ((err = mfSetAK(&ccr, 0, ak)) != LERR_SUCCESS) goto ferr_exit;
    // установка команды авторизации по ключу A
    acm = PICC_AUTHENT1A;   // для ключа B следует использовать команду PICC_AUTHENT1B
    for(;;)
    {
        // чтение данных из карты Mifare Classic (CL) или Mifare Plus (PL)
        err = mfDataRead(&ccr, 7, MFDM_NMC | MFDM_BLOCK,
            acm, 0, blk, blklen, buf, NULL, NULL, NULL, NULL);
        if ((err == LERR_SUCCESS) ||
            (err == LERR_HARDWARE) ||
            (err == LERR_NOREADER)) break;
    }
    // вывод данных на экран
    if (err == LERR_SUCCESS)
    {
        UINT        cnt;
        _tprintf(TEXT("Block %u..%u\n--\n"), blk, blk + blklen - 1);
```

```
        for (cnt = 0; cnt != 64; cnt++)
        {
            if ((cnt % 16) == 0) _tprintf(TEXT("%02X: "), cnt);
            _tprintf(TEXT("%02X"), buf[cnt]);
            if ((cnt % 16) == 15) _tprintf(TEXT("\n"));
        }
    }
ferr_exit:
    link_close(&ccr);
    if (err != LERR_SUCCESS) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Пример чтения данных из карты Mifare Ultralight.

```
// UL read demo, V1.6, Windows
#include "mf0xAB.h"

// *** main ***
int __cdecl _tmain(int argc, TCHAR *argv[], TCHAR *envp[])
{
    CCR         ccr;
    DWORD       err = LERR_SUCCESS;
    BYTE        pg = 0; // номер начального блока для чтения
    BYTE        pglen = 16; // длина в блоках
    BYTE        buf[64];// буфер для 16ти страниц: 64 = 16 * 4

    _tprintf(TEXT("UL read demo for CCR RD-0xAB, V1.6.\n--\n"));
    // открытие первого попавшегося ридера по USB
    if ((link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto ferr_exit;
    // установка временного режима DIR с включенным бипером
    if ((err = mfInit(&ccr, NULL, NULL, NULL)) != LERR_SUCCESS) goto ferr_exit;
    for(;;)
    {
        // чтение данных из карты Mifare Ultralight
        err = mfDataRead(&ccr, 7, MFDM_NMC | MFDM_PAGE,
            0, 0, pg, pglen, buf, NULL, NULL, NULL, NULL);
        if ((err == LERR_SUCCESS) ||
            (err == LERR_HARDWARE) ||
            (err == LERR_NOREADER)) break;
    }
    // вывод данных на экран
    if (err == LERR_SUCCESS)
    {
        UINT        cnt;
        _tprintf(TEXT("Page %u..%u\n--\n"), pg, pg + pglen - 1);
        for (cnt = 0; cnt != 64; cnt++)
        {
            if ((cnt % 16) == 0) _tprintf(TEXT("%02X: "), cnt);
            _tprintf(TEXT("%02X"), buf[cnt]);
            if ((cnt % 16) == 15) _tprintf(TEXT("\n"));
        }
    }
ferr_exit:
    link_close(&ccr);
    if (err != LERR_SUCCESS) _tprintf(TEXT("Error: code %u!\n"), err);
    return 0;
}
```

Параметры для функции *mfDataWrite* аналогичны параметрам функции *mfDataRead*, но данные из буфера будут записаны на карту, а не считаны. **После записи карт Mifare Ultralight требуется обязательная верификация данных**, т.к. запись в закрытые страницы (lock pages) и OTP зону может быть невозможна, а установить это факт можно только верификацией записанных данных.

Фиксация уровня PHD, для чего это надо

Уровень PHD, уровень фазового детектора, PHase Detector level – это уровень напряжения на компараторе приёмника, который используется для приёма данных, передаваемых бесконтактной картой. Слабое потокосцепление антенны передатчика и антенны карты крайне затрудняет детектирование замыкания-размыкания контура антенны карты и требуется чрезвычайно плотный контакт двух антенн, чтобы детектировать данные таким способом. Поэтому все ридеры бесконтактных карт независимо от фирмы производителя детектируют не изменение амплитуды в передающем контуре, а быстрое изменение фазы сигнала относительно фазы задающего генератора. Но и здесь не всё просто.

Оптимальный уровень детектирования PHD зависит от типа карты, разности резонансных частот контуров карты и ридера, расстояния между картой и ридером, положения карты и т.п. Неправильно выбранный уровень фазового детектора, неважно какой, низкий, средний или высокий, ведёт к потере приёма данных – уровень должен быть оптимальным, и поэтому уровень устанавливается в процессе соединения с картой. Алгоритм выбора уровня фазового детектора в режиме “авто” приблизительно такой: при подсоединении к карте, ридер устанавливает минимальный уровень PHD, затем подаёт команду PICC_REQxxx для активации карты. Карта посылает ответ ATQA, но если ридер не распознаёт ответ, уровень PHD инкрементируется, а затем снова посылается команда PICC_REQxxx для активации карты, и так по кругу. Если ридер вдруг опознает правильный ответ карты, инкрементирование уровня PHD останавливается до окончания обмена или ошибки.

Поиск оптимального уровня PHD требует времени и, при выполнении большого числа операций обмена данными с картой, потери времени на поиск правильного уровня фазового детектора становятся заметными. Поэтому во многих программах практикуется фиксация уровня PHD после перевода карты на уровень SL0 (Security Level 0), например, с помощью функции выбора карты *mfSelect*.

Реализация фиксации уровня PHD в программе mfoc.

```
// установка уровня PHD в режим авто
if ((err = link_packet(pccr, CMS_ISO, 0x00, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto error;
// активация карты и перевод её в режим SL0
if ((err = mfSelect(&ccr, 7, PICC_REQALL, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto error;
// чтение текущего уровня PHD
if ((err = link_packet(&ccr, CMG_ISO, 0, NULL, 0, isobuf, 2)) != LERR_SUCCESS) goto error;
// фиксация уровня PHD на считанном уровне
if ((err = link_packet(&ccr, CMS_ISO, isobuf[1] | 0x20, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto error;
_ftprintf(stdout, TEXT("Complete. Fix PHD level at %u.\n"), isobuf[1] & 0x1F);
```

Карту специально активируют функцией *mfSelect*, чтобы зафиксировать уровень фазового детектора; в дальнейшем эта функция ни для чего не используется, с её помощью просто узнали оптимальный уровень фазового детектора и зафиксировали детектор на этом уровне. Разумеется, что менять положение карты после фиксации уровня фазового детектора нежелательно.

Описание функций API

link_hidopen (rd0xAB)

Функция открытия ридера, подключенного по USB:

```
DWORD link_hidopen(DWORD idx, PDWORD pusn, PCCR pccr);
```

Входными параметрами являются **idx** (обязательный параметр), указатель на серийный номер ридера **pusn** (необязательный параметр) и указатель на структуру псевдо-хэндла **pccr** (необязательный параметр).

