a-sign Client

Developers Manual

Version: 1.8
Author: Franz Brandl

Table of Content

1 About this Document

1.1 Purpose

This document is the developer's manual for A-Trust's *a-sign Library*.

1.2 Abbreviations

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BCS	Basic Command Set
BER	Basic Encoding Rules
CT	CardTerminal
DER	Distinguished Encoding Rules
DES	Digital Encryption Standard
HTSI	Host Transport Service Interface
ISO	International Organization for Standardization
MBS	Multi Bank Standard
PIN	Personal Identification Number
PUK	Personal Unblocking Key

1.3 Bibliography

[1] RFC 2459, Internet X509 Public Key Infrastructure – Certificate and CRL Profile, Jänner 1999

[2] CT-API 1.1, Anwendungsunabhängiges **C**ard**T**erminal **A**pplication **P**rogramming **I**nterface für Chipkartenanwendungen, 15. April 1999

[3] CT-BCS, Anwendungsunabhängiger **C**ard**T**erminal **B**asic **C**ommand **S**et, 15. April 1999

[4] ELU Card Specification A-Trust, v3.8

[5] SPK Card Specification TRUSTSIGN, v1.2

[6] KOBIL Chipkartenterminal, v1.4

[7] PKCS#11 – Cryptographic Token Interface Standard

1.4 Document History

Version	Date	Changes
0.1	7.2.2002	First Draft
0.2	8.4.2002	Second Draft, adapted to Netscape Security Library
0.3	24.7.2002	Final Draft, after implementation
0.3.1	25.7.2002	Final Draft, additional information, typos etc…
0.4	3.10.2002	Adapted to a-sign library v1.03
1.0	24.4.2003	Adapted to a-sign library v1.04 Translated to english
1.01	13.5.2003	Clarification on *C_WaitForSlotEvent* implementation
1.1	30.7.2003	a-sign Uni card added description of PKCS#11 objects edited
1.2	9.9.2003	CryptoAPI description added
1.3	9.12.2003	adapted to a-sign Client version 1.0.4.7
1.4	14.5.2004	adapted to a-sign Client version 1.0.4.9
1.5	22.7.2004	bugfixes
1.6	16.8.2004	silent installation specified
1.7	11.4.2005	installation reader parameter modified
1.8	17.8.2005	CardNumber registry entry specified

1.5 Client History

Version	Date	Changes
1.0.4.5	26.8.2003	first customer release
1.0.4.7	11.12.2003	- PC/SC reader support added - optional PIN cache added for CSP - C_Login, C_Logout implemented - certificate and key objects renamed - a-sign JKU card implemented
1.0.4.9	14.5.2004	- CPSetProvParam with param PP_KEYEXCHANGE_PIN implemented - slot order changed (reader slots first) for Notes compliance - bugfixes in PIN change and unblock sample

		programs
1.1.0.0	12.11.2004	registry path moved (“a.sign Client” instead of “asign104”) UTIMACO 3621 reader added to, UTIMACO 2020 reader removed from setup
1.1.0.1	5.1.2005	- several fixes added - installation reader parameter format changed (from setup version “f”)
1.1.0.3	3.5.2005	- domain login capability added - dynamic PC/SC reader configuration feature added - display of PIN description added for readers equipped with displays
1.1.0.4	30.6.2005	- infobox default PIN behaviour changed
1.1.0.5	19.7.2005	- readers are no longer initialized on library load time - a.sign Plus added to setup

2 Overview

The *a-sign Library* implements common cryptographic interfaces under Microsoft Windows $^{\text{tm}}$.

The interfaces implemented are

- PKCS #11 Version 2.10
- Microsoft CryptoAPI 2.0

It supports the following crypto tokens:

- A-Trust TrustMark Card
- A-Trust TrustSign Card
- A-Trust a-sign Premium Card
- A-Trust a-sign Premium Uni Card
- A-Trust a-sign JKU Card

Access to the crypto tokens is implemented via the following smart card terminals:

- Cherry Smartboard G83-6700 or compatible
- KOBIL KAAN professional
- KOBIL B1 PCMCIA
- REINER SCT cyberJack KB
- REINER SCT cyberJack pinpad
- REINER SCT cyberJack e-com
- SCM/Towitoko SPR532
- SCM/Towitoko SCR 241 (PCMCIA)
- Siemens Sign@tor, Version 1.0
- UTIMACO CardMan 3621
- UTIMACO CardMan 8630
- any PC/SC reader

The PIN handling is implemented within the *a-sign Library*. PINs are requested from the user either via the reader's PIN pad (if present) or on the PC's keyboard.

The cryptographic methods implemented are

- PKCS#1 signing and verification of documents (signing of message digests)
- PKCS#1 encryption and decryption of documents (encryption and decryption of session keys)
- Calculation of message digests
- Generation of DES, 2DES und 3DES session keys
- encryption and decryption of documents (encryption and decryption using session keys)
- import of PKCS#1 public keys
- import of X.509 certificates

2.1 Installation

Installation of the *a-sign Library* is implemented as a Installshield^{tm} installation procedure.

Note: Installation has to be done from the current user.

2.1.1 Supported Operating Systems

a-sign Library supports the following versions of Microsoft Windows^{tm}:

- Windows 98, Second Edition, Internet Explorer 6.0
- Windows ME, Internet Explorer 6.0
- Windows NT 4.0 Workstation, Service Pack 6+, Internet Explorer 6.0
- Windows 2000 professional, Service Pack 2, Internet Explorer 6.0
- Windows XP professional, Internet Explorer 6.0

Note that not all of the suppoted card readers might work with all of the operating systems supported.

2.1.2 Supported Netscape Products

a-sign Library supports the following versions of Netscape:

- Netscape Communicator 4.78
- Netscape 7.0 and above
- Mozilla 1.2 and above

2.1.3 Prerequisites

The *a-sign library* may access the smart card terminal on demand and either over the CT-API or the PC/SC interface.

Since the CT-API interface ist not built for several applications accessing the reader at the same time, there may be NO applications running on the same PC which permanently access the smart card terminal (iD2 Personal, ...).

3 Microsoft CryptoAPI 2.0

a-sign Library is installed on the target system as a *Cryptographic Service Provider* (CSP). Therefore an application may use CryptoAPI standard commands to access the *a-sign library*.

Note that the *a-sign Library* is defined as a ‘full CSP’. Full CSP’s usually implement a broad range of functions as defined by Microsoft. Note that *a-sign Library* does **NOT** comply to MS’s requirements for a full CSP however – the definition is made that way to enable *a-sign Library* to work together with a couple of standard applications which require full CSP’s but do not use the full functionality.

The following chapters describe the functionality as implemented by *a-sign Library.*

3.1 Algorithms

Name	Algorithm ID	Min Length	Max Length
RC2	26114	40	128
DES	26113	56	56
3DES TWO KEY	26121	112	112
3DES	26115	168	168
SHA-1	32772	160	160
MD5	32771	128	128
SHA-1 MD5	32776	228	228
RSA_SIGN	9216	1024 / 1536 [1]	1024 / 1536
RSA_KEYX	41984	1024 / 1536	1024 / 1536

[1] depending on the card used

3.2 *CryptoAPI Functions*

3.2.1 General Functions

CryptAcquireContext

The CryptAcquireContext function is used to acquire a handle to a particular key container within a particular cryptographic service provider (CSP). This returned handle is used in calls to CryptoAPI functions that use the selected CSP.

a-sign Library implementation: *a-sign Library* supports the following modes of calling CryptAcquireContext:

- pszContainer set to NULL, dwFlags set to CRYPT_VERIFYCONTEXT:
- pszContainer set to the CIN (cardholder identification number) of the card associated to the certificate the application intends to use, dwFlags set to 0

CryptReleaseContext

The CryptReleaseContext function releases the handle of a cryptographic service provider (CSP) and a key container.

a-sign Library implementation: fully implemented.

