libbasiccard / libbccrypto

Version 0.2.5
Copyright (C) 2004-2009 CHZ-Soft, Christian Zietz, <czietz@gmx.net>
Documentation last updated August 30, 2009

libbasiccard and libbccrypto (some times referenced together as libbasiccard in this document) are a libraries to communicate with a BasicCard, a smart card made by ZeitControl. (see http://www.basiccard.com/) They try to follow the API for Microsoft Windows defined by ZeitControl as closely as possible while extending its functionality where this is appropriate. They run on Linux, Microsoft Windows and probably as well on other operating systems with an implementation of PC/SC. They were successfully compiled on FreeBSD.

What's new

Version 0.2.5

- Encryption support in libbccrypto was updated to support ECC-211 (available on request) and SHA-256.
- Minor bugs were fixed.

Version 0.2.0

- A port of the BasicCard cryptographic API (zccrypt) in form of a new library called libbccrypto was added. (Elliptic curve cryptography support is available on request.)
- Support for all the commands of the MultiApplication BasicCard was added. See “MultiApplication BasicCard support” below for more information.
- A more comfortable management of the default smart card reader was added.
- The parser for key definition files was improved again. The PCRE library is not needed anymore.
- The included cryptographic library libtomcrypt was updated to version 1.16.
- Minor bugs were fixed.

Version 0.1.5

- Preliminary support for the MultiApplication BasicCard has been added.
- The included cryptographic library libtomcrypt has been updated to version 0.98.
- The parser for key definition files used in ZCBciReadKeyFile has been improved.

Prerequisites

libbasiccard is built on top of PC/SC. For Linux and FreeBSD the PC/SC lite library, a part of the MUSCLE project, is needed. (http://www.musclecard.com/) Additionally you'll need a PC/SC compatible driver for your smart card reader to use libbasiccard.

Compilation and installation

Linux/Unix specific

Binary packages for your distribution may be provided. If you want to compile from source instead, extract the provided source code archive and call `make` in its top level directory to build libbasiccard, libbccrypto and all other components needed. Depending on your distribution, you may need the development package of PC/SC lite containing winscard.h and other header files. These header files are needed to build libbasiccard. If they are not located at /usr/include/PCSC or /usr/local/include/PCSC you will have to add them to your compiler's include path manually.

To install libbasiccard and libbccrypto call `make install` in the top level directory. This will install static and shared libraries, header files, this documentation and a program called bcsetreader to set the default reader for libbasiccard. Optionally you can pass a base directory under which files will be installed:
```
make PREFIX=/directory install
```

In the libbasiccard-x.x.x/demo subdirectory you'll find a simple example application for libbasiccard called calcdemo. It communicates with a 'BasicCard Mini-Calculator'-card whose source code is provided with the BasicCard development environment.
Try running 'calcdemo 17+4' with an appropriate smart card inserted into the default reader.

Usage

To use libbasiccard from a C(++) program include zccri.h, zcbci.h and possibly chzbci.h and tell your compiler to link against libbasiccard (that's `-lbasiccard` for gcc).

To use libbccrypto (see also chapter “Cryptography support” below) include zccrypt.h and possibly chzcrypt.h and link against libbccrypto.

You can find a description of the functions in the documentation provided by ZeitControl with their Windows SDK and API. See below for a list of issues and functions specific to this implementation.

Cryptography support

An additional library providing an implementation of the cryptographic API for BasicCards is included. You'll find further information in the section “BasicCard cryptography library”.

MultiApplication BasicCard support

Version 0.2.0 of libbasiccard adds full support for all commands of the MultiApplication BasicCard. Currently there is no API for C by ZeitControl which supports this new smart card and perhaps there never will be. Because of this, everything described in this chapter is only supported by libbasiccard and not by the official API from ZeitControl. To show this, the unofficial identifiers start with CHZBci... instead of ZCBci.... To use them, include chzbci.h as well as zcbci.h.

libbasiccard supports all algorithms used by Enhanced, Professional and MultiApplication BasicCards for encrypted communication, i.e. DES, 3DES, AES, EAX and OMAC (the last three with key lengths of 128, 192 and 256 bits). The newly added algorithm IDs are defined in chzbci.h. The procedure to start encrypted communication has changed for MultiApplication BasicCards. On this card keys are no longer identified by a key number but by a component ID (CID). Thus a new function CHZBciStartMultiAppEncryption has been added. To get the CID of key whose file name on the card is known, there is the function CHZBciFindComponent. In the terminal keys are still referenced by a numerical index so you need to pass both the CID of a key on the card and the index of the matching key in the terminal to all MultiApplication BasicCard functions which use keys.

Please see “BasicCard command interface / Unofficial functions and constants” (below) for more information on the implementation as well as the BasicCard manual for details about the commands implemented in MultiApplication BasicCards.

License

When referring to source files, this license only applies to source files which are copyrighted by Christian Zietz. Some files (include files and most of the files not in the libbasiccard and libbccrypto folders) have different copyright owners and may have different license terms. Nevertheless care has been taken that the whole library can be used according to this license.

Copyright (C) 2004-2009 CHZ-Soft, Christian Zietz

You have a royalty-free right to use, modify, reproduce and distribute the source and binary files covered by the license or any modified version of these files or any derivative work in any way you find useful, provided that:

- You agree that CHZ-Soft has no warranty, obligations or liability for any of these files.
- When distributing in source form: The respective copyright and license notices in the source files and in the documentation remain intact.
- When distributing in binary form: You include the following copyright notice in the program or documentation: Portions Copyright (C) 2004-2009 CHZ-Soft, Christian Zietz

Acknowledgments

- Many thanks go to Tom St Denis for his excellent libtomcrypt and libtommath (http://www.libtomcrypt.com/) libraries that are used in libbasiccard and libbccrypto.
- ZeitControl supported my development with information about their BasicCards, code snippets and samples. Thank you very much!