Параметр **idx** указывает номер ридера при перечислении USB-ридеров. Например, если подключено несколько ридеров, то первый найденный ридер будет иметь индекс 0, второй - индекс 1, третий - индекс 2 и т.д.. Если при увеличении индекса функция дала ошибку, значит "свободные" ридеры в системе закончились; это именно это свойство используется для перечисления доступных ридеров в системе. Для открытия первого попавшегося ридера используется **idx = 0** (примеры ниже).

Указатель на серийный номер pusn не является обязательным параметром. Если параметр не задан, то **pusn = NULL**. Если параметр задан, то начинает играть роль значение по указателю **pusn**: если по указателю находится нулевое значение, то поиск и открытие ридера производится по параметру **idx**, а по указателю **pusn** заносится значение серийного номера открытого ридера. Если по указателю **pusn** находится ненулевое значение, оно интерпретируется как SN ридера, который требуется открыть; если при перечислении ридеров, ридер с указанным SN не найден, то *link_hidopen* вернёт ошибку. Т.е. функция *link_hidopen* позволяет открывать ридер с заданным SN, что крайне полезно при проектировании системы с подключением нескольких ридеров.

Указатель на структуру псевдо-хэндла **pccr** не является обязательным параметром. Если параметр не задан (**pccr = NULL**), то открытие хэндла и выделение памяти под буферы не происходит; **pccr = NULL** обычно используется при перечислении ридеров в системе. Если параметр **pccr** указывает на структуру псевдо-хэндла, то при успешном открытии ридера будет инициализирована структура **CCR** по указателю **pccr.**

Функция *link_hidopen* может возвращать следующие коды ошибок: **LERR_SUCCESS**, **LERR_HARDWARE**.

Пример открытия первого попавшегося ридера в системе ридера, подключенного по USB:

```
CCR     ccr;
if ((err = link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto lbl_fail;
```

Пример перечисления всех ридеров, подключенных по USB:

```
for (DWORD cnt = 0;; cnt++)
{
    DWORD   usn = 0;
    if ((link_hidopen(cnt, &usn, NULL)) != LERR_SUCCESS) break;
    printf("USB CCR#%08X\n", usn);
}
```

Пример открытия ридера с заданным серийным номером:

```
CCR     ccr;
DWORD   usn = 0x30000100; // SN
if ((link_hidopen(0, &usn, &ccr)) != LERR_SUCCESS) break;
```

link_comopen (rd0xAB)

Функция открытия ридера, подключенного по COM-порту:

```
DWORD link_comopen(DWORD port, DWORD cbr, PCCR pccr);   // для ОС Windows
DWORD link_comopen(char *portnm, DWORD cbr, PCCR pccr); // для ОС Linux
```

Входными параметрами являются **port** и **portnm** (обязательный параметр), **cbr** и указатель на структуру псевдо-хэндла **pccr** (необязательный параметр).

Параметр port указывает номер COM-порта, который следует открыть (1..255); параметр portnm указывает имя устройства, которое следует открыть (обычно **/dev/tty#** или **/dev/ttyUSB#**).

Параметр **cbr** задаёт скорость обмена данными через COM-порт. При использовании COM-подключения под ОС Linux желательно использовать стандартные значения скоростей 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200, в противном случае могут возникнуть проблемы установки скорости обмена; это зависит от конкретной сборки Linux.

Указатель на структуру псевдо-хэндла **pccr** не является обязательным параметром. Если параметр не задан (**pccr = NULL**), то открытие хэндла и выделение памяти под буферы не происходит; значение **pccr = NULL** можно использовать для проверки наличия/занятости порта в системе. Если параметр **pccr** указывает на структуру псевдо-хэндла, то при успешном открытии ридера будет инициализирована структура **CCR** по указателю **pccr**.

Функция *link_comopen* может возвращать следующие коды ошибок:
LERR_SUCCESS, **LERR_HARDWARE**.

ВНИМАНИЕ! Функция *link_comopen* не проверяет наличие подключенного ридера, она просто открывает COM-порт и подготавливает его к обмену данными.

Пример открытия ридера под ОС Windows, порт COM12, 9600 бод:

```
CCR     ccr;
if ((link_comopen(12, CBR_9600, &ccr)) != LERR_SUCCESS) goto lbl_fail;
```

Пример открытия ридера под ОС Linux, устройство /dev/ttyUSB0, 9600 бод:

```
CCR     ccr;
if ((link_comopen("/dev/ttyUSB0", 9600, &ccr)) != LERR_SUCCESS) goto lbl_fail;
```

Пример проверки наличия 12го COM-порта для ридера под ОС Windows:

```
if ((link_comopen(12, CBR_115200, NULL)) != LERR_SUCCESS) goto lbl_fail;
```

Пример проверки наличия tty-устройства для ридера под ОС Linux:

```
if ((link_comopen("/dev/ttyUSB0", 115200, NULL)) != LERR_SUCCESS) goto lbl_fail;
```

link_close (rd0xAB)

Функция закрытия ридера:

```
void link_close(PCCR pccr);
```

Обязательный параметр pccr указывает на структуру псевдо-хэндла. При закрытии проверяется валидность хэндла (под ОС Windows на равенство **INVALID_HANDLE_VALUE**, под ОС Linux на равенство 0), если хэндл принимает верное значение, то хэндл устройства закрывается, затем освобождается память буферов, если она была выделена. При работе через библиотеку libusb, возврат к предыдущему драйверу, даже если он был ранее отключен, не происходит: это бессмысленно, т.к. связь с ридером и в следующем подключении будет производиться через libusb, а подключение/отключение kernel-драйверов занимает катастрофически много времени.

link_isopen (rd0xAB)

Функция закрытия ридера:

```
BOOL link_isopen(PCCR pccr);
```

Обязательный параметр pccr указывает на структуру псевдо-хэндла. Проверяется валидность хэндлов (под ОС Windows на равенство 0 и **INVALID_HANDLE_VALUE**, под ОС Linux на равенство 0), если хэндл принимает валидное значение, то функция возвращает TRUE, иначе - FALSE.

link_flush (rd0xAB)

Функция очистки очередей:

```
DWORD link_flush(PCCR pccr);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла. Вызывает функции очистки USB и COM/tty очередей, после вызова обнуляет длины данных буферов в структуре псевдо-хэндла. В случае ошибки функций очистки очередей возвращает **LERR_HARDWARE**.

Функция *link_flush* может возвращать следующие коды ошибок:
LERR_SUCCESS, **LERR_HARDWARE**.

link_echobyte (rd0xAB)

Функция передачи байта с ожиданием эха:

```
DWORD link_echobyte(PCCR pccr, BYTE dt);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

В параметре **dt** указывается значение передаваемого байта. В случае ошибки приёма или передачи, функция вернёт ошибку **LERR_TIMEOUT_RX** или **LERR_TIMEOUT_TX**. Если после передачи байта, принятое значение (эхо) не совпало с переданным, функция вернёт ошибку **LERR_NOTSYNC**.