CryptGetProvParam

The CryptGetProvParam function retrieves parameters that govern the operations of a cryptographic service provider (CSP).

a-sign Library implementation: *a-sign Library* supports the following modes of calling CryptGetProvParam:

- PP_CONTAINER
- PP_ENUMALGS
- PP_ENUMALGS_EX
- PP_ENUMCONTAINERS
- PP_IMPTYPE
- PP_NAME
- PP_VERSION

CryptSetProvParam

The CryptSetProvParam function customizes the operations of a cryptographic service provider (CSP).

a-sign Library implementation:

- parameter PP_KEYEXCHANGE_PIN supported to supply a PIN value to the internal PIN cache. The application should call the function again using a NULL PIN value to clear the cache as soon as the cached PIN is no longer needed.

3.2.2 Hash Functions

CryptCreateHash

The CryptCreateHash function initiates the hashing of a stream of data. It creates and returns to the calling application a handle to a CSP hash object. This handle is used in subsequent calls to CryptHashData and CryptHashSessionKey to hash session keys and other streams of data.

a-sign Library implementation:

- no support for keyed hashes -> hKey has to be 0.
- For supported algorithms see chapter 'Algorithms'.

CryptDestroyHash

The CryptDestroyHash function destroys the hash object referenced by the hHash parameter. After a hash object has been destroyed, it can no longer be used.

a-sign Library implementation: fully implemented.

CryptGetHashParam

The CryptGetHashParam function retrieves data that governs the operations of a hash object. The actual hash value can be retrieved by using this function.

a-sign Library implementation: fully implemented.

CryptHashData

The CryptHashData function adds data to a specified hash object. This function can be called multiple times to compute the hash of long or discontinuous data streams.

a-sign Library implementation:

- dwFlags is ignored

CryptHashSessionKey

The CryptHashSessionKey function computes the cryptographic hash of a session key object.

a-sign Library implementation: function not supported*.*

CryptSetHashParam

The CryptSetHashParam function customizes the operations of a hash object including setting up initial hash contents and selecting a specific hashing algorithm.

a-sign Library implementation: fully implemented.

3.2.3 Encryption and Decryption Functions

CryptDecrypt

The CryptDecrypt function decrypts data previously encrypted using the CryptEncrypt function.

a-sign Library implementation:

- simultaneously decryption and hashing of data not supported, therefore hHash must be 0.

CryptEncrypt

The CryptEncrypt function encrypts data. The algorithm used to encrypt the data is designated by the key held by the CSP module and is referenced by the hKey parameter.

a-sign Library implementation:

- simultaneously hashing and encryption of data not supported, therefore hHash must be 0.

3.2.4 Random Functions

CryptGenRandom

The CryptGenRandom function fills a buffer with cryptographically random bytes.

a-sign Library implementation: fully implemented.

3.2.5 Key Functions

CryptGenKey

The CryptGenKey function generates a random cryptographic session key or a public/private key pair. A handle to the key or key pair is returned in phKey. This handle can then be used as needed with any CryptoAPI function requiring a key handle.

a-sign Library implementation:

- only symmetrical session keys, no key pairs are supported.
- For supported algorithms see chapter ‘Algorithms’.

CryptDeriveKey

The CryptDeriveKey function generates cryptographic session keys derived from a base data value.

a-sign Library implementation: function not supported.

CryptSetKeyParam

The CryptSetKeyParam function customizes various aspects of a session key's operations. The values set by this function are not persisted to memory and can only be used with in a single session.

a-sign Library implementation:

- dwParam KP_MODE_BITS not supported.

CryptGetKeyParam

The CryptGetKeyParam function retrieves data that governs the operations of a key.

a-sign Library implementation:

- dwParam KP_MODE_BITS not supported.

CryptExportKey

The CryptExportKey function exports a cryptographic key or a key pair from a cryptographic service provider (CSP) in a secure manner.

a-sign Library implementation:

- only key blobs SIMPLEBLOB (for symmetric session keys) and PUBLICKEYBLOB (for public keys) are supported.
- dwFlags is not supported, must be 0.

CryptImportKey

The CryptImportKey function transfers a cryptographic key from a key BLOB into a cryptographic service provider (CSP).

a-sign Library implementation:

- only key blobs SIMPLEBLOB (for symmetric session keys) and PUBLICKEYBLOB (for public keys) are supported.
- dwFlags is not supported, must be 0.

CryptDestroyKey

The CryptDestroyKey function releases the handle referenced by the hKey parameter. After a key handle has been released, it becomes invalid and cannot be used again.

a-sign Library implementation: fully implemented.

CryptGetUserKey

The CryptGetUserKey function retrieves a handle of one of a user's two public/private key pairs.

a-sign Library implementation: fully implemented.

3.2.6 Signing and Verification Functions

CryptSignHash

The CryptSignHash function signs data. Because all signature algorithms are asymmetric and thus slow, the CryptoAPI does not allow data to be signed directly. Instead, data is first hashed and CryptSignHash is used to sign the hash.

a-sign Library implementation:

- dwKeySpec is not supported.
- sDescription is not supported, must be NULL.
- dwFlags is not supported, must be 0.

CryptVerifySignature

The CryptVerifySignature function verifies the signature of a hash object.

a-sign Library implementation:

- sDescription is not supported. Must be set to NULL.
- dwFlags is not supported, must be 0.

3.3 PIN Cache

The *a-sign Library* may optionally cache the decryption PIN after it has been entered by the cardholder for the first time (during a private key operation).

Alternatively, the PIN may be supplied to the PIN cache using the CryptSetProvParam method.

To switch on PIN cacheing, a registry key has to be set (see chapter ‘CSP Configuration’). By default, this key is not set and PIN cacheing is switched off.

The PIN cache is emptied whenever the calling application terminates or the *CryptSetProvParam* function is called using the PP_KEYEXCHANGE_PIN parameter and a NULL pin value.

4 PKCS#11 Module

The *a-sign Library* PKCS#11 module is implemented as a WIN32 Dynamic Link Library (DLL).
The DLL is named **asignp11.dll** and is installed in the Windows System(32) directory.

4.1 PKCS#11 Structures

4.1.1 Library Info

Description

The structure CK_INFO contains common information about the PKCS#11 module.

Content

Name	**Content**	**Description**
cryptokiVersion	2.10	version of PKCS#11 specification
manufacturerID	„A-Trust“	library manufacturer
flags	0	RFU
libraryDescription	„A-Trust a-sign Client “	description of library
libraryVersion	x.x	version number of library

4.1.2 Slot Info

Description

The structure CK_SLOT_INFO contains information about the smart card terminal assigned to the respective slot.

The *a-sign Library* implements two kinds of slots:

- ❑ A-Trust Root Store (optional) and
- ❑ one slot for each attached smart card terminal

Note that the slot sequence may vary due to version and configuration issues. Do NOT “hard code” a slot number in your application.