Contact

If you want to send bug-reports and feedback concerning libbasiccard, you can contact me via e-mail: czietz@gmx.net. For general questions about the BasicCard that have nothing to do with this implementation please write to ZeitControl or use their web based discussion board (http://www.basiccard.com/)

Common reader interface

The common reader interface (Cri) is a low-level API defined by ZeitControl to allow third-party applications to communicate with their BasicCard smart cards. It is normally used together with the BasicCard command interface (Bci).

The following text is not intended as a documentation of the Cri. It only describes issues related to this specific implementation of the Cri. For an actual documentation of the functions please read the documents provided by ZeitControl with their Windows SDK and API or have a look at zccri*.h.

General issues

- This implementation only supports smart card readers with a PC/SC interface. Starting with version 0.2.0 the names of the readers are prefixed with “PCSC:” to improve compatibility with the official API. Please note that the names may still differ between Windows and Linux/Unix because they are assigned by the driver.
- No more than eight readers are supported.
- The following procedure is used to determine the name of the default smart card reader under Linux/Unix:
    - If a default reader was set earlier during the run time of the current process, that reader is used
    - else if an environment variable named CRIREADER exists, its value is used as the name of the default reader
    - else if a file named .crireader exists in the user's home directory, its first line is used.
    - else if the file /etc/crireader exists, its first line is used.
    - Otherwise or in case the reader specified by one of the methods above does not exist, the first PC/SC reader will be used.
- Setting the default smart card reader creates or modifies the .crireader file in the user's home directory. Please note that due to compatibility reasons if the CRIREADER environment variable exists, processes other than the one in which the default reader was set will continue to use the value of the environment variable instead of the name stored in the file.

Specific remarks

Function	**works**	**Remarks**
ZCCriGetReaderCount	x	
ZCCriGetReaderName	x	
ZCCriOpenReader	x	
ZCCriCloseReader	x	
ZCCriWaitCard	x	
ZCCriConnect	x	
ZCCriReconnect	x	
ZCCriDisconnect	x	
ZCCriTransaction	x	
ZCCriTransaction2	x	
ZCCriSetLogFile	x	
ZCCriRegisterService		Not supported, just returns an error
ZCCriDeregisterService		Not supported, just returns an error
ZCCriSetDefaultReader	x	See “General issues” for more information.
ZCCriGetDefaultReader	x	See “General issues” for more information.
ZCCriSelectReaderDialog	x	A simple text-based dialog is displayed.
ZCCriGetFeatures	x	No features supported
ZCCriCardInReader		Not supported, just returns an error
ZCCriSetOptions		Not supported, just returns an error

Function	works	Remarks
ZCCriGetOptions		Pretends that no options are set
ZCCriSetOptions2		Not supported, just returns an error
ZCCriGetOptions2		Pretends that no options are set

BasicCard command interface

The BasicCard command interface (Bci) is an API defined by ZeitControl to allow third-party applications to communicate with their BasicCard smart cards. It needs the low-level common reader interface (Cri) to work.

The following text is not intended as a documentation of the Bci. It only describes issues related to this specific implementation of the Bci. For an actual documentation of the functions please read the documents provided by ZeitControl with their Windows SDK and API or have a look at zcbci.h.

See below for unofficial additions to the Bci in libbasiccard.

General issues

- Encrypted communication with a Compact BasicCard (SG-LFSR algorithm) is not supported.

Specific remarks

Function	works	Remarks
ZCBciAttach	x	
ZCBciDetach	x	
ZCBciCreateHandle		Not supported, just returns an error
ZCBciAttachHandle		Not supported, just returns an error
ZCBciSetKey	x	
ZCBciSetPoly		Compact BasicCard is not supported.
ZCBciReadKeyFile	x	Correctly reads files created by ZeitControls Keygen.exe containing keys up to 64 bytes (512 bits) long. Reading polynomials is not supported.
ZCBciTransaction	x	
ZCBciTransaction2	x	
ZCBciGetState	x	
ZCBciGetStateVersion	x	
ZCBciEepromSize	x	
ZCBciClearEeprom	x	
ZCBciWriteEeprom	x	
ZCBciReadEeprom	x	
ZCBciEepromCRC	x	
ZCBciSetState	x	
ZCBciGetApplicationID	x	
ZCBciStartEncryption	x	Random value is generated internally, supplied one will be discarded. Compact BasicCard (SG-LFSR algorithm) is not supported.
ZCBciStartProEncryption	x	Random values are generated internally, supplied ones will be discarded. Compact BasicCard (SG-LFSR algorithm) is not supported.
ZCBciEndEncryption	x	
ZCBciEcho	x	
ZCBciMkDir	x	
ZCBciRmDir	x	
ZCBciChDir	x	
ZCBciCurDir	x	
ZCBciRename	x	
ZCBciDirCount	x	
ZCBciDirFile	x	

Function	works	Remarks
ZCBciGetAttr	x	
ZCBciEraseFile	x	
ZCBciOpenFile	x	
ZCBciCloseFile	x	
ZCBciCloseAllFiles	x	
ZCBciFileLength	x	
ZCBciGetFilepos	x	
ZCBciSetFilepos	x	
ZCBciQueryEof	x	
ZCBciFileGet	x	
ZCBciFileGetPos	x	
ZCBciFilePut	x	
ZCBciFilePutPos	x	
ZCBciFileRead	x	
ZCBciFileReadLong	x	
ZCBciFileReadSingle	x	
ZCBciFileReadString	x	
ZCBciFileReadUser	x	
ZCBciFileReadUser2	x	
ZCBciFileReadLine	x	
ZCBciFileWriteLong	x	
ZCBciFileWriteSingle	x	
ZCBciFileWriteUser	x	
ZCBciFileWriteString	x	
ZCBciFileWrite	x	
ZCBciFilePrint	x	
ZCBciGetLockInfo	x	
ZCBciSetFileLock	x	

BasicCard command interface / Unofficial functions and constants

These functions are not part of the original API by ZeitControl and are only defined in libbasiccard. Include chzbci.h to use them. (Most of the functions were added to support the MultiApplication BasicCard, so please see the chapter on MultiApplication BasicCard support, too.)