Функция *link_echobyte* может возвращать следующие коды ошибок:
LERR_SUCCESS, **LERR_HARDWARE**, **LERR_TIMEOUT_RX**, **LERR_TIMEOUT_TX**, **LERR_NOTSYNC**

link_rqwait (rd0xAB)

Функция предназначена для приёма запросов на пропуск/блокирование карты.

```
DWORD link_rqwait(PCCR pccr, PBYTE buf, BYTE len);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

buf указывает на буфер для приёма данных: размер буфера должен быть не менее размера принимаемых данных.

len - точная длина принимаемого пакета. Если передаются только параметры карты без данных, len = SZ_SCCD = 14 байт. Параметры карты SCCD - это длина UID с флагом RW карты, 1 байт; UID длиной 10 байт, SAK - 1 байт, ATQA - 1 байт. Если активен режим SRD с передачей данных, то к длине SCCD следут добавить длину передаваемых данных. Подробности проиводятся в демонстрационных примерах acs-ibe, rqv, srd.

Функция *link_rqwait* может возвращать следующие коды ошибок:
LERR_SUCCESS, **LERR_HARDWARE**, **LERR_NOTSYNC**, **LERR_TIMEOUT_RX**

link_rqack (rd0xAB)

Функция предназначена для передачи ответа на запрос, принятый по команде link_rqwait.

```
DWORD link_rqack(PCCR pccr, BYTE ack);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **ack** может принимать следующие значения: 0 - блокирование прохода, 1 - разрешение прохода, 2 - активация реакции ридера по умолчанию.

Функция link_rqack может возвращать следующие коды ошибок:
LERR_SUCCESS, LERR_HARDWARE, LERR_NOTSYNC, LERR_TIMEOUT_RX, LERR_TIMEOUT_TX

link_packet (rd0xAB)

Главная функция ввода-вывода.

```
DWORD link_packet(PCCR pccr, BYTE cmd, BYTE parm,
       PBYTE outbuf, WORD outlen, PBYTE inbuf, WORD inlen);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

В **cmd** указывается код команды, передаваемой ридеру; обязательный параметр.

В **parm** указывается параметр команды; обязательный параметр.

В необязательном параметре **outbuf** устанавливается указатель на выходной буфер данных, которые будут переданы в ридер. Если outbuf = NULL, то команда будет сформирована без исходящего кадра данных.

В необязательном параметре **outlen** указывается длина данных в выходном буфере. Если outlen = 0, то команда будет сформирована без исходящего кадра данных.

В необязательном параметре **inbuf** устанавливается указатель на входной буфер данных, которые будут считаны из ридера. Если inbuf = NULL, то команда будет сформирована без входящего кадра данных.

В необязательном параметре **inlen** указывается длина принимаемых данных. Если inlen = 0, то команда будет сформирована без входящего кадра данных.

Пример использования *link_packet*

```
// открытие ридера, подключенного по USB; возвращает коды LERR_SUCCESS, LERR_HARDWARE
if ((err = link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS) goto main_exit;
// посылка команды подачи звукового сигнала
if ((err = link_packet(pccr, CMF_BUZ, 2, NULL, 0, NULL, 0)) != LERR_SUCCESS) goto main_exit;
// закрытие подключения
link_close(&ccr);
```

acq2type (mf0xAB)

Возвращает тип карты по длине UID, ATQA, SAK, длине ответа ATS и значению ATS.

```
BYTE acq2type(DWORD acq, BYTE atslen, PBYTE ats);
```

Обязательный параметр **acq** формируется из длины UID, ATQA, SAK.

Структура acq

биты 31...24 acq	биты 23...16 acq	биты 15…8 acq	биты 7…0 acq
4 ≤ uidlen ≤ 10	биты 15…8 ATQA	биты 7…0 ATQA	SAK

```
acq = (((DWORD)uidlen) << 24) | (((DWORD)atqa) << 8) | ((DWORD)sak;
```

Необязательные параметры **atslen** и **ats** позволяют передать в функцию дополнительные данные для опознавания типа карты: например, *Mifare Plus SL1 4K* успешно имитирует *Mifare Classic 4K*, но возвращает правильный ATS, позволяющий определить действительный тип карты. Максимальная длина ATS определяется константой **SZ_ATS** = 30 (*mf0xAB.h*). Если ATS не используется при опознавании типа, следует указать **atslen** = 0, **ats** = NULL.

Функция *acq2type* определяет все типы карт Mifare, но может быть дополнена для определения карт любого другого типа.

```
#define SCD_UNKNOWN             0     // тип карты неизвестен
#define SCD_UL                  0x01  // Mifare Ultralight
#define SCD_CL1K                0x02  // Mifare Classic 1K
#define SCD_CL4K                0x03  // Mifare Classic 4K
#define SCD_CL1K1               0x04  // новые Mifare Classic 1K, длина UID 7 байт
#define SCD_CL4K1               0x05  // новые Mifare Classic 4K, длина UID 7 байт
#define SCD_PL2S1               0x06..// Mifare Plus 2K, Security Level 1
#define SCD_PL4S1               0x07  // Mifare Plus 4K, Security Level 1
#define SCD_PL2S2               0x08  // Mifare Plus 2K, Security Level 2
#define SCD_PL4S2               0x09  // Mifare Plus 4K, Security Level 2
#define SCD_PL2S3               0x0A  // Mifare Plus 2K, Security Level 3
#define SCD_PL4S3               0x0B  // Mifare Plus 4K, Security Level 3
#define SCD_MINI                0x0C  // Mifare Classic Mini
#define SCD_DF1                 0x0D  // Mifare DesFire, EV1
#define SCD_DF2                 0x0E  // Mifare DesFire, EV2
#define SCD_SMMX                0x0F  // Mifare SmartMX (эмуляция SL1)
#define SCD_SM1K                0x10  // Mifare SmartMX, эмуляция Classic 1K
#define SCD_SM2K                0x11  // Mifare SmartMX, эмуляция Classic 2K
#define SCD_SM4K                0x12  // Mifare SmartMX, эмуляция Classic 4K
```

Пример использования *acq2type*

```
// установка значения переменных для проверки функции
sc_uidlen = 4;      // длина UID – 46айта
sc_atqa = 0x0002;   // ответ карты при активации
sc_sak = 0x18;      // ответ карты после завершения процедуры выбора
// определение типа карты
sc_type = acq2type((((DWORD)sc_uidlen) << 24) | (((DWORD)sc_atqa) << 8) | ((DWORD)sc_sak);
// после вызова acq2type sc_type принимает значение SCD_CL4K
```

mfInit (mf0xAB)

Инициализирует временный DIR-режим (RAM) с включенным бипером; возвращает тип ридера, версию аппаратного обеспечения, серийный номер ридера. Также *mfInit* включает несущую и переводит фазовый детектор в режим авто, а после инициализации подаёт звуковой сигнал “***good*** ” (звуковой код 2).