Content

A-Trust Root Store

Name	Content	Description
slotDescription	„A-Trust Root Store“	name of Root Store slot
manufacturerID	„A-Trust“	name of manufacturer
flags	CKF_TOKEN_PRESENT	token is always present
hardwareVersion	1.0	‚hardware version’ of slot
firmwareVersion	x.x	‚firmware version’ of slot

Note that the A-Trust Root Store slot may not be present due to individual registry settings.

Smart card terminal slot

Name	Content	Description
slotDescription	e.g. „KOBIL KAAN professional“	name of smart card terminal
manufacturerID	e.g. „KOBIL Systems“ or “Unbekannt”	name of reader manufacturer
Flags	CKF_REMOVABLE_DEVICE CKF_HW_SLOT CKF_TOKEN_PRESENT (set if card inserted)	• token may be removed from terminal • token is implemented in hardware
hardwareVersion	e.g. 2.8	firmware version of smart card terminal , if applicable
firmwareVersion	x.x	version number of CT-API DLL or driver, if applicable

4.1.3 Token Info

Description

The structure CK_TOKEN_INFO contains information about either the virtual ‚token' inserted in the A-Trust Root Store or the signature card (=PKCS#11 Token) inserted in the respective smart card terminal.

Content

A-Trust Root Store

Name	Content	Description
label	„A-Trust Root Store"	token name
manufacturerID	„A-Trust"	token manufacturer
model	„unbekannt"	token model
serialNumber	„unbekannt"	
flags	CKF_WRITE_PROTECTED CKF_TOKEN_INITIALIZED	• token is write protected • PINs are initialized
ulMaxSessionCount	CK_EFFECTIFELY_INFINITE	max. number of PKCS#11 sessions
ulSessionCount	X	number of current PKCS#11 sessions
ulMaxRWSessionCount	CK_UNAVAILABLE_ INFORMATION	max. number of PKCS#11 RW Sessions
ulRWSessionCount	CK_UNAVAILABLE_ INFORMATION	number of current PKCS#11 RW sessions
ulMaxPINLength	CK_UNAVAILABLE_ INFORMATION	max. PIN length in bytes
ulMinPINLength	CK_UNAVAILABLE_ INFORMATION	min. PIN length in bytes
ulTotalPublicMemory	CK_UNAVAILABLE_ INFORMATION	total amount of memory for public PKCS#11 objects on token
ulFreePublicMemory	CK_UNAVAILABLE_ INFORMATION	amount of free memory for public PKCS#11 objects on token
ulTotalPrivateMemory	CK_UNAVAILABLE_ INFORMATION	total amount of memory for private PKCS#11 objects on token
ulFreePrivateMemory	CK_UNAVAILABLE_ INFORMATION	amount of free memory for private PKCS#11 objects on token
hardwareVersion	1.0	token hardware version

firmwareVersion	1.0	token firmware version
utcTime	--	token time

a-sign token / TrustMark Card

Name	Content	Description
label	„TrustMark“	token name
manufacturerID	„A-Trust“	token manufacturer
model	“Starcos SPK 2.2+“	token model
serialNumber	„xxxxxx“	card number
flags	CKF_RNG CKF_TOKEN_INITIALIZED CKF_SECONDARY_AUTH	• token has random number generator • PINs are initialized • library implements PIN handling
ulMaxSessionCount	CK_EFFECTIFELY_INFINITE	max. number of PKCS#11 sessions
ulSessionCount	X	number of current PKCS#11 sessions
ulMaxRWSessionCount	CK_UNAVAILABLE_INFORMATION	max. number of PKCS#11 RW Sessions
ulRWSessionCount	CK_UNAVAILABLE_INFORMATION	number of current PKCS#11 RW sessions
ulMaxPINLength	8	max. PIN length in bytes
ulMinPINLength	4	min. PIN length in bytes
ulTotalPublicMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for public PKCS#11 objects on token
ulFreePublicMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for public PKCS#11 objects on token
ulTotalPrivateMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for private PKCS#11 objects on token
ulFreePrivateMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for private PKCS#11 objects on token
hardwareVersion	1.0	token hardware version
firmwareVersion	1.0	token firmware version
utcTime	--	token time

a-sign Premium / TrustSign Card

Name	Content	Description
label	„TrustSign“	token name
manufacturerID	„A-Trust“	token manufacturer
model	“Starcos SPK 2.3“	token model
serialNumber	„xxxxxx“	card number
flags	CKF_RNG CKF_TOKEN_INITIALIZED CKF_SECONDARY_AUTH	• token has random number generator • PINs are initialized • library implements PIN handling
ulMaxSessionCount	CK_EFFECTIFELY_INFINITE	max. number of PKCS#11 sessions
ulSessionCount	x	number of current PKCS#11 sessions
ulMaxRWSessionCount	CK_UNAVAILABLE_INFORMATION	max. number of PKCS#11 RW Sessions
ulRWSessionCount	CK_UNAVAILABLE_INFORMATION	number of current PKCS#11 RW sessions
ulMaxPINLength	CK_UNAVAILABLE_INFORMATION	max. PIN length in bytes
ulMinPINLength	CK_UNAVAILABLE_INFORMATION	min. PIN length in bytes
ulTotalPublicMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for public PKCS#11 objects on token
ulFreePublicMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for public PKCS#11 objects on token
ulTotalPrivateMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for private PKCS#11 objects on token
ulFreePrivateMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for private PKCS#11 objects on token
hardwareVersion	1.0	token hardware version
firmwareVersion	1.0	token firmware version
utcTime	--	token time

a-sign Premium University Card

Name	Content	Description
label	„a-sign Uni“	token name
manufacturerID	„A-Trust“	token manufacturer
model	“CARDOS M4.01“	token model
serialNumber	„xxxxxx“	card number
flags	CKF_RNG CKF_TOKEN_INITIALIZED CKF_SECONDARY_AUTH	• token has random number generator • PINs are initialized • library implements PIN handling
ulMaxSessionCount	CK_EFFECTIFELY_INFINITE	max. number of PKCS#11 sessions
ulSessionCount	x	number of current PKCS#11 sessions
ulMaxRWSessionCount	CK_UNAVAILABLE_INFORMATION	max. number of PKCS#11 RW Sessions
ulRWSessionCount	CK_UNAVAILABLE_INFORMATION	number of current PKCS#11 RW sessions
ulMaxPINLength	CK_UNAVAILABLE_INFORMATION	max. PIN length in bytes
ulMinPINLength	CK_UNAVAILABLE_INFORMATION	min. PIN length in bytes
ulTotalPublicMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for public PKCS#11 objects on token
ulFreePublicMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for public PKCS#11 objects on token
ulTotalPrivateMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for private PKCS#11 objects on token
ulFreePrivateMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for private PKCS#11 objects on token
hardwareVersion	1.0	token hardware version
firmwareVersion	1.0	token firmware version
utcTime	--	token time

a-sign JKU University Card

Name	Content	Description
label	„a-sign JKU“	token name
manufacturerID	„A-Trust“	token manufacturer
model	“ACOS EMV-A01“	token model
serialNumber	„xxxxxx“	card number
flags	CKF_RNG CKF_TOKEN_INITIALIZED CKF_SECONDARY_AUTH	• token has random number generator • PINs are initialized • library implements PIN handling
ulMaxSessionCount	CK_EFFECTIFELY_INFINITE	max. number of PKCS#11 sessions
ulSessionCount	x	number of current PKCS#11 sessions
ulMaxRWSessionCount	CK_UNAVAILABLE_INFORMATION	max. number of PKCS#11 RW Sessions
ulRWSessionCount	CK_UNAVAILABLE_INFORMATION	number of current PKCS#11 RW sessions
ulMaxPINLength	CK_UNAVAILABLE_INFORMATION	max. PIN length in bytes
ulMinPINLength	CK_UNAVAILABLE_INFORMATION	min. PIN length in bytes
ulTotalPublicMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for public PKCS#11 objects on token
ulFreePublicMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for public PKCS#11 objects on token
ulTotalPrivateMemory	CK_UNAVAILABLE_INFORMATION	total amount of memory for private PKCS#11 objects on token
ulFreePrivateMemory	CK_UNAVAILABLE_INFORMATION	amount of free memory for private PKCS#11 objects on token
hardwareVersion	1.0	token hardware version
firmwareVersion	1.0	token firmware version
utcTime	--	token time

4.2 *PKCS#11 Mechanisms*

4.2.1 RSA using PKCS#1

The CKM_RSA_PKCS mechanism is used for signing of data or encryption and decryption of keys.