Parameters and return values that aren't explained are the same as for ZCBci... functions.

Functions (in alphabetic order)

CHZBciAuthenticateFile (new in 0.2.0)

```
ZCBCIRET CHZBciAuthenticateFile(ZCBCICARD card, WORD *pSW1SW2, WORD keyCID, BYTE
algo, const CHAR *pszPath, const void *pSignature)
```

Authenticates the file at *pszPath* with a signature given the component ID of the key on the card *keyCID*, the algorithm used for the signature *algo* (EC167 or OMAC-AES, see below for appropriate constants) and a pointer to the signature data *pSignature*. The size of the signature is derived from the algorithm.

CHZBciComponentName (new in 0.2.0)

```
ZCBCIRET CHZBciComponentName(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID, CHAR
*pszName, BYTE cbName)
```

Returns the name of the component with the ID *cmpCID* in the buffer *pszName* which has a length of at least *cbName*.

CHZBciCreateComponent (new in 0.2.0)

```
ZCBCIRET CHZBciCreateComponent(ZCBCICARD card, WORD *pSW1SW2, BYTE type, const CHAR
*pszPath, const void *pAttr, BYTE cbAttr, const void *pData, BYTE cbData, WORD *pCID)
```

Creates a component of type *type* with the name and in the directory specified in *pszPath* given a pointer to its attributes *pAttr* and to its data *pData* as well the size of attributes and data *cbAttr* and *cbData*. The ID of the newly created component is returned in *pCID*.

Please note that the definition of attributes and data differs between different types of components. See below for constants and structures to help with that. Details are given in the BasicCard user manual (section 5.8).

CHZBciDeleteComponent (new in 0.2.0)

```
ZCBCIRET CHZBciDeleteComponent(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID)
```

Deletes the component with the ID *cmpCID*.

CHZBciExternalAuthenticate

```
ZCBCIRET CHZBciExternalAuthenticate(ZCBCICARD card, WORD *pSW1SW2, BYTE algo, BYTE
key, WORD keyCID)
```

Authenticates the terminal to the BasicCard by getting a challenge from the card, encrypting it using the specified key with the number *key* and sending the response back to card together with the component ID *keyCID* of the matching key on the card. Thus it is possible to prove to the card that the terminal has knowledge of a specific key without transmitting this key.
Call CHZBciFindComponent to get a CID. Type CHZBCI_COMP_KEY (0x70) is predefined for this purpose.

CHZBciFindComponent

```
ZCBCIRET CHZBciFindComponent(ZCBCICARD card, WORD *pSW1SW2, BYTE type, const CHAR
*pszPath, WORD *pCID)
```

Finds a component of type *type* and with a file name of *pszPath* on a MultiApplication BasicCard and returns its component ID in *pCID*. See below for defined types.

CHZBciGetExtendedID

ZCBCIRET CHZBciGetExtendedID(ZCBCICARD card, WORD *pSW1SW2, BYTE index, BYTE type, CHAR *pszID, BYTE cbID)

Sends an extended GET APPLICATION ID command with index *index* and type *type* to a MultiApplication BasicCard. The result will be stored in the buffer *pszID* which is *cbID* bytes long. This function can be used for example to get the file name of an application or the serial number of a card. Please see the BasicCard manual (section 8.7.10) for detailed information.

CHZBciGetFreeMemory (new in 0.2.0)

```
ZCBCIRET CHZBciGetFreeMemory(ZCBCICARD card, WORD *pSW1SW2, WORD *pTotal, WORD
*pBlock)
```

Returns the total amount of free memory on the BasicCard in *pTotal* and the size of the largest consecutive block of memory in *pBlock*.

CHZBciGetKey (new in 0.2.0)

```
ZCBCIRET CHZBciGetKey(ZCBCICARD card, BYTE keynum, BYTE *pKey, BYTE *pcbKey)
```

Gets a key previously stored by ZCBciSetKey or ZCBciReadKeyFile. The numerical index of the key to be read is *keynum*, *pKey* points to the buffer where it will be stored. When calling this function *pcbKey* points to the size of the buffer, when the function returns it will contain the size of the key. ZCBCI_ERROR_OVERFLOW is returned when the buffer is too small for the key.

CHZBciGetLibVersion (new in 0.2.0)

```
ZCBCIRET CHZBciGetLibVersion(DWORD *pVer)
```

Returns the version of libbasiccard in *pVer*. The version is returned as BCD value where 0x00000xxyz means version xx.y.z. Currently 0x00000020 is returned.

CHZBciGrantPrivilege

```
ZCBCIRET CHZBciGrantPrivilege(ZCBCICARD card, WORD *pSW1SW2, WORD prvCID, const CHAR
*pszPath)
```

Grants the privilege with the component ID *prvCID* to the application with the file name *pszPath*. If *pszPath* is NULL, the privilege is granted to the terminal. Please note that granting a privilege to the terminal is not supported in revisions A and B of the BasicCard 6.5.