```
DWORD mfInit(PCCR pccr, PBYTE pmodel, PBYTE phw, PDWORD psn);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Необязательный параметр **pmodel** указывает на байт, принимающий тип ридера 1…0x0F. Значение **2** соответствует типу ридера **RD-02AB**, значение **3** соответствует типу ридера **RD-03AB**, и т.д.. Если параметр не используется, значение **pmodel** = NULL.

Необязательный параметр **phw** указывает на байт, принимающий версию аппаратного обеспечения ридера, значение **0x20** соответствует версии **2.0**, значение **0x21** соответствует версии **2.1**, и т.д.. Минимальное значение версии для нормальной работы API указывается в константе **UDEV_REV** = 0x20.

Необязательный параметр **psn** указывает на 4 байта, принимающие серийный номер ридера в BCD формате: номеру ридера 30001234 соответствует значение 0x30001234, шестнадцатеричные цифры A, B, C, D, E, F – недопустимы. Серийный номер ридера не может быть равным нулю. Старшая тетрада серийного номера указывает модель ридера: значение именно этой тетрады записывается по указателю **pmodel.**

Пример использования *mfInit*

```
// открытие ридера, подключенного по USB; возвращает коды LERR_SUCCESS, LERR_HARDWARE
if ((err = link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS)
{
    err = LERR_NOREADER;  // подменяем код LERR_HARDWARE кодом отсутствия ридера
    goto main_exit;
}
// инициализация режима прямого управления DIR
if ((err = mfInit(&ccr, &r_model, &r_hw, &r_sn)) != LERR_SUCCESS) goto main_exit;
// вывод данных о подключенном ридере
printf("RD-0%uAB, SN:%08X, HW:%u.%u\n--\n", r_model, r_sn, r_hw >> 4, r_hw & 0x0F);
// закрытие подключения
link_close(&ccr);
```

mfBeep (mf0xAB)

Подаёт звуковой сигнал заданного тона, всего 6 значений тона (см. описание команды **CMF_BUZ**).

```
DWORD mfBeep(PCCR pccr, BYTE tone);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **tone** задаёт тип воспроизводимого звука:

0 – **check**, звук, используемый при успешном завершении операции;
1 – **fail** - звук, используемый при ошибочном завершении операции;
2 – **good** - короткий звук при успешном завершении операции;
3 – **short_fail** - короткий звук при ошибочном завершении операции;
4 – **trill**, трель, используется для обозначения важных событий;
5 – **wow**, нарастающий звук сирены, используется при ошибке важной операции.

Пример использования *mfBeep*

```
// открытие ридера, подключенного по USB; возвращает коды LERR_SUCCESS, LERR_HARDWARE
if ((err = link_hidopen(0, NULL, &ccr)) != LERR_SUCCESS)
{
    err = LERR_NOREADER;  // подменяем код LERR_HARDWARE кодом отсутствия ридера
    goto main_exit;
}
// инициализация режима прямого управления DIR
if ((err = mfInit(&ccr, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;

// включение сирены WOW
if ((err = mfBeep(&ccr, 0x05)) != LERR_SUCCESS) goto main_exit;

// закрытие подключения
link_close(&ccr);
```

mfBrkCarrier (mf0xAB)

Выключает несущую на заданное время. Функцию нельзя использовать в качестве функции реального времени, например, для пропуска заданного количества тактов несущей: эта функция предназначена для временного снятия электропитания, например, для разблокирования карты Mifare Plus при блокировке при подборе ключа.

```
DWORD mfBrkCarrier(PCCR pccr, WORD time_ms);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **time_ms** устанавливает время, на которое будет выключена несущая, в миллисекундах.

mfType (mf0xAB)

Определяет тип карты, RW-свойства, а также возвращает параметры, связанные с определением типа карты, UID, ATQA, SAK, ATS.

```
DWORD mfType(PCCR pccr, BYTE timeout, PBYTE ptype, PBYTE puidlen,
      PBYTE puid, PWORD patqa, PBYTE psak, PBYTE patslen, PBYTE pats);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **timeout** устанавливает время, в течение которого ридер будет пытаться связаться с картой для определения её параметров, допустимы значения 0…7. Время ожидания вычисляется по формуле **time** = (**timeout** * 4 + 3) * 50млс (см. описание команды **CMF_SEL**); минимальное время – 0.15с, максимальное – 1.55с.

Необязательный параметр **ptype** указывает на байт, принимающий значение типа карты. Тип карты определяется с помощью функции **acq2type** по длине UID, ATQA, SAK и ATS. Если ATS не запрашивается (**patslen** = 0 и **pats** = NULL), то опознавание типа производится без участия ATS, и такое опознавание может быть неточным, например, в случае, когда Mifare Plus 4K имитирует Mifare Classic 4K. Перед началом опознавания типа карты, по указателю **ptype** записывается значение SCD_UNKNOWN = 0. Тип карты по указателю **ptype** может принимать следующие значения:

```
#define SCD_UNKNOWN             0     // тип карты неизвестен
#define SCD_UL                  0x01  // Mifare Ultralight
#define SCD_CL1K                0x02  // Mifare Classic 1K
#define SCD_CL4K                0x03  // Mifare Classic 4K
#define SCD_CL1K1               0x04  // новые Mifare Classic 1K, длина UID 7 байт
#define SCD_CL4K1               0x05  // новые Mifare Classic 4K, длина UID 7 байт
#define SCD_PL2S1               0x06..// Mifare Plus 2K, Security Level 1
#define SCD_PL4S1               0x07  // Mifare Plus 4K, Security Level 1
#define SCD_PL2S2               0x08  // Mifare Plus 2K, Security Level 2
#define SCD_PL4S2               0x09  // Mifare Plus 4K, Security Level 2
#define SCD_PL2S3               0x0A  // Mifare Plus 2K, Security Level 3
#define SCD_PL4S3               0x0B  // Mifare Plus 4K, Security Level 3
#define SCD_MINI                0x0C  // Mifare Classic Mini
#define SCD_DF1                 0x0D  // Mifare DesFire, EV1
#define SCD_DF2                 0x0E  // Mifare DesFire, EV2
#define SCD_SMMX                0x0F  // Mifare SmartMX (эмуляция SL1)
#define SCD_SM1K                0x10  // Mifare SmartMX, эмуляция Classic 1K
#define SCD_SM2K                0x11  // Mifare SmartMX, эмуляция Classic 2K
#define SCD_SM4K                0x12  // Mifare SmartMX, эмуляция Classic 4K
```

Необязательный параметр **puidlen** указывает на байт, принимающий значение длины UID в байтах и признак RW-карты в бите 7 (старшем бите). Максимальная длина UID описывается константой **SZ_UID** = 10 байт. Длина UID может принимать значения 0 (для RW-карт), 4, 7 и 10 байт. Для того чтобы функция *mfType* проверила RW-свойства карты, значение по указателю **puidlen** должно быть задано с установленным битом 7 (константа **MFDM_RWC** = 0x80) перед вызовом функции. В случае отсутствия параметра (**puidlen** = NULL) или, если бит 7 по указателю **puidlen** равен 0, RW-свойства карты не проверяются. Перед началом опознавания типа карты, по указателю **puidlen** записывается 0.

Необязательный параметр **puid** является указателем на приёмный буфер UID длиной **SZ_UID** = 10 байт. Рекомендуется объявлять буфер для приёма UID с указанием константы максимального размера буфера: **BYTE uid[SZ_UID]**. Перед началом опознавания типа карты, буфер по указателю **puid** обнуляется.

Необязательный параметр **patqa** является указателем на 2 байта (слово), принимающие значение ATQA. Перед началом опознавания типа карты, по указателю **patqa** записывается 0.

Необязательный параметр **psak** является указателем на байт, принимающий значение SAK. Перед началом опознавания типа карты, по указателю **psak** записывается 0.

Необязательный параметр **patslen** указывает на байт, принимающий значение длины ATS в байтах (значение длины ответа карты на посылку PICC_RATS). Максимальная длина ATS описывается константой **SZ_ATS** = 30 байт. Длина ATS может принимать значения от 0 (если карта не ответила на запрос) до **SZ_ATS**. Для того чтобы функция *mfType* проверила ответ карты на запрос PICC_RATS, требуется чтобы указатель **patslen** или **pats** принимал не нулевое значение. В случае отсутствия параметра (**patslen** = NULL и **pats** = NULL) ответ карты на запрос PICC_RATS не проверяются. Перед началом опознавания типа карты, по указателю **patslen** записывается 0. Если важно точное определение типа карты, следует позаботиться о правильном указателе **patslen** или **pats**: в этом случае будет считан ATS, который примет участие в опознавании типа карты.

Необязательный параметр **pats** является указателем на приёмный буфер UID длиной **SZ_ATS** = 30 байт. Рекомендуется объявлять буфер для приёма ATS с указанием константы максимального размера буфера: **BYTE ats[SZ_ATS]**. Перед началом опознавания типа карты, буфер по указателю **pats** обнуляется. В случае 4х битного ответа, в приёмный буфер будет записан байт, у которого младшая тетрада содержит ответ карты, а старшая равна 0; при этом длина ответа по указателю **patslen** будет установлена равной 1 байту, хотя реальная длина ответа составила 0.5 байта.

ВНИМАНИЕ! Опознавание типа карты производится по ответам стандартного протокола подключения после посылки команды **PICC_REQALL** = 0x52. Если значения полей, ответственных за опознавание карты, установлены неверно, установить тип карты будет невозможно, но можно будет установить наличие RW-свойств. Если карта является RW-картой, то при доступе по RW-протоколу, в карту можно записать правильные значения ответов и восстановить работоспособность карты. После определения типа карты, карта выключается командой **PICC_HALT** = 0x50.

Пример использования *mfType*

“Быстрое” определение типа карты: RW-свойства и ATS не проверяются:

```
sc_uidlen = 0;
if ((err = mfType(&ccr, 7, &sc_type, &sc_uidlen, sc_uid,
    NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto thr_exit;
```

Определение типа карты и RW-свойства без проверки ATS

```
sc_uidlen = MFDM_RWC;
if ((err = mfType(&ccr, 7, &sc_type, &sc_uidlen, sc_uid,
    NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto thr_exit;
if (sc_uidlen & MFDM_RWC) printf(“RW card detected!\n”);
```

Определение типа карты, RW-свойств, ATQA, SAK с уточнением типа по ATS:

```
sc_uidlen = MFDM_RWC;
if ((err = mfType(&ccr, 7, &sc_type, &sc_uidlen, sc_uid,
    &sc_atqa, &sc_sak, &sc_atslen, sc_ats)) != LERR_SUCCESS) goto thr_exit;
if (sc_uidlen & MFDM_RWC) printf(“RW card detected!\n”);
```

mfSelect (mf0xAB)

Подключение к карте и процедура выбора карты SELECT (перевод карты на уровень Security Level 0).

```
DWORD mfSelect(PCCR pccr, BYTE timeout, BYTE rq,
    PBYTE puidlen, PBYTE puid, PWORD patqa, PBYTE psak);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **timeout** устанавливает время, в течение которого ридер будет пытаться связаться с картой для определения её параметров, допустимы значения 0…7. Время ожидания вычисляется по формуле **time** = (**timeout** * 4 + 3) * 50млс (см. описание команды **CMF_SEL**); минимальное время – 0.15с, максимальное – 1.55с.

Обязательный параметр **rq** устанавливает команду включения карты, обычно выбирается из 3х возможных PICC_REQIDL = 0x26, PICC_REQALL = 0x52, PICC_REQRW = 0x40, но для нестандартных карт команда может быть другой.

Необязательный параметр **puidlen** указывает на длину UID. Если параметр вместе с **puid** установлен до вызова функции *mfSelect*, то подключение будет производиться только к карте с указанным в **puid** UID – карта у которой UID будет отличаться от указанного в **puid** и **puidlen** не будет подключена, функция вернёт ошибку. Если значение по **puidlen** перед вызовом функции равно нулю, то подключение будет произведено без учёта UID-а, а UID подключенной карты можно будет считать из **puidlen** и **puid** после вызова функции. Если параметр не используется, значение **puidlen** = NULL.

Необязательный параметр **puid** указывает на буфер UID длиной **SZ_UID** = 10 байт. Если параметр не используется, значение **puid** = NULL.

Необязательный параметр **patqa** является указателем на 2 байта (слово), принимающие значение ATQA. Если параметр не используется, значение **patqa** = NULL.

Необязательный параметр **psak** является указателем на байт, принимающий значение SAK. Если параметр не используется, значение **psak** = NULL.

Пример использования *mfSelect*

Подключение к карте с произвольным UID:

```
if ((err = mfSelect(&ccr, 7, PICC_REQALL, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
```

Подключение к карте с заданным UID (UID = DF3112EA, длина UID 4 байта):

```
sc_uid[0] = 0xDF; sc_uid[1] = 0x31; sc_uid[2] = 0x12; sc_uid[3] = 0xEA;
sc_uidlen = 4;
if ((err = mfSelect(&ccr, 7, PICC_REQALL, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
```

Подключение к RW-карте:

```
if ((err = mfSelect(&ccr, 7, PICC_REQRW, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
```

mfSetAK (mf0xAB)

Установка ключа авторизации (authorization key) в памяти ридера.

```
DWORD mfSetAK(PCCR pccr, BYTE ak, PBYTE key);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

В обязательном параметре **ak** указывается номер ячейки памяти, в которую записывается ключ авторизации. Значение **ak** = 0 соответствует RAM ячейке (ключ пропадает при снятии напряжения питания), значения **ak** = 1..84 сохраняются в энергонезависимой памяти.

Обязательный параметр **key** указывает на ключ авторизации длиной 6 байт.

Пример использования *mfSetAK*

Установка временного ключа (RAM-ключа) AK0 = A0A1A2A3A4A5:

```
if ((err = mfSetAK(&ccr, 0, “\xA0\xA1\xA2\xA3\xA4\xA5”)) != LERR_SUCCESS) goto main_exit;
```

Запись ключа в энергонезависимую память AK84 = B0B1B2B3B4B5:

```
if ((err = mfSetAK(&ccr, 84, “\xB0\xB1\xB2\xB3\xB4\xB5”)) != LERR_SUCCESS) goto main_exit;
```

mfAuth (mf0xAB)

Проверка валидности ключа авторизации в указанной ячейке памяти; производится последовательностью операций: подключение к карте по команде **PICC_REQALL** и процедура выбора карты SELECT (перевод карты на уровень Security Level 0), авторизация по алгоритму Crypto1 (попытка перевода карты на уровень Security Level 1), остановка карты командой **PICC_HALT** после успешной авторизации.