CK_MECHANISM_INFO

Name	**Content**	**Description**
ulMinKeySize	512	min. key size in bit
ulMaxKeySize	1024	max. key size in bit
flags	CKF_HW CKF_WRAP CKF_UNWRAP CKF_SIGN CKF_VERIFY	mechanism is implemented in hardware and may be used for signing and encryption (wrapping) and decryption (unwrapping) of keys.

The CKM_SHA1_RSA_PKCS mechanism is used for encryption and decryption of data.

CK_MECHANISM_INFO

Name	**Content**	**Description**
ulMinKeySize	512	min. key size in bit
ulMaxKeySize	1024	max. key size in bit
flags	CKF_HW CKF_WRAP CKF_UNWRAP CKF_SIGN CKF_VERIFY	mechanism is implemented in hardware and may be used for signing and encryption (wrapping) and decryption (unwrapping) of keys.

4.2.2 SHA-1 Message Digest

The CKM_SHA1 mechanism is used to calculate a SHA-1 message digest over supplied data.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	CKM_UNAVAILABLE_ INFORMATION	min. key size in bit
ulMaxKeySize	CKM_UNAVAILABLE_ INFORMATION	max. key size in bit
flags	CKF_DIGEST	mechanism may be used to calculate a message digest.

4.2.3 DES Key Generation

The CKM_DES_KEY_GEN mechanism is used to generate a DES session key.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	56	min. key size in bit
ulMaxKeySize	56	max. key size in bit
flags	CKF_GENERATE	mechanism may be used to generate a key.

4.2.4 2DES Key Generation

The CKM_DES2_KEY_GEN mechanism is used to generate a 2DES (‘double DES’) session key.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	112	min. key size in bit
ulMaxKeySize	112	max. key size in bit
flags	CKF_GENERATE	mechanism may be used to generate a key.

4.2.5 3DES Key Generation

The CKM_DES3_KEY_GEN mechanism is used to generate a 3DES (‘triple DES’) session key.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	168	min. key size in bit
ulMaxKeySize	168	max. key size in bit
flags	CKF_GENERATE	mechanism may be used to generate a key.

4.2.6 DES Encryption / Decryption

The CKM_DES_CBC_PAD mechanismus is used to DES encrypt and decrypt data.
The DES algorithm is applied in CBC chaining mode.
On encryption, the data is padded using PKCS#1.
On decryption, the PKCS#1 padding is removed from the data.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	56	min. key size in bit
ulMaxKeySize	56	max. key size in bit
flags	CKF_ENCRYPT CKF_DECRYPT	mechanism may be used for encryption and decryption of data.

4.2.7 2DES und 3DES Encryption / Decryption

The CKM_DES3_CBC_PAD mechanism is used to 2DES and 3DES encrypt and decrypt data.
The 3DES algorithm is applied in CBC chaining mode.
On encryption, the data is padded using PKCS#1.
On decryption, the PKCS#1 padding is removed from the data.

CK_MECHANISM_INFO

Name	Content	Description
ulMinKeySize	112	min. key size in bit
ulMaxKeySize	168	max. key size in bit
flags	CKF_ENCRYPT CKF_DECRYPT	mechanism may be used for encryption and decryption of data.

4.3 PKCS#11 Objects

4.3.1 Certificates

During installation, the A-Trust a-sign Client retrieves all required root certificates from an A-Trust server. They can be accessed via the A-Trust Root Store Token.

Smart card certificates can be accessed via the smart card reader slots and the respective tokens

Certificates

Name	Content	Description
CKA_CLASS	CKO_CERTIFICATE	type of PKCS#11 object
CKA_TOKEN	True	objekt is a token object
CKA_PRIVATE	False	object is public accessible
CKA_MODIFIABLE	False	object may not be changed
CKA_LABEL	CA Name (Root Store certificates) „C.CH.EKEY“ (card encryption certificate) “C.CH.SIG” (card signature certificate)	name of object
CKA_CERTIFICATE_TYPE	CKC_X_509	type of certificate
CKA_SUBJECT	xxxxxxx	ASN.1 DER coded content of certificate subject
CKA_ISSUER	xxxxxxx	ASN.1 DER coded content of certificate issuer
CKA_SERIAL_NUMBER	xxxxxxx	ASN.1 DER coded content of certificate serial number
CKA_VALUE	xxxxxxx	ASN.1 DER coded certificate
CKA_ID	xxxxxxx	PKCS#11 ID of certificate. This ID can be used to tie together public and private keys and certificates.
CKA_NETSCAPE_EMAIL	xxxxxxx	certificate extension SubjectAltName (rfc822 e-mail address), if available

4.3.2 Public Keys

Encryption Key

Name	Content	Description
CKA_CLASS	CKO_PUBLIC_KEY	type of PKCS#11 object
CKA_TOKEN	true	objekt is a token object
CKA_PRIVATE	false	object is public accessible
CKA_MODIFIABLE	false	object may not be changed
CKA_LABEL	„PK.CH.EKEY“ (card encryption key) “PK.CH.SIG” (card signature key, may not be accessible)	name of object
CKA_KEY_TYPE	CKK_RSA	type of key
CKA_DERIVE	false	no masterkey
CKA_LOCAL	false	key was not generated by *a-sign Library*
CKA_ID	xxxxxx	PKCS#11 ID of key. This ID can be used to tie together public and private keys and certificates.
CKA_KEY_GEN_MECHANISM	CKM_UNAVAILABLE_ INFORMATION	mechanism used to generate key
CKA_ENCRYPT	false	key does not support encryption
CKA_VERIFY	true	key does support signature verification
CKA_VERIFY_RECOVER	false	key does not support signature verification with message recovery
CKA_WRAP	true	key does support key wrapping
CKA_MODULUS	card data from IPF	modulus value of key
CKA_MODULUS_BITS	card data from IPF	modulus length of key
CKA_PUBLIC_EXPONENT	card data from IPF	public exponent value of key