CHZBciInternalAuthenticate

```
ZCBCIRET CHZBciInternalAuthenticate(ZCBCICARD card, WORD *pSW1SW2, BYTE algo, BYTE
key, WORD keyCID)
```

Authenticates the BasicCard to the terminal by sending a random challenge to the card together with a component ID *keyCID* of a suitable key and an algorithm ID *algo* and comparing the response with the challenge encrypted with the matching key (with the number *key*) in the terminal. Thus the BasicCard can prove that it knows a specific key without transmitting it.
Returns ZCBCI_ERROR_TRANS_FAILED with an SW1SW2 of 0x66C7 (swBadAuthenticate) when the card couldn't authenticate itself properly (i.e. the response does not match).

CHZBciLoadSequence (new in 0.2.0)

```
ZCBCIRET CHZBciLoadSequence(ZCBCICARD card, WORD *pSW1SW2, BYTE phase)
```

Starts, ends or aborts a load sequence depending on the parameter *phase*. See below for constants to be used here.

CHZBciReadComponentAttr (new in 0.2.0)

```
ZCBCIRET CHZBciReadComponentAttr(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID, void
*pAttr, BYTE *pcbAttr)
```

Reads the attribute field of the component with the ID *cmpCID* and stores it in the memory buffer to which *pAttr* points. When calling the function, *pcbAttr* points to the size of the buffer in bytes, when the function returns, the size of data stored at *pAttr* is returned in *pcbAttr*.
A return value of ZCBCI_ERROR_OVERFLOW signals that the buffer was too small to store the attribute data.
The format of a component's attribute and data fields depend on the type of the component and also varies between read and write operations. See below for structures that can be used as buffer and read the BasicCard user manual (section 5.8) for more details.

CHZBciReadComponentData (new in 0.2.0)

```
ZCBCIRET CHZBciReadComponentData(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID, void
*pData, BYTE *pcbData)
```

Reads the data field of a component. See CHZBciReadComponentAttr for a definition of the parameters.

CHZBciReadRightsList (new in 0.2.0)

```
ZCBCIRET CHZBciReadRightsList(ZCBCICARD card, WORD *pSW1SW2, const CHAR *pszPath,
WORD *pList, BYTE *pcbList)
```

Reads the rights list of the component named *pszPath* and stores them into a buffer pointed to by *pList*. The size of the buffer in WORDs is passed in *pcbList.* When the function returns successfully *pcbList* contains the number of items read into the list. Every WORD in the list contains the component ID of a privilege or a key.

CHZBciSecureTransport (new in 0.2.0)

```
ZCBCIRET CHZBciSecureTransport(ZCBCICARD card, WORD *pSW1SW2, WORD keyCID, BYTE algo,
const void *pNonce, BYTE cbNonce)
```

Starts or ends a secure transport session using the key with ID *keyCID*, the algorithm *algo* (EAX-AES) and the nonce pointed to by *pNonce* which has the length *cbNonce*.

CHZBciSelectApplication

```
ZCBCIRET CHZBciSelectApplication(ZCBCICARD card, WORD *pSW1SW2, const CHAR *pszPath)
```

Selects the application with file name *pszPath* as current application.

CHZBciStartMultiAppEncryption

```
ZCBCIRET CHZBciStartMultiAppEncryption(ZCBCICARD card, WORD *pSW1SW2, BYTE *pAlgo,
BYTE key, WORD keyCID)
```

Starts encrypted communication with a MultiApplication BasicCard using the key with the number *key* as added with ZCBciSetKey or ZCBciReadKeyFile in the terminal and the matching key with the component ID *keyCID* on the card. Random values are generated internally.

CHZBciVerify

```
ZCBCIRET CHZBciVerify(ZCBCICARD card, WORD *pSW1SW2, WORD keyCID, const CHAR
*pszPassword)
```

Verifies that a user's password or PIN *pszPassword* matches the one stored on the card under the component ID *keyCID* of a given key.

CHZBciWriteComponentAttr (new in 0.2.0)

```
ZCBCIRET CHZBciWriteComponentAttr(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID, const
void *pAttr, BYTE cbAttr)
```

Writes the attribute field of the component with the ID *cmpCID* using data from the memory buffer pointed to by *pAttr* which has a size in bytes of *cbAttr*.
The format of a component's attribute and data fields depend on the type of the component and also varies between read and write operations. See below for structures that can be used as buffer and read the BasicCard user manual (section 5.8) for more details.

CHZBciWriteComponentData (new in 0.2.0)

```
ZCBCIRET CHZBciWriteComponentData(ZCBCICARD card, WORD *pSW1SW2, WORD cmpCID, const
void *pData, BYTE cbData)
```

Writes the data field of a component. The parameters have the same function as for CHZBciWriteComponentAttr.

Constants

SW1SW2 values

A lot of SW1SW2 return values are defined in chzbci.h. Most of them are only used in MultiApplication BasicCards. They have the same names and values as in the BasicCard user manual.

Encryption algorithms

```
CHZBCI_AESxxx, CHZBCI_EAX_AESxxx, CHZBCI_OMAC_AESxxx, CHZBCI_EC167
```

These are IDs for encryption algorithms not already defined in zcbci.h. xxx represents the key size in bits and can be 128, 192 or 256. Not all algorithms can be used for all purposes.

Component types

```
CHZBCI_COMP_FILE, CHZBCI_COMP_ACR, CHZBCI_COMP_PRIVILEGE, CHZBCI_COMP_FLAG,
CHZBCI_COMP_KEY
```

These are component types used for example in CHZBciFindComponent.