```
DWORD mfAuth(PCCR pccr, BYTE timeout, BYTE acm, BYTE ak, BYTE block,
    PBYTE puidlen, PBYTE puid, PWORD patqa, PBYTE psak);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **timeout** устанавливает время, в течение которого ридер будет пытаться связаться с картой для определения её параметров, допустимы значения 0…7. Время ожидания вычисляется по формуле **time** = (**timeout** * 4 + 3) * 50млс (см. описание команды **CMF_SEL**); минимальное время – 0.15с, максима

Обязательный параметр **acm** устанавливает команду авторизации, которая будет использована для перевода карты на уровень Security Level 1. Обычно это команды **PICC_AUTHENT1A** и **PICC_AUTHENT1B**, но могут быть использованы и другие коды команд.

Пример использования *mfAuth*

Проверка ключа авторизации A0A1A2A3A4A5 в качестве ключа A:

```
// установка временного ключа авторизации (RAM-ключа)
if ((err = mfSetAK(&ccr, 0, “\xA0\xA1\xA2\xA3\xA4\xA5”)) != LERR_SUCCESS) goto main_exit;
// проверка установленного выше ключа авторизации, как ключа авторизации A
if (((err = mfAuth(&ccr, 4, PICC_AUTHENT1A, 0, sc_block,
    &sc_uidlen, sc_uid, NULL, NULL)) != LERR_SUCCESS) && (err != LERR_INVALIDKEY)) goto main_exit;
if (err == LERR_SUCCESS) printf(“Authorization complete.”);
else printf(“Authorization failed.”);
```

mfLong (mf0xAB)

Посылка “длинной” команды и приём ответа на неё. Перед посылкой данных карте, к данным добавляется 9й бит паритета и контрольная сумма (2 байта), а если была произведена авторизация (переход на уровень Security Level 1), то производится шифрование передаваемых данных по алгоритму Crypto1. После приёма данных из карты, производится удаление паритета принятых данных и дешифрование, если карта находится в состоянии Security Level 1. С помощью *mfLong* очень удобно строить низкоуровневое взаимодействие с картой.

```
DWORD mfLong(PCCR pccr, PBYTE txbuf, BYTE txlen, PBYTE ans, PBYTE pabitlen)
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **txbuf** указывает на буфер с передаваемыми данными.

Обязательный параметр **txlen** содержит длину передаваемых данных в буфере **txbuf**. Максимальный размер передаваемых данных – 26 байт.

Необязательный параметр **ans** указывает на буфер для приёма ответа. Максимальный размер приёмного буфера **ans** – 32 байта.

Необязательный параметр **pabitlen** указывает на длину данных в приёмном буфере **ans** в битах (не в байтах!).

Пример использования *mfLong*

Остановка карты командой PICC_HALT:

```
BYTE    sc_halt[2] = { PICC_HALT, 0 };
if (((err = mfLong(&ccr, sc_halt, sizeof(sc_halt), NULL, NULL)) != LERR_SUCCESS) &&
    (err != LERR_FAIL)) goto main_exit;
```

Чтение 16 байт данных из карты Ultralight, начиная со страницы 0 и включая 2 байта CRC:

```
BYTE    sc_rdblk[2] = { PICC_READ16, 0 };
BYTE    rxbuf[32], bitlen;
if ((err = mfSelect(&ccr, 7, PICC_REQALL, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
if (((err = mfLong(&ccr, sc_rdblk, sizeof(sc_rdblk), rxbuf, &bitlen)) != LERR_SUCCESS) &&
    (err != LERR_FAIL)) goto main_exit;
```

mfCalcCrcA (mf0xAB)

Вычисление CRC для содержимого буфера по алгоритму ISO-14443A. Функция используется для “ручного” формирования данных, обычно при исследовании уязвимостей карт, например, в программе *mfoc*. Функция `mfCalcCrcA возвращает CRCA.`

```
WORD mfCalcCrcA(PBYTE buf, DWORD len)
```

Обязательный параметр **buf** указывает на данные, для которых вычисляется CRC.

Обязательный параметр **len** устанавливает длину данных в буфере.

mfAppendCrcA (mf0xAB)

Вычисление CRC для содержимого буфера по алгоритму ISO-14443A и добавление вычисленного значения к содержимому буфера. Функция используется для “ручного” формирования данных, обычно при исследовании уязвимостей карт, например, в программе *mfoc*.

```
void mfAppendCrcA(PBYTE buf, DWORD len)
```

Обязательный параметр **buf** указывает на буфер с данными, для которых вычисляется CRC. **ВНИМАНИЕ!** Размер буфера должен быть как минимум на 2 байта больше, чем длина данных в нём, чтобы добавление CRC не вызвало программных сбоев.

Обязательный параметр **len** устанавливает длину данных в буфере. После вызова *mfAppendCrcA* параметр **len** не изменяется! Для указания новой длины буфера параметр **len** следует увеличить “вручную”.

Пример использования *mfAppendCrcA*

Формирование команды авторизации по ключу A для блока 0 “вручную”:

```
BYTE    sc_auth[4] = { PICC_AUTHENT1A, 0, 0, 0 }; // команда(26) + место для CRC (26)
DWORD   sc_authlen = 2;  // длина команды авторизации 2 байта: код_команды + номер_блока
mfAppendCrcA(sc_auth, sc_authlen);  // добавление CRCA к команде авторизации
sc_authlen += 2;         // коррекция длины данных в буфере с учётом длины CRC
```

mfIO (mf0xAB)

Базовая функция байтового взаимодействия с картой: на базе этой функции можно построить большую часть интерфейса с картой. Функция используется для передачи данных, сформированных “вручную”; обычно используется при исследовании уязвимостей карт, например, в программе *mfoc*.

```
DWORD mfIO(PCCR pccr, PBYTE buf, BYTE len, PBYTE par,
    PBYTE ans, PBYTE pabitlen, PBYTE anspar);
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **buf** указывает на буфер с передаваемыми данными.

Обязательный параметр **len** содержит длину передаваемых данных в буфере **buf**. Максимальный размер передаваемых данных – 27 байт.

Обязательный параметр **par** указывает на буфер с битами паритета для передаваемых данных в буфере **buf**. Буфер **par** содержит 1(единицы) и 0(нули).

Необязательный параметр **ans** указывает на приёмный буфер. Длина приёмного буфера **ans** – 32 байта.

Необязательный параметр **pabitlen** указывает на байт, принимающий длину данных в приёмном буфере **ans** в битах. Максимальное значение – 255.