4.3.3 Private Keys

Decryption Key

Name	Content	Description
CKA_CLASS	CKO_PRIVATE_KEY	type of PKCS#11 object
CKA_TOKEN	true	objekt is a token object
CKA_PRIVATE	false	object is public accessible
CKA_MODIFIABLE	false	object may not be changed
CKA_LABEL	„SK.CH.EKEY“	name of object
CKA_KEY_TYPE	CKK_RSA	type of key
CKA_DERIVE	false	no masterkey
CKA_LOCAL	false	key was not generated by *a-sign Library*
CKA_ID	xxxxxxx	PKCS#11 ID of key. This ID can be used to tie together public and private keys and certificates.
CKA_KEY_GEN_MECHANISM	CKM_UNAVAILABLE_ INFORMATION	mechanism used to generate key
CKA_DECRYPT	false	key does not support encryption
CKA_SIGN	true	key does support signing
CKA_SIGN_RECOVER	false	key does not support signing with message recovery
CKA_UNWRAP	true	key supports key wrapping
CKA_SENSITIVE	true	key is secret
CKA_EXTRACTABLE	false	key may not be exported from token
CKA_ALWAYS_SENSITIVE	true	key is always secret
CKA_NEVER_EXTRACTABLE	true	key may never be exported from token
CKA_SECONDARY_AUTH	RFU	setting this attribute causes the assigned PIN to be changed
CKA_AUTH_PIN_FLAGS	RFU	setting this attribute causes the assigned PIN to be unlocked

Signature Key

Name	**Content**	**Description**
CKA_CLASS	CKO_PRIVATE_KEY	type of PKCS#11 object
CKA_TOKEN	true	objekt is a token object
CKA_PRIVATE	false	object is public accessible
CKA_MODIFIABLE	false	object may not be changed
CKA_LABEL	“SK.CH.SIG”	name of object
CKA_KEY_TYPE	CKK_RSA	type of key
CKA_DERIVE	false	no masterkey
CKA_LOCAL	false	key was not generated by *a-sign Library*
CKA_ID	xxxxxxx	PKCS#11 ID of key. This ID can be used to tie together public and private keys and certificates.
CKA_KEY_GEN_MECHANISM	CKM_UNAVAILABLE_ INFORMATION	mechanism used to generate key
CKA_DECRYPT	false	key does not support encryption
CKA_SIGN	false	key does not support signing
CKA_SIGN_RECOVER	false	key does not support signing with message recovery
CKA_UNWRAP	false	key does not support key wrapping
CKA_SENSITIVE	true	key is secret
CKA_EXTRACTABLE	false	key may not be exported from token
CKA_ALWAYS_SENSITIVE	true	key is always secret
CKA_NEVER_EXTRACTABLE	true	key may never be exported from token
CKA_SECONDARY_AUTH	RFU	setting this attribute causes the assigned PIN to be changed
CKA_AUTH_PIN_FLAGS	RFU	setting this attribute causes the assigned PIN to be unlocked

Note that the private signature key is accessible only for PIN handling (PIN change and PIN reset). Therefore, all operational flags (CKA_SIGN, ...) are set to FALSE.

4.4 PKCS#11 Functions

4.4.1 General purpose functions

C_Initialize
Fully implemented for argument pInitArgs = NULL. For argument pInitArgs <> NULL, points 1, 2 and 4 are implemented.

C_Finalize
Fully implemented.

C_GetInfo
Fully implemented.

C_GetFunktionList
Fully implemented.

4.4.2 Slot and token management functions

C_GetSlotList
Fully implemented.

C_GetSlotInfo
Fully implemented.

C_GetTokenInfo
Fully implemented.

C_WaitForSlotEvent
Only non blocking mode (flags = CKF_DONT_BLOCK) implemented.

C_GetMechanismList
Fully implemented.

C_GetMechanismInfo
Fully implemented.

C_InitToken
Not implemented.

C_InitPIN
Not implemented.

C_SetPIN
Not implemented.

4.4.3 Session managment functions

C_OpenSession
Fully implemented.

C_CloseSession
Fully implemented.

C_CloseAllSessions
Fully implemented.

C_GetSessionInfo
Fully implemented.

C_GetOperationState
Not implemented.

C_SetOperationState
Not implemented.

C_Login
Fully implemented.

C_Logout
Fully implemented.

4.4.4 Object managment functions

C_CreateObject
This function can be used to create RSA public key objects as well as X.509 certificate objects. In case of the creation of certificate objects, the public key is extracted from the certificate and also created as an object. Public key and certificate are linked together via the CKA_ID attribute.

Note: all created objects are session objects.

C_CopyObject
Not implemented.

C_DestroyObject
This function can be used to delete session objects.

C_GetObjectSize
Not implemented.

C_GetAttributeValue
Fully implemented.

C_SetAttributeValue
Partially implemented. The attributes CKA_SECONDARY_AUTH and CKA_AUTH_PIN_FLAGS can be set.

C_FindObjectsInit
Fully implemented.

C_FindObjects
Fully implemented.

C_FindObjectsFinal
Fully implemented.

4.4.5 Encryption functions

C_EncryptInit
Fully implemented.

C_Encrypt
Not implemented.

C_EncryptUpdate
Fully implemented.

C_EncryptFinal
Fully implemented.

4.4.6 Decryption functions

C_DecryptInit
Fully implemented.

C_Decrypt
Not implemented.

C_DecryptUpdate
Fully implemented.

C_DecryptFinal
Fully implemented.

4.4.7 Message digesting functions

C_DigestInit
Fully implemented.

C_Digest
Fully implemented.

C_DigestUpdate
Fully implemented.

C_DigestFinal
Fully implemented.

C_DigestKey
Not implemented.

4.4.8 Signing and MACing functions

C_SignInit
Fully implemented.

C_Sign
Fully implemented.

C_SignUpdate
Not implemented.

C_SignFinal
Not implemented.

C_SignRecoverInit
Not implemented.

C_SignRecover
Not implemented.

4.4.9 Functions for verifying signatures and MACs

C_VerifyInit
Fully implemented.

C_Verify
Fully implemented.

C_VerifyUpdate
Not implemented.

C_VerifyFinal
Not implemented.

C_VerifyRecoverInit
Not implemented.

C_VerifyRecover
Not implemented.

4.4.10 Dual-function cryptographic functions

Not implemented.

4.4.11 Key managment functions

C_GenerateKey
This function can be used to generate symmetric keys.
Note: all generated keys are session objects.

C_GenerateKeyPair
Not implemented.

C_WrapKey
Fully implemented.

C_UnwrapKey
Fully implemented.
Note: all generated keys are session objects.

C_DeriveKey
Not implemented.

4.4.12 Random number generation functions

C_SeedRandom
Not implemented.

C_GenerateRandom
Fully implemented. The function uses the smart card's random number generator.

4.4.13 Parallel function managment functions

Not implemented.

4.4.14 Callback functions

Not implemented.

Appendix

Appendix A - Sample Programs

This sample programs shows how the *a-sign Library* PKCS#11 module can be used to generate and verify signatures, encrypt and decrypt data, and to manage PINs and PUKs.