Key usage masks

```
CHZBCI_KU_VERIFY_MASK, CHZBCI_KU_EXTAUTH_MASK, CHZBCI_KU_SMENC_MASK,
CHZBCI_KU_SMMAC_MASK, CHZBCI_KU_SIGN_MASK, CHZBCI_KU_INTAUTH_MASK,
CHZBCI_KU_SECTRANS_MASK
```

Any sum of these values in the corresponding *usagemask* attribute field of a key defines for which purposes this key may be used on a MultiApplication BasicCard.

Algorithm masks

`CHZBCI_DES_CRC_MASK, CHZBCI_3DES_CRC_MASK, CHZBCI_AESxxx_MASK, CHZBCI_EAX_AESxxx_MASK, CHZBCI_OMAC_AESxxx_MASK, CHZBCI_EC167_MASK`

Any sum of these values in the corresponding *algorithmmask* attribute field of a key defines with which algorithms this key may be used on a MultiApplication BasicCard. xxx represents the key size in bits and may be 128, 192 or 256.

Miscellaneous constants

`CHZBCI_LOADSEQUENCE_START, CHZBCI_LOADSEQUENCE_END, CHZBCI_LOADSEQUENCE_ABORT`

To be used as *phase* parameter in CHZBciLoadSequence.

`CHZBCI_FLAG_PERMANENT, CHZBCI_FLAG_CLEAR_NEW_APP, CHZBCI_FLAG_CLEAR_COMMAND`

These are attributes to be assigned to a flag component.

`CHZBCI_VER`

The version of libbasiccard. Note that this is evaluated at compile time while CHZBciGetLibVersion returns the version at run time.

Typedefs of structures

Attribute field definitions

`CHZBCI_xxx_yyy_ATTR, CHZBCI_zzz_ATTR`

chzbci.h defines structures to handle the component's attribute fields. The contents of the attribute field of a component depends on the type of the component and in some cases also on whether the component is created or its attributes are read or written after creation.
Then xxx represents the component's type and can be DIRECTORY, DATAFILE or ACR. yyy represents the operation for which the attributes will be used: CREATE (when used for CHZBciCreateComponent), READ (for CHZBciReadComponentAttr) or WRITE (for CHZBciWriteComponentAttr). zzz represents the type of components which don't have different sets of attributes depending on the operation and can be PRIVILEGE, FLAG or KEY.

The fields inside this structures have the same names as in the Component Details section of the BasicCard user manual except the names are completely in lower case.

Special attention must be given to fields of the type WORD. To avoid problems due to different endiannesses these fields were defined as two fields of the type BYTE with the suffixes _HI and _LO appended to their names. Two macros were also defined to ease the access to WORD fields:

bci_getword

`bci_getword(x)`

Accesses the _HI and _LO parts of the WORD field *x* in a component attribute structure and returns the word contained in it.

bci_setword

`bci_setword(x,y)`

Sets the _HI and _LO parts of the WORD field *x* to the high and low byte of the word *y* respectively.

Examples

The following snippet was taken from an example that reads the attributes of the key with the previously determined component ID cmpCID:

```
CHZBCI_KEY_ATTR a;
BYTE len = sizeof(a);
/* ... */
CHZBciReadComponentAttr(card, &SW1SW2, cmpCID, &a, &len);
printf("acrcid=%.4x, usagemask=%.4x, algomask=%.4x, ec=%d, ecrst=%d\n",
       bci_getword(a.acrcid), bci_getword(a.usagemask),
       bci_getword(a.algorithmmask), a.errorcounter, a.ecresetvalue);
/* ... */
```

The following example initializes the attribute data field for the creation of a file:

```
CHZBCI_DATAFILE_CREATE_ATTR a;
BYTE len = sizeof(a);
bci_setword(a.acrcid, 0); /* No associated ACR */
a.attributes = 0; /* signaling a data file */
bci_setword(a.blocklen, 0x100);
/* last line is equivalent to a.blocklen_HI = 0x01; a.blocklen_LO = 0x00; */
/* ... */
```

BasicCard cryptography library

Starting with version 0.2.0 an implementation of the cryptographic API for BasicCards (named zccrypt by ZeitControl) is provided in form of an additional library called libbccrypto. It supports all of the official functions with the exception of elliptic curve cryptography (ECC) related ones and it also provides a number of unofficial functions (i.e. not existing in zccrypt) which are defined in chzcrypt.h and start with CHZCrypt... instead of ZCCrypt.... See the BasicCard Crypto API C/C++ Programming Manual for a description of the official functions. The unofficial functions are explained below.

Elliptic curve cryptography (ECC) support for ECC-161, ECC-167 and ECC-211 is available on request. Please contact me for further details.

Cryptography / Unofficial functions

Include chzcrypt.h to use these functions not defined in the official API.

EAX Functions

EAX is an encryption and authentication mode used together with the AES algorithm.

CHZCryptEAXStart

```
ZCCRYPTRET CHZCryptEAXStart(PCHZCRYPT_CONTEXT_EAX pEax, CHZCRYPT_ALGO algo, const
BYTE* pKey, const void* pNonce, DWORD cbNonce, const void* pHeader, DWORD cbHeader)
```

Creates a context for EAX encryption and authentication. This context is stored in a buffer of the type CHZCRYPT_CONTEXT_EAX pointed to by *pEax*. The algorithm *algo* may be one of CHZCRYPT_ALGO_EAXAESxxx (see below). *pKey* points to the key to be used, *pNonce* points to a nonce whose length in bytes is *cbNonce*. *pHeader* can point to an optional header (*cbHeader* bytes long). If *cbHeader* is zero, *pHeader* can be NULL.