Необязательный параметр **anspar** указывает на буфер, в который записываются биты паритета для принятых в буфер **ans** данных. Буфер **anspar** содержит 1(единицы) и 0(нули).

Пример использования *mfIO*

Примеры использования *mfIO* лучше всего смотреть в исходных текстах программы *mfoc*.

Синтаксис вызова приводится ниже:

```
// посылка шифрованной команды авторизации
if (((err = mfIO(pccr, AuthEnc, 4, AuthEncPar, Rx, &RxBitlen, RxPar)) != LERR_SUCCESS) &&
    (err != LERR_FAIL)) goto main_exit;
if (RxBitlen != 32)
{
    err = LERR_FAIL;
    goto main_exit;
}
```

mfBitIO (mf0xAB)

Базовая функция битового взаимодействия с картой: на базе этой функции можно построить весь интерфейс управления картой. Функция используется для передачи данных, сформированных “вручную”; обычно используется при исследовании уязвимостей карт, например, в программе *scprobe*.

```
DWORD mfBitIO(PCCR pccr, PBYTE buf, BYTE blen,
    PBYTE ans, PBYTE pabitlen, PBYTE anspar)
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **buf** указывает на буфер с передаваемыми данными.

Обязательный параметр **blen** содержит длину передаваемых данных в буфере **buf** в битах. Максимальный размер передаваемых данных – 255 битов (32 байта).

Необязательный параметр **ans** указывает на приёмный буфер. Длина приёмного буфера **ans** – 32 байта.

Необязательный параметр **pabitlen** указывает на байт, принимающий длину данных в приёмном буфере **ans** в битах. Максимальное значение – 255.

Необязательный параметр **anspar** указывает на буфер, в который записываются биты паритета для принятых в буфер **ans** данных. Буфер **anspar** содержит 1(единицы) и 0(нули).

Пример использования *mfBitIO*

Примеры использования *mfBitIO* лучше всего смотреть в исходных текстах программы *scprobe*.

Синтаксис вызова приводится ниже:

```
// посылка команды подключения к карте (длина 7бит)
txbuf[0] = PICC_REQALL;
if (((err = mfBitIO(&ccr, txbuf, 7, rxbuf, &bitlen, NULL)) != LERR_SUCCESS) &&
    (err != LERR_FAIL)) goto main_exit;
if ((err == LERR_SUCCESS) && (bitlen == 16)) printf(“ATQA = %02X%02X\n”, rxbuf[0], rxbuf[1]);
```

mfDataRead (mf0xAB)

Универсальная функция для чтения данных с карты. Выполняет все необходимые для чтения данных дополнительные операции: подключение, выбор карты, авторизацию Crypto1 (если требуется) и чтение данных. Максимальный объём считываемых за один раз данных – 64 байта. Функция унифицирована для чтения данных с большинства Mifare карт: для чтения UL, CL, PL и RW-карт достаточно правильно указать параметры.

```
DWORD mfDataRead(PCCR pccr, BYTE timeout, BYTE mfdm, BYTE acm, BYTE ak,
    BYTE start, BYTE len, PBYTE pdata,
    PBYTE puidlen, PBYTE puid, PWORD patqa, PBYTE psak)
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **timeout** устанавливает время, в течение которого ридер будет пытаться связаться с картой для определения её параметров, допустимы значения 0…7. Время ожидания вычисляется по формуле **time** = (**timeout** * 4 + 3) * 50млс (см. описание команды **CMF_SEL**); минимальное время – 0.15с, максимальное – 1.55с.

Обязательный параметр **mfdm** (**Mi**ф**F**are **D**ata **M**ode) устанавливает режим чтения данных. Бит 0 кодирует режим чтения (блоковый / страничный, константы **MFDM_BLOCK** / **MFDM_PAGE** соответственно), бит 7 кодирует режим открытия карты (нормальная карта / RW-карта, константы **MFDM_NMC** / **MFDM_RWC** соответственно).

Обязательный параметр **acm** (**A**uthorization **C**o**M**mand) устанавливает режим чтения данных. Обычно это команда **PICC_AUTHENT1A** или **PICC_AUTHENT1B**, но может быть указана любая другая команда. Если **acm** = 0 (например, для UL карт), то авторизация не производится.

Обязательный параметр **ak** (**A**uthorization **K**ey) устанавливает номер ключа авторизации 0…84. Если **acm** = 0 (авторизация не требуется), то значение **ak** безразлично.

Обязательный параметр **start** = 0…255 указывает значение начального блока/страницы для чтения.

Обязательный параметр **len** = 1…4 для блокового чтения, или **len** = 1…16 для страничного чтения, указывает длину читаемых данных в блоках или страницах. Длина считываемых данных суммарно не должна превышать 64 байт. Длину можно найти как **datalen = len * SZ_MFBLK** при чтении блоков, или **datalen = len * SZ_MFPAGE** при чтении страниц.

Необязательный параметр **pdata** указывает на приёмный буфер, в который считываются данные. Следует следить за тем, чтобы размер буфера соответствовал размеру считываемых данных.

Необязательный параметр **puidlen** указывает на длину UID. Если параметр вместе с **puid** установлен до вызова функции *mfDataRead*, то подключение будет производиться только к карте с указанным в **puid** UID – карта у которой UID будет отличаться от указанного в **puid** и **puidlen**, не будет подключена, функция вернёт ошибку. Если значение по **puidlen** перед вызовом функции равно нулю, то подключение будет произведено без учёта UID-а, а UID подключенной карты можно будет считать из **puidlen** и **puid** после вызова функции. Если параметр не используется, значение **puidlen** = NULL.

Необязательный параметр **puid** указывает на буфер UID длиной **SZ_UID** = 10 байт. Если параметр не используется, значение **puid** = NULL.

Необязательный параметр **patqa** является указателем на 2 байта (слово), принимающие значение ATQA. Если параметр не используется, значение **patqa** = NULL.

Необязательный параметр **psak** является указателем на байт, принимающий значение SAK. Если параметр не используется, значение **psak** = NULL.