Signature Calculation

```
CK_RV                rv;
CK_ULONG             ulSlotCount;
CK_SLOT_ID_PTR       pSlotList;
CK_SLOT_INFO_PTR     pSlotInfo;
CK_SLOT_ID           SlotToUse;
CK_TOKEN_INFO        token;
CK_SESSION_HANDLE    session;
CK_OBJECT_HANDLE     key;
CK_MECHANISM         mechanism_sign = {CKM_SHA1_RSA_PKCS, NULL_PTR, 0};
CK_BYTE              signature[256];
CK_ULONG             signatureLength=sizeof(signature);
CK_ULONG             ObjectCount;
CK_OBJECT_CLASS      keyClass = CKO_PRIVATE_KEY;
CK_BYTE              TRUE = true;
CK_ATTRIBUTE         keytemplate[] = {
                           { CKA_CLASS, &keyClass, sizeof(keyClass) },
                           { CKA_SIGN, &TRUE, sizeof(TRUE) }

                     };
/*
      In content steht das null-terminierte zu signierende Dokument.
      Die Applikation muß dafür sorgen, daß der Zeiger gültig ist.
*/
CK_BYTE_PTR          content;
int                  i;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
```

```
      pSlotInfo =
            (CK_SLOT_INFO_PTR)
            malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
      for ( i = 0; i < ulSlotCount; i++) {
            rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
            assert (rv == CKR_OK);
      }
      /*
            Hier muß der Benutzer den gewünschten Slot (=Leser)
            auswählen.
            In SlotToUse muß dannach die zu benutzende ID stehen
            Weiters muß der Benutzer aufgefordert werden eine karte in
            den Leser zu stecken
      */
      rv = C_GetTokenInfo( SlotToUse, &token);
      if (rv == CKR_OK) {
            if(token.flags & CKF_LOGIN_REQUIRED)
            == 0) {
                  rv = C_OpenSession(SlotToUse,
                        CKF_SERIAL_SESSION, NULL, NULL, &session);
                  assert (rv == CKR_OK);
                  /*
                        suchen des Verschlüsselungsschlüssels
                  */
                  rv = C_FindObjectsInit(session, &keytemplate,
                        2);
                  if (rv == CKR_OK) {
                        rv = C_FindObjects(session, &key, 1,
                              &ObjectCount);
                        assert (rv == CKR_OK);

                        rv = C_FindObjectsFinal(session);
                        assert (rv == CKR_OK);

                        rv = C_SignInit(session, &mechanism_sign,
                        key);
                        assert (rv == CKR_OK);

                        rv = C_Sign(session, content, strlen(content),
                              signature, &signaturelength);
                        assert (rv == CKR_OK);
                        /*
                              jetzt ist die Signatur erstellt
                        */
                  }
                  C_CloseSession(session);
            }
      }
      free(pSlotInfo);
}
```

```
        free(pSlotList);
}

C_Finalize( NULL );
```

Signature Verification

```
CK_RV             rv;
CK_ULONG          ulSlotCount;
CK_SLOT_ID_PTR    pSlotList;
CK_SLOT_INFO_PTR  pSlotInfo;
CK_SLOT_ID        SlotToUse;
CK_TOKEN_INFO     token;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE  key;
CK_MECHANISM      mechanism_sign = {CKM_SHA1_RSA_PKCS, NULL_PTR, 0};
CK_BYTE           signature[256];
CK_ULONG          signatureLength=sizeof(signature);
CK_ULONG          ObjectCount;
CK_OBJECT_CLASS   keyClass = CKO_PUBLIC_KEY;
CK_BYTE           TRUE = true;
CK_ATTRIBUTE      keytemplate[] = {
                        { CKA_CLASS, &keyClass, sizeof(keyClass) },
                        { CKA_VERIFY, &TRUE, sizeof(TRUE) }

                  };
/*
      In content steht das null-terminierte zu signierende Dokument.
      Die Applikation muß dafür sorgen, daß der Zeiger gültig ist.
*/
CK_BYTE_PTR       content;
int               i;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
```

```
                  Weiters muß der Benutzer aufgefordert werden eine karte in
                  den Leser zu stecken
            */
            rv = C_GetTokenInfo( SlotToUse, &token);
            if (rv == CKR_OK) {
                  if(token.flags & CKF_LOGIN_REQUIRED)
                  == 0) {
                        rv = C_OpenSession(SlotToUse,
                              CKF_SERIAL_SESSION, NULL, NULL, &session);
                        assert (rv == CKR_OK);
                        /*
                              suchen des Verschlüsselungsschlüssels
                        */
                        rv = C_FindObjectsInit(session, &keytemplate,
                              2);
                        if (rv == CKR_OK) {
                              rv = C_FindObjects(session, &key, 1,
                                    &ObjectCount);
                              assert (rv == CKR_OK);

                              rv = C_FindObjectsFinal(session);
                              assert (rv == CKR_OK);

                              rv = C_VerifyInit(session, &mechanism_sign,
                              key);
                              assert (rv == CKR_OK);

                              rv = C_Verify(session, content,
                                    strlen(content), signature,
                                    &signaturelength);
                              assert (rv == CKR_OK);
                              /*
                                    jetzt ist die Signatur verifiziert
                              */
                        }
                        C_CloseSession(session);
                  }
            }
            free(pSlotInfo);
      }
      free(pSlotList);
}

C_Finalize( NULL );
```

Encryption of Data using a Session Key

```
CK_RV               rv;
CK_ULONG            ulSlotCount;
CK_SLOT_ID_PTR      pSlotList;
CK_SLOT_INFO_PTR    pSlotInfo;
CK_SLOT_ID          SlotToUse;
CK_TOKEN_INFO       token;
CK_SESSION_HANDLE   session;
CK_OBJECT_HANDLE    key;
CK_BYTE             iv[8];
CK_MECHANISM        mechanism_gen_key = {CKM_DES_KEY_GEN, NULL_PTR, 0};
CK_MECHANISM        mechanism_use_key = {CKM_DES_CBC_PAD, &iv, sizeof(iv)};
char                label[]=”DES Key”;
CK_ATTRIBUTE        temp={CKA_LABEL, &label, strlen(label)};
int                 i;
CK_ULONG            count;
CK_BYTE_PTR         encryptedBuffer;

/* In Content steht der nullterminierte Content der verschlüsselt werden
soll.
Muß valid sein.*/
CK_BYTE_PTR         content;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
                  Weiters muß der Benutzer aufgefordert werden eine karte in
                  den Leser zu stecken
            */
            rv = C_GetTokenInfo( SlotToUse, &token);
            if (rv == CKR_OK) {
```

```
                  if(token.flags & CKF_LOGIN_REQUIRED)
                  == 0) {
                        rv = C_OpenSession(SlotToUse,
                              CKF_SERIAL_SESSION, NULL, NULL, &session);
                        assert (rv == CKR_OK);

                        rv=C_GenerateKey(session, &mechanism_gen_key, temp,
                        1, &key);
                        assert (rv == CKR_OK);

                        /* der IV muß natürlich auch gefüllt werden. */

                        rv=C_EncryptInit(session, &mechanism_use_key, key);
                        assert (rv == CKR_OK);

                        rv=C_EncryptUpdate(session, content,
                        strlen(content), NULL, &count);
                        assert (rv == CKR_OK);

                        encryptedBuffer=(CK_BYTE_PTR)malloc(count);
                        rv=C_EncryptUpdate(session, content,
                        strlen(content), encryptedBuffer, &count);
                        assert (rv == CKR_OK);

                        free(encryptedBuffer);
                        rv=C_EncryptFinal(session, NULL, &count);
                        assert (rv==CKR_OK);

                        encryptedBuffer=(CK_BYTE_PTR)malloc(count);
                        rv=C_EncryptFinal(session, encryptedBuffer, &count);
                        assert (rv=CKR_OK);
                        free(encryptedBuffer);

                        /*
                              jetzt ist das Dokument verschlüsselt
                        */
                        C_CloseSession(session);
                  }
            }
            free(pSlotInfo);
      }
      free(pSlotList);
}

C_Finalize( NULL );
```