CHZCryptEAXAddHeader

```
ZCCRYPTRET CHZCryptEAXAddHeader(CHZCRYPT_CONTEXT_EAX Eax, const void* pHeader, DWORD
cbHeader)
```

pHeader points to a header which is *cbHeader* bytes long and which will be added to the EAX context *Eax*.

CHZCryptEAXEncrypt

```
ZCCRYPTRET CHZCryptEAXEncrypt(CHZCRYPT_CONTEXT_EAX Eax, void* pData, DWORD cbData)
```

Encrypts *cbData* bytes of data pointed to by *pData* using the EAX context *Eax*. The ciphertext overwrites the plaintext at *pData*.

CHZCryptEAXDecrypt

```
ZCCRYPTRET CHZCryptEAXDecrypt(CHZCRYPT_CONTEXT_EAX Eax, void* pData, DWORD cbData)
```

Decrypts *cbData* bytes of data pointed to by *pData* using the EAX context *Eax*. The plaintext overwrites the ciphertext at *pData*.

CHZCryptEAXEnd

```
ZCCRYPTRET CHZCryptEAXEnd(PCHZCRYPT_CONTEXT_EAX pEax, void* pTag)
```

Ends a session of EAX encryption, invalidates the context pointed to by *pEax* and returns a 16 byte long tag in the memory buffer pointed to by *pTag* that can be used for authentication purposes.

CHZCryptEAXEncryptOnce

```
ZCCRYPTRET CHZCryptEAXEncryptOnce(CHZCRYPT_ALGO algo, const BYTE* pKey, const void*
pNonce, DWORD cbNonce, const void* pHeader, DWORD cbHeader, void* pData, DWORD
cbData, void* pTag)
```

Encrypts and authenticates a single chunk of data in one pass without the need for a context. All parameters have the same meanings as in the EAX functions above.

CHZCryptEAXDecryptOnce

```
ZCCRYPTRET CHZCryptEAXDecryptOnce(CHZCRYPT_ALGO algo, const BYTE* pKey, const void*
pNonce, DWORD cbNonce, const void* pHeader, DWORD cbHeader, void* pData, DWORD
cbData, void* pTag)
```

Decrypts and authenticates a single chunk of data in one pass without the need for a context. All parameters have the same meanings as in the EAX functions above.

OMAC functions

OMAC together with AES is an algorithm for message authentication.

CHZCryptOMACStart

```
ZCCRYPTRET CHZCryptOMACStart(PCHZCRYPT_CONTEXT_OMAC pOmac, CHZCRYPT_ALGO algo, const
BYTE* pKey)
```

Creates an OMAC context for later use and stores it in a buffer of the type CHZCRYPT_CONTEXT_OMAC. *algo* can be one of CHZCRYPT_ALGO_OMACAESxxx constants (see below), *pKey* points to the key to use.

CHZCryptOMACUpdate

```
ZCCRYPTRET CHZCryptOMACUpdate(CHZCRYPT_CONTEXT_OMAC Omac, const void* pData, DWORD
cbData)
```

Processes a *cbData* bytes long chunk pointed to by *pData* in the OMAC context *Omac* for authentication.

CHZCryptOMACEnd

```
ZCCRYPTRET CHZCryptOMACEnd(PCHZCRYPT_CONTEXT_OMAC pOmac, void* pTag)
```

Ends a session of OMAC, invalidates the context pointed to by *pOmac* and computes a 16 bytes long tag out of all data processed through CHZCryptOMACUpdate which is stored at a memory buffer pointed to by *pTag*.

SHA-256 functions

SHA-256 is a cryptographic hash function. See the original API documentation for information about SHA-1, the other supported hash function.

CHZCryptSha256Start

```
ZCCRYPTRET CHZCryptSha256Start(PZCCRYPT_CONTEXT_SHA pSHAContext)
```

Creates an SHA-256 context and stores in a buffer of the type ZCCRYPT_CONTEXT_SHA. Note that it is not possible to mix SHA-1 and SHA-256 function calls using the same context.

CHZCryptSha256Append

```
ZCCRYPTRET CHZCryptSha256Append(ZCCRYPT_CONTEXT_SHA SHAContext, const void *pData,
DWORD cbData)
```

Processes *cbData* bytes of data pointed to by *pData* using the context *SHAContext*.

CHZCryptSha256End

```
ZCCRYPTRET CHZCryptSha256End(PZCCRYPT_CONTEXT_SHA pSHAContext, void *pHash)
```

Hashes all data previously passed to CHZCryptSha256Append in the context *pSHAContext* and stores the resulting 32 byte long hash in the buffer pointed to by *pHash*. The context is subsequently invalidated.

CHZCryptSha256Once

```
ZCCRYPTRET CHZCryptSha256Once(const void *pData, DWORD cbData, void *pHash)
```

Hashes *cbData* bytes of data in *pData* and stores the resulting 32 byte long hash in the buffer *pHash*. No context needs to be created for this function call.

Miscellaneous functions

CHZCryptGetVer

```
ZCCRYPTRET CHZCryptGetVer(DWORD *pVer)
```

pVer points to a DWORD in which the version of libbccrypto will be stored in BCD format. 0x000wxxyz is thus decoded to version xx.y.z. Additionally w is 1 when ECC support is enabled in this version of libbccrypto and 0 otherwise. Currently either 0x00000025 or 0x00010025 is returned.