Пример использования *mfDataRead*

Чтение данных из карты Mifare Ultralight с произвольным UID в буфер *sc_data*. После чтения карты возвращаются длина UID карты, UID, ATQA и SAK.

```
// считывание начинается с 4ой страницы, читается 8 страниц (32 байта)
sc_uidlen = 0;  // читаем карту с произвольным UID, а не с заданным
if ((err = mfDataRead(&ccr, 7, MFDM_NMC | MFDM_PAGE, 0, 0, 4, 8,
    sc_data, &sc_uidlen, sc_uid, &sc_atqa, &sc_sak)) != LERR_SUCCESS) goto main_exit;
```

Чтение данных из карт Mifare Classic, Mifare Plus в буфер *sc_data*. После чтения карты возвращаются длина UID карты, UID, ATQA и SAK.

```
// задаём значение ключа авторизации
memcpy(sc_ak, "\xA0\xA1\xA2\xA3\xA4\xA5", 6);
// установка ключа авторизации в RAM (ключ номер 0)
if ((err = mfSetAK(&ccr, 0, sc_ak)) != LERR_SUCCESS) goto main_exit;
// устанавливаем команду авторизации: по ключу A - PICC_AUTHENT1A, по ключу B - PICC_AUTHENT1B
sc_acm = PICC_AUTHENT1A;
// чтение данных из карты: стартовый блок 1, длина 2 блока
sc_uidlen = 0;  // читаем карту с произвольным UID, а не с заданным
if ((err = mfDataRead(&ccr, 7, MFDM_NMC) | MFDM_BLOCK, sc_acm, 0, 1, 2,
    sc_data, &sc_uidlen, sc_uid, &sc_atqa, &sc_sak)) != LERR_SUCCESS) goto main_exit;
```

Чтение данных из RW-карты, имитирующей Mifare Classic, Mifare Plus в буфер *sc_data*.

```
// считывание начинается с 0го блока, читается 4 блока (64 байта)
if ((err = mfDataRead(&ccr, 7, MFDM_RWC | MFDM_BLOCK, 0, 0, 0, 4,
    sc_data, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
```

mfDataWrite (mf0xAB)

Универсальная функция для записи данных на карту. Выполняет все необходимые для записи данных дополнительные операции: подключение, выбор карты, авторизацию Crypto1 (если требуется) и запись данных. Максимальный объём записываемых за один раз данных – 64 байта. Функция унифицирована для записи данных на большинство Mifare карт: для записи UL, CL, PL и RW-карт достаточно правильно указать параметры.

```
DWORD mfDataWrite(PCCR pccr, BYTE timeout, BYTE mfdm, BYTE acm, BYTE ak,
    BYTE start, BYTE len, PBYTE pdata,
    PBYTE puidlen, PBYTE puid, PWORD patqa, PBYTE psak)
```

Обязательный параметр **pccr** указывает на структуру псевдо-хэндла.

Обязательный параметр **timeout** устанавливает время, в течение которого ридер будет пытаться связаться с картой для определения её параметров, допустимы значения 0…7. Время ожидания вычисляется по формуле **time** = (**timeout** * 4 + 3) * 50млс (см. описание команды **CMF_SEL**); минимальное время – 0.15с, максимальное – 1.55с.

Обязательный параметр **mfdm** (**Mi**ﬁare **D**ata **M**ode) устанавливает режим чтения данных. Бит 0 кодирует режим записи (блоковый / страничный, константы **MFDM_BLOCK** / **MFDM_PAGE** соответственно), бит 7 кодирует режим открытия карты (нормальная карта / RW-карта, константы **MFDM_NMC** / **MFDM_RWC** соответственно).

Обязательный параметр **acm** (**A**uthorization **C**o**M**mand) устанавливает режим записи данных. Обычно это команда **PICC_AUTHENT1A** или **PICC_AUTHENT1B**, но может быть указана любая другая команда. Если **acm** = 0 (например, для UL карт), то авторизация не производится.

Обязательный параметр **ak** (**A**uthorization **K**ey) устанавливает номер ключа авторизации 0…84. Если **acm** = 0 (авторизация не требуется), то значение **ak** безразлично.

Обязательный параметр **start** = 0…255 указывает значение начального блока/страницы для записи.

Обязательный параметр **len** = 1…4 для блоковой записи, или **len** = 1…16 для страничной записи, указывает длину записываемых данных в блоках или страницах. Длина записываемых данных суммарно не должна превышать 64 байт. Длину можно найти как **datalen = len * SZ_MFBLK** при записи блоков, или **datalen = len * SZ_MFPAGE** при записи страниц.
ВНИМАНИЕ! Запись страниц производится в обратном порядке, по убыванию номеров страниц, для того, чтобы OTP и Lock области были записаны последними.

Обязательный параметр **pdata** указывает на буфер, данные из которого записываются на карту.

Необязательный параметр **puidlen** указывает на длину UID. Если параметр вместе с **puid** установлен до вызова функции *mfDataWrite*, то подключение будет производиться только к карте с указанным в **puid** UID – карта у которой UID будет отличаться от указанного в **puid** и **puidlen**, не будет подключена, функция вернёт ошибку. Если значение по **puidlen** перед вызовом функции равно нулю, то подключение будет произведено без учёта UID-а,

а UID подключенной карты можно будет считать из **puidlen** и **puid** после вызова функции. Если параметр не используется, значение **puidlen** = NULL.

Необязательный параметр **puid** указывает на буфер UID длиной **SZ_UID** = 10 байт. Если параметр не используется, значение **puid** = NULL.

Необязательный параметр **patqa** является указателем на 2 байта (слово), принимающие значение ATQA. Если параметр не используется, значение **patqa** = NULL.

Необязательный параметр **psak** является указателем на байт, принимающий значение SAK. Если параметр не используется, значение **psak** = NULL.

Пример использования *mfDataWrite*

Запись данных в карту Mifare Ultralight с произвольным UID из буфера *sc_data*. После записи карты возвращаются длина UID карты, UID, ATQA и SAK.

```
// запись начинается с 4ой страницы, записывается 8 страниц (32 байта)
sc_uidlen = 0;  // записываем карту с произвольным UID, а не с заданным
if ((err = mfDataWrite(&ccr, 7, MFDM_NMC | MFDM_PAGE, 0, 0, 4, 8,
    sc_data, &sc_uidlen, sc_uid, &sc_atqa, &sc_sak)) != LERR_SUCCESS) goto main_exit;
```

Запись данных в карту Mifare Classic, Mifare Plus с произвольным UID из буфера *sc_data*. После записи карты возвращаются длина UID карты, UID, ATQA и SAK.

```
// задаём значение ключа авторизации
memcpy(sc_ak, "\xB0\xB1\xB2\xB3\xB4\xB5", 6);
// установка ключа авторизации в RAM (ключ номер 0)
if ((err = mfSetAK(&ccr, 0, sc_ak)) != LERR_SUCCESS) goto main_exit;
// устанавливаем команду авторизации: по ключу A - PICC_AUTHENT1A, по ключу B - PICC_AUTHENT1B
sc_acm = PICC_AUTHENT1B;
// запись данных в карту: стартовый блок 1, длина 2 блока
sc_uidlen = 0;  // записываем карту с произвольным UID, а не с заданным
if ((err = mfDataWrite(&ccr, 7, MFDM_NMC) | MFDM_BLOCK, sc_acm, 0, 1, 2,
    sc_data, &sc_uidlen, sc_uid, &sc_atqa, &sc_sak)) != LERR_SUCCESS) goto main_exit;
```

Запись данных в RW-карту, имитирующую Mifare Classic, Mifare Plus из буфера *sc_data*.

```
// запись начинается с 0го блока, записывается 4 блока (64 байта)
if ((err = mfDataWrite(&ccr, 7, MFDM_RWC | MFDM_BLOCK, 0, 0, 0, 4,
    sc_data, NULL, NULL, NULL, NULL)) != LERR_SUCCESS) goto main_exit;
```