Decryption of Data (Unwrapping a Session Key)

```
CK_RV             rv;
CK_ULONG          ulSlotCount;
CK_SLOT_ID_PTR    pSlotList;
CK_SLOT_INFO_PTR  pSlotInfo;
CK_SLOT_ID        SlotToUse;
CK_TOKEN_INFO     token;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE  key;
CK_OBJECT_HANDLE  session_key;
CK_MECHANISM      mechanism = {CKM_RSA_PKCS, NULL_PTR, 0};
char              label[]=”Unwrapped Session Key”;
CK_ATTRIBUTE      temp={CKA_LABEL, &label, strlen(label)};
CK_ULONG          ObjectCount;
CK_OBJECT_CLASS   keyClass = CKO_PRIVATE_KEY;
CK_BYTE           TRUE = true;
CK_ATTRIBUTE      keytemplate[] = {
                        { CKA_CLASS, &keyClass, sizeof(keyClass) },
                        { CKA_UNWRAP, &TRUE, sizeof(TRUE) }
                  };
int               i;


CK_BYTE_PTR       wrappedKey;
CK_ULONG          wrappedKey_length;
/*
      in wrappedKey muß der session key stehen.
      wrappedKey_length muß ebenfalls gültig sein
*/

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
```

```
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
                  Weiters muß der Benutzer aufgefordert werden eine karte in
                  den Leser zu stecken
            */
            rv = C_GetTokenInfo( SlotToUse, &token);
            if (rv == CKR_OK) {
                  if(token.flags & CKF_LOGIN_REQUIRED)
                  == 0) {
                        rv = C_OpenSession(SlotToUse,
                              CKF_SERIAL_SESSION, NULL, NULL, &session);
                        assert (rv == CKR_OK);
                        /*
                              suchen des Verschlüsselungsschlüssels
                        */
                        rv = C_FindObjectsInit(session, &keytemplate,
                              2);
                        if (rv == CKR_OK) {
                              rv = C_FindObjects(session, &key, 1,
                                    &ObjectCount);
                              assert (rv == CKR_OK);

                              rv = C_FindObjectsFinal(session);
                              assert (rv == CKR_OK);

                              rv=C_UnwarpKey(session, &mechanism, key,
                              wrappedKey, wrappedKey_length, temp, 1,
                              &session_key);
                              assert (rv=CKR_OK);

                              /* session_key halt jetzt das Handle des
                              importierten Keys.
                              */
                        }
                        C_CloseSession(session);
                  }
            }
            free(pSlotInfo);
      }
      free(pSlotList);
}

C_Finalize( NULL );
```

Changing a PIN

```
CK_RV             rv;
CK_ULONG          ulSlotCount;
CK_SLOT_ID_PTR    pSlotList;
CK_SLOT_INFO_PTR  pSlotInfo;
CK_SLOT_ID        SlotToUse;
CK_TOKEN_INFO     token;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE  key;
CK_OBJECT_CLASS   keyClass = CKO_PRIVATE_KEY;
CK_BBOOL          bTRUE = CK_TRUE;
CK_BBOOL          bFALSE = CK_FALSE;

/* Dieses Beispiel benutzt das Attribut CKA_UNWRAP zur Unterscheidung
zwischen Signatur- und Geheimhaltungsschlüssel. Je nach Anwendung können
auch andere Attribute zur Auswahl des korrekten Schlüssels sinnvoll sein. */


CK_ATTRIBUTE      keytemplate[] = {
                        { CKA_CLASS, &keyClass, sizeof(keyClass) },
                        { CKA_UNWRAP, &bFALSE, sizeof(bFALSE) }

                  };
CK_ATTRIBUTE      changetemplate[] = {
                        { CKA_SECONDARY_AUTH, &bFALSE, sizeof(bFALSE) }
                  };
int               i;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
```

```
            Weiters muß der Benutzer aufgefordert werden eine karte in
            den Leser zu stecken
        */
        rv = C_GetTokenInfo( SlotToUse, &token);
        if (rv == CKR_OK) {
            if(token.flags & CKF_LOGIN_REQUIRED)
            == 0) {
                rv = C_OpenSession(SlotToUse,
                    CKF_SERIAL_SESSION, NULL, NULL, &session);
                assert (rv == CKR_OK);
                /*
                    Suchen des Schlüssels, dessen PIN geändert
                    werden soll.
                    Signaturschlüssel besitzt CKA_UNWRAP = FALSE.
                */
                rv = C_FindObjectsInit(session, &keytemplate,
                    2);
                if (rv == CKR_OK) {
                    rv = C_FindObjects(session, &key, 1,
                        &ObjectCount);
                    assert (rv == CKR_OK);

                    rv = C_FindObjectsFinal(session);
                    assert (rv == CKR_OK);

                    rv = C_SetAttributeValue(session, key,
                    changetemplate, 1);
                    assert (rv==CKR_OK);

                    /*
                        jetzt ist die PIN geändert
                    */
                }
                C_CloseSession(session);
            }
        }
        free(pSlotInfo);
    }
    free(pSlotList);
}

C_Finalize( NULL );
```

Unblocking a PIN

```
CK_RV             rv;
CK_ULONG          ulSlotCount;
CK_SLOT_ID_PTR    pSlotList;
CK_SLOT_INFO_PTR  pSlotInfo;
CK_SLOT_ID        SlotToUse;
CK_TOKEN_INFO     token;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE  key;
CK_OBJECT_CLASS   keyClass = CKO_PRIVATE_KEY;
CK_BBOOL          bTRUE = CK_TRUE;
CK_BBOOL          bFALSE = CK_FALSE;

/* Dieses Beispiel benutzt das Attribut CKA_UNWRAP zur Unterscheidung
zwischen Signatur- und Geheimhaltungsschlüssel. Je nach Anwendung können
auch andere Attribute zur Auswahl des korrekten Schlüssels sinnvoll sein. */


CK_ATTRIBUTE      keytemplate[] = {
                        { CKA_CLASS, &keyClass, sizeof(keyClass) },
                        { CKA_UNWRAP, &bTRUE, sizeof(bTRUE) }

                  };
CK_ATTRIBUTE      unblocktemplate[] = {
                        { CKA_AUTH_PIN_FLAGS, &bTRUE, sizeof(bTRUE) }
                  };
int               i;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
```

```
                  Weiters muß der Benutzer aufgefordert werden eine karte in
                  den Leser zu stecken
            */
            rv = C_GetTokenInfo( SlotToUse, &token);
            if (rv == CKR_OK) {
                  if(token.flags & CKF_LOGIN_REQUIRED)
                  == 0) {
                        rv = C_OpenSession(SlotToUse,
                              CKF_SERIAL_SESSION, NULL, NULL, &session);
                        assert (rv == CKR_OK);
                        /*
                              Suchen des Schlüssels, dessen PIN geändert
                              werden soll.
                              Verschlüsselungsschlüssel besitzt CKA_UNWRAP =
                              TRUE.
                        */
                        rv = C_FindObjectsInit(session, &keytemplate,
                              2);
                        if (rv == CKR_OK) {
                              rv = C_FindObjects(session, &key, 1,
                                    &ObjectCount);
                              assert (rv == CKR_OK);

                              rv = C_FindObjectsFinal(session);
                              assert (rv == CKR_OK);

                              rv = C_SetAttributeValue(session, key,
                              unblocktemplate, 1);
                              assert (rv==CKR_OK);

                              /*
                                    jetzt ist die PIN entsperrt
                              */
                        }
                        C_CloseSession(session);
                  }
            }
            free(pSlotInfo);
      }
      free(pSlotList);
}

C_Finalize( NULL );
```