ECC functions

Elliptic curve cryptography support is not included by default. Contact the author of libbcrypto if you want to enable these functions. Additionally to the functions already defined in the official API the following ones are provided if ECC support is enabled:

CHZCryptEC167Sign

```
ZCCRYPTRET CHZCryptEC167Sign(ZCCRYPT_CONTEXT_EC167 Ec167, const
ZCCRYPT_EC167PRIVATEKEY *pPrivateKey, const void *pMessage, ZCCRYPT_EC167SIGNATURE
*pSignature)
```

Signs a 20 bytes long message (or a hash of that length) pointed to by *pMessage* using 167 bit ECC, given an EC167 context *Ec167* and a private key pointed to by *pPrivateKey.* The resulting signature is stored in a buffer of the type ZCCRYPT_EC167SIGNATURE pointed to by *pSignature*.

CHZCryptEC167SignAlg

```
ZCCRYPTRET CHZCryptEC167SignAlg(ZCCRYPT_CONTEXT_EC167 Ec167, const
ZCCRYPT_EC167PRIVATEKEY *pPrivateKey, const void *pMessage, ZCCRYPT_EC167SIGNATURE
*pSignature, CHZCRYPT_ALGO algo)
```

Signs a message as described in CHZCryptEC167Sign. The signature algorithm to be used is chosen by *algo*. CHZCRYPT_ALGO_ECC_NR selects the Nyberg-Rueppel algorithm supported by all BasicCards and also used by CHZCryptEC167Sign, CHZCRYPT_ALGO_ECC_DSA selects the Digital Signature Algorithm supported by many newer cards.

CHZCryptEC167HashAndSign

```
ZCCRYPTRET ZCCRYPTLINK CHZCryptEC167HashAndSign(ZCCRYPT_CONTEXT_EC167 Ec167, const
ZCCRYPT_EC167PRIVATEKEY *pPrivateKey, const void *pMessage, DWORD cbMessage,
ZCCRYPT_EC167SIGNATURE *pSignature)
```

pMessage points to a message which is *cbMessage* bytes long which is hashed and then signed as described in CHZCryptEC167Sign.

CHZCryptEC167HashAndSignAlg

```
ZCCRYPTRET ZCCRYPTLINK CHZCryptEC167HashAndSignAlg(ZCCRYPT_CONTEXT_EC167 Ec167, const
ZCCRYPT_EC167PRIVATEKEY *pPrivateKey, const void *pMessage, DWORD cbMessage,
ZCCRYPT_EC167SIGNATURE *pSignature, CHZCRYPT_ALGO algo)
```

Hashes and signs a message as described for CHZCryptEC167HashAndSign. Additionally, the algorithm is chosen by *algo*.

CHZCryptEC167VerifyAlg

```
ZCCRYPTRET CHZCryptEC167VerifyAlg(ZCCRYPT_CONTEXT_EC167 Ec167, const
ZCCRYPT_EC167SIGNATURE *pSignature, const ZCCRYPT_EC167PUBLICKEY *pPublicKey, const
void *pMessage, CHZCRYPT_ALGO algo)
```

Verifies a digital signature using the algorithm *algo*. See ZCCryptEC167Verify for information about the other function parameters.

CHZCryptEC167HashAndVerifyAlg

```
ZCCRYPTRET ZCCRYPTLINK CHZCryptEC167HashAndVerifyAlg(ZCCRYPT_CONTEXT_EC167 Ec167,
const ZCCRYPT_EC167SIGNATURE *pSignature, const ZCCRYPT_EC167PUBLICKEY *pPublicKey,
const void *pMessage, DWORD cbMessage,  CHZCRYPT_ALGO algo)
```

Hashes a message and verifies its digital signature using the algorithm algo. See ZCCryptEC167HashAndVerify for information about the other function parameters.

CHZCryptEC161Sign

```
ZCCRYPTRET CHZCryptEC161Sign(ZCCRYPT_CONTEXT_EC161 Ec161, const
ZCCRYPT_EC161PRIVATEKEY *pPrivateKey, const void *pMessage, ZCCRYPT_EC161SIGNATURE
*pSignature)
```

Signs a message using 161 bit ECC. See CHZCryptEC167Sign.

CHZCryptEC161HashAndSign

```
ZCCRYPTRET ZCCRYPTLINK CHZCryptEC161HashAndSign(ZCCRYPT_CONTEXT_EC161 Ec161, const
ZCCRYPT_EC161PRIVATEKEY *pPrivateKey, const void *pMessage, DWORD cbMessage,
ZCCRYPT_EC161SIGNATURE *pSignature)
```

Hashes and signs a message using 161 bit ECC. See CHZCryptEC167HashAndSign.

ECC-211 functions

The official BasicCard C API does not support 211 bit elliptical curve cryptography available in newer BasicCards. Libbccrypto, however, provides functions to use ECC-211. Like all ECC support, it is not included in libbccrypto by default but can be requested from the author.

The function calls are similar to those for 167 bit ECC. The following documentation only points out the differences.

CHZCryptEC211CreateContext

```
ZCCRYPTRET CHZCryptEC211CreateContext(PCHZCRYPT_CONTEXT_EC211 pEC211, int iCurve)
```

Creates a context using the predefined curve *iCurve*.

CHZCryptEC211CreateContextFromParams

```
ZCCRYPTRET CHZCryptEC211CreateContextFromParams(PCHZCRYPT_CONTEXT_EC211 pEC211, const
CHZCRYPT_EC211DOMAINPARAMS *pParams)
```

Creates a context from the given curve parameters *pParams*.

CHZCryptEC211DestroyContext

```
ZCCRYPTRET CHZCryptEC211DestroyContext(PCHZCRYPT_CONTEXT_EC211 pEC211)
```

Destroys the context.