Import of a Certificate

```
CK_RV             rv;
CK_ULONG          ulSlotCount;
CK_SLOT_ID_PTR    pSlotList;
CK_SLOT_INFO_PTR  pSlotInfo;
CK_SLOT_ID        SlotToUse;
CK_TOKEN_INFO     token;
CK_SESSION_HANDLE session;
CK_OBJECT_HANDLE  cert;
CK_MECHANISM      mechanism_sign = {CKM_SHA1_RSA_PKCS, NULL_PTR, 0};
CK_BYTE           signature[256];
CK_ULONG          signatureLength=sizeof(signature);
CK_ULONG          ObjectCount;
CK_OBJECT_CLASS   certClass = CKO_CERTIFICATE;
CK_CERTIFICATE_TYPE     certType = CKC_X_509;
CK_BYTE           TRUE = true;
/*
      In certificate steht der ASN.1 DER kodierte Content eines X.509
      Zertifikats, daß eine PKCS#1 RSA Schlüssel enthält.
      Weiters muß certificate_length korrekt gesetzt sein.
*/
CK_BYTE_PTR       certificate;
CK_ULONG          certificate_length;
CK_ATTRIBUTE      certtemplate[] = {
                        { CKA_CLASS, &certClass, sizeof(certClass) },
                        { CKA_CERTIFICATE_TYPE, &certType, sizeof(certType)
},
                        { CKA_VALUE, &certificate, certificate_length}
                  };
int               i;

rv = C_Initialize( NULL );
assert( rv == CKR_OK);

rv = C_GetSlotList( FALSE, NULL, &ulSlotCount);
if (rv == CKR_OK) {
      pSlotList =
            (CK_SLOT_ID_PTR) malloc( ulSlotCount * sizeof(CK_SLOT_ID));
      rv = C_GetSlotList ( FALSE, pSlotList, &ulSlotCount);
      if (rv == CKR_OK) {
            pSlotInfo =
                  (CK_SLOT_INFO_PTR)
                  malloc (ulSlotCount * sizeof(CK_SLOT_INFO));
            for ( i = 0; i < ulSlotCount; i++) {
                  rv = C_GetSlotInfo( pSlotList[i], pSlotInfo[i]);
                  assert (rv == CKR_OK);
            }
            /*
```

```
                  Hier muß der Benutzer den gewünschten Slot (=Leser)
                  auswählen.
                  In SlotToUse muß dannach die zu benutzende ID stehen
                  Weiters muß der Benutzer aufgefordert werden eine karte in
                  den Leser zu stecken
            */
            rv = C_GetTokenInfo( SlotToUse, &token);
            if (rv == CKR_OK) {
                  if(token.flags & CKF_LOGIN_REQUIRED)
                  == 0) {
                        rv = C_OpenSession(SlotToUse,
                              CKF_SERIAL_SESSION, NULL, NULL, &session);
                        assert (rv == CKR_OK);

                        rv = C_CreateObject(session, certtemplate, 3,
                        &cert);
                        assert (rv=CKR_OK);

                        /*
                              in cert steht jetzt der Handle des erzeugten
                              Zertifikats.
                              Der zugehörige RSA Public Key ist über das
                              Attribut CKA_ID mit dem Zertifikat verknüpft.
                        */

                        C_CloseSession(session);
                  }
            }
            free(pSlotInfo);
      }
      free(pSlotList);
}

C_Finalize( NULL );
```

Appendix B - Library Configuration

Library Configuration can be done using a couple of Registry switches.

CSP Configuration

CSP Configuration entries are located at

HKEY_CURRENT_USER\Software\A-Trust GmbH\a.sign Client\Csp

Entry	**Type**	**Value**	**Meaning**
CachePIN	String	Yes	Decryption PIN is cached after first PIN entry
		no entry or other value	no PIN cache used
CardNumber	String	xxxxxxxxxxxxxxxx	card number (16 digit) of card to be used in CSP

Appendix C - Silent Installation

For client software rollout within large organisations, *a.sign client* supports silent installation. This chapter describes the setup parameters required for silent installation.

Silent Installation with Reader Scan

To have *a.sign client* setup scan for attached readers, simply call

```
setup.exe /s
```

The setup will accept the first reader found in the system. This method is not suitable for systems with more than one reader attached.

Silent Installation without Reader Scan

Note: using this setup parameter on an already installed *a.sign client* changes the reader setup to the specified reader parameters.

```
setup.exe /s /z"/R:rr,n,xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx"
```

This method is used to specify the smart card reader from command line, and may be used if

- the reader is not yet installed
- more than one reader is installed, and a specific reader has to be picked for use with *a.sign client*

rr .... specifies the reader type (see table below)
n .... specifies the port/interface number for the reader (usually 1 for CT-API readers, 0 for PC/SC readers
xxxxxxxx ...... specifies the exact reader name as found in the system (see below).

Reader types:

Type	**Description**
01	KOBIL KAAN Professional
02	KOBIL KAAN Standard Plus
03	KOBIL B1 PCMCIA
04	KOBIL B1 S
05	REINER SCT cyberJack KB
06	REINER SCT cyberJack e-com
07	REINER SCT cyberJack pinpad
08	Cherry Smartboard G83-6700
09	SCM Microsystems SCR241 (PCMCIA)
10	Towitoko Chipdrive pinpad / SCM Microsystems SPR532
11	UTIMACO CardMan 8630
12	UTIMACO CardMan 3621
13	Siemens Sign@tor
99	PC/SC Reader

Note that the format of the resáder type parameter has changed since client setup version 1.1.0.1f.

Reader names – CT-API readers:

The reader name contains the CT-API MDO in hex. To determine the name of your reader, install a.sign client normally and check the registry under

```
HKLM\SOFTWARE\A-Trust GmbH\a.sign Client\Reader0\Status0 and
HKLM\SOFTWARE\A-Trust GmbH\a.sign Client\Reader0\CTPort
```

Status0 contains the reader name, CTPort contains the port number n.

Reader names – PC/SC readers:

To determine the name of your PC/SC reader, install a.sign client normally and check the registry under

```
HKLM\SOFTWARE\A-Trust GmbH\a.sign Client\Reader0\ReaderName
```

This entry contains the reader name followed by the PC/SC interface number.

You can also use the string “dynamic” as the reader name. In this case, the first PC/SC reader found in the system will be used by a.sign Client. This feature is particularly useful for terminal server installations, where different readers are attached to the client PC’s.

Examples:

```
setup.exe /s /z"/R:10,1,444553434d5350527833346342e3135"
```

installs a SCM SPR532 reader on CT-API port 1.

```
setup.exe /s /z"/R:99,0,SCM Microsystems Inc. SPRx32 USB Smart
Card Reader"
```

installs a PC/SC reader named “SCM Microsystems Inc. SPRx32 USB Smart Card Reader” on PC/SC interface 0.

```
setup.exe /s /z"/R:99,1,dynamic"
```

installs an arbitrary PC/SC reader.

Silent De-Installation

To uninstall *a.sign client* in silent mode, simply call

```
setup.exe /s
```

Installation Logfile

To generate an installation logfile, add the parameter

```
/verbose"c:\is_silent.log"
```

to the setup.exe parameters.