CHZCryptEC211GetPublicKey

```
ZCCRYPTRET CHZCryptEC211GetPublicKey(CHZCRYPT_CONTEXT_EC211 Ec211,
PCHZCRYPT_EC211PRIVATEKEY pPrivateKey, PCHZCRYPT_EC211PUBLICKEY pPublicKey)
```

Gets the public key from a known private key. Note that keys are 27 bytes long for ECC-211.

CHZCryptEC211SetupPrivateKey

```
ZCCRYPTRET CHZCryptEC211SetupPrivateKey(CHZCRYPT_CONTEXT_EC211 Ec211,
PCHZCRYPT_EC211PRIVATEKEY pPrivateKey, void *pKeyData, DWORD cbKeyData)
```

Makes a valid private key from 27 bytes of binary data.

CHZCryptEC211GenerateKeyPair

```
ZCCRYPTRET CHZCryptEC211GenerateKeyPair(CHZCRYPT_CONTEXT_EC211 Ec211,
PCHZCRYPT_EC211PRIVATEKEY pPrivateKey, PCHZCRYPT_EC211PUBLICKEY pPublicKey, void
*pSeed, DWORD cbSeed)
```

Generates a new key pair (private key and public key), preferably seeding the random generator first.

CHZCryptEC211VerifyAlg

```
ZCCRYPTRET CHZCryptEC211VerifyAlg(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211SIGNATURE *pSignature, const CHZCRYPT_EC211PUBLICKEY *pPublicKey, const
void *pMessage, CHZCRYPT_ALGO algo)
```

Verifies a digital signature using either the Nyberg-Rueppel algorithm or the DSA as determined by *algo*. Note that signatures are 54 bytes long, while the message/hash pointed to by *pMessage* needs to be 32 bytes long.

CHZCryptEC211HashAndVerifyAlg

```
ZCCRYPTRET CHZCryptEC211HashAndVerifyAlg(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211SIGNATURE *pSignature, const CHZCRYPT_EC211PUBLICKEY *pPublicKey, const
void *pMessage, DWORD cbMessage, CHZCRYPT_ALGO algo)
```

Hashes a message using SHA-256 and verifies its signature.

CHZCryptEC211SignAlg

```
ZCCRYPTRET CHZCryptEC211SignAlg(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211PRIVATEKEY *pPrivateKey, const void *pMessage, CHZCRYPT_EC211SIGNATURE
*pSignature, CHZCRYPT_ALGO algo)
```

Signs a 32 byte long message/hash using the signature algorithm *algo*.

CHZCryptEC211HashAndSignAlg

```
ZCCRYPTRET CHZCryptEC211HashAndSignAlg(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211PRIVATEKEY *pPrivateKey, const void *pMessage, DWORD cbMessage,
CHZCRYPT_EC211SIGNATURE *pSignature, CHZCRYPT_ALGO algo)
```

Hashes a message using SHA-256 and signs it.

CHZCryptEC211SharedSecret

```
ZCCRYPTRET CHZCryptEC211SharedSecret(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211PRIVATEKEY *pPrivateKey, const CHZCRYPT_EC211PUBLICKEY *pPublicKey,
PCHZCRYPT_EC211SHAREDSECRET pSecret)
```

Calculates a shared secret given a private key and the public key of another party. Note that the shared secret is 27 bytes long.

CHZCryptEC211SessionKey

```
ZCCRYPTRET CHZCryptEC211SessionKey(CHZCRYPT_CONTEXT_EC211 Ec211, const
CHZCRYPT_EC211PRIVATEKEY *pPrivateKey, const CHZCRYPT_EC211PUBLICKEY *pPublicKey,
const void *pKdp, DWORD cbKdp, PCHZCRYPT_EC211SESSIONKEY pKey)
```

Calculates a 32 byte long session key.

Constants and typedefs

Encryption algorithms

```
CHZCRYPT_ALGO_EAXAESxxx, CHZCRYPT_ALGO_OMACAESxxx
```

These are the IDs to be used in the EAX and OMAC functions as parameter *algo*. xxx represents the key size in bits and can be 128, 192 or 256.

```
CHZCRYPT_ALGO_ECC_NR, CHZCRYPT_ALGO_ECC_DSA
```

IDs to be passed to CHZCryptEC...Alg, representing the Nyberg-Ryppel (NR) algorithm and the Digital Signature Algorithm (DSA) respectively.

Contexts

```
CHZCRYPT_CONTEXT_EAX, PCHZCRYPT_CONTEXT_EAX, CHZCRYPT_CONTEXT_OMAC,
PCHZCRYPT_CONTEXT_OMAC, CHZCRYPT_CONTEXT_EC211, PCHZCRYPT_CONTEXT_EC211
```

Contexts and pointers to them for EAX, OMAC and ECC-211.

ECC-211 types

```
CHZCRYPT_EC211DOMAINPARAMS, CHZCRYPT_EC211PRIVATEKEY, CHZCRYPT_EC211PUBLICKEY,
CHZCRYPT_EC211SIGNATURE, CHZCRYPT_EC211SHAREDSECRET, CHZCRYPT_EC211SESSIONKEY ...
```

... and the respective pointers PCHZCRYPT_EC211...

See the documentation of ECC-167 types for more information.

Miscellaneous

`CHZCRYPT_VER`

Version of the header files for the libbccrypto library.

`CHZCRYPT_VER_MASK`

Perform a bit wise AND operation between the DWORD returned by CHZCryptGetVer and this constant to get only the version number of the library in the lower WORD.

`CHZCRYPT_EC_SUPPORT`

Perform a bit wise AND operation between the DWORD returned by CHZCryptGetVer and this constant to see whether the library has ECC support.