

Advanced Card Systems Ltd.



**A E T 6 3 B i o T R U S T K e y**



**APPLICATION PROGRAMMING INTERFACE**

**Version 1.7 02-2006**

## Contents

<b>1.</b>	<b>Introduction</b>	<b>4</b>
<b>2.</b>	<b>TFM.DLL</b>	<b>5</b>
2.1	Overview	5
2.2	Communication Speed	5
2.3	TFM API – Smart Card Reader	5
2.3.1	Interface Data Structure	5
2.3.1.1	AC_APDU	5
2.3.1.2	AC_SESSION	6
2.3.1.3	AC_INFO	7
2.4	TFM API – Fingerprint Scanner	8
2.4.1	Type Declarations	8
2.4.2	Interface Data Structure	8
2.4.2.1	PT_GLOBAL_INFO	8
2.4.2.2	PT_DATA	9
2.4.2.3	PT_MEMORY_FUNCS	9
2.4.2.4	PT_BIR_HEADER	9
2.4.2.5	PT_BIR	10
2.4.2.6	PT_INPUT_BIR	10
2.4.2.7	PT_INFO	10
2.4.2.8	PT_SESSION_CFG	11
2.4.2.9	PT_FINGER_LIST	12
2.4.2.10	PT_NAVIGATION_CALLBACK	13
2.4.3	Functions	14
2.4.3.1	Application General Functions	14
2.4.3.1.a	PTInitialize	14
2.4.3.1.b	PTTerminate	14
2.4.3.1.c	PTGlobalInfo	14
2.4.3.1.d	PTOpen	15
2.4.3.1.e	PTClose	15
2.4.3.1.f	PTSetGUICallbacks	15
2.4.3.1.g	PTFree	17
2.4.3.2	PerfectTrust Biometric functions	17
2.4.3.2.a	PTCapture	17
2.4.3.2.b	PTEnroll	18
2.4.3.2.c	PTEnrollSC	20
2.4.3.2.d	PTEnrollSC3	20
2.4.3.2.e	PTVerifyMatch	21
2.4.3.2.f	PTVerify	22
2.4.3.2.g	PTVerifySC	23
2.4.3.2.h	PTVerifySCAll	24
2.4.3.2.i	PTVerifyEx	26
2.4.3.2.j	PTVerifyAll	27
2.4.3.2.k	PTDetectFingerEx	29
2.4.3.2.l	PTStoreFinger	29
2.4.3.2.m	PTDeleteFinger	30
2.4.3.2.n	PTDeleteAllFingers	30
2.4.3.2.o	PTSetFingerData	31
2.4.3.2.p	PTGetFingerData	31
2.4.3.2.q	PTListAllFingers	31
2.4.3.2.r	PTCalibrate	32
2.4.3.2.s	PTNavigate	32
2.4.3.2.t	PTClickCalibrate	33
2.4.3.2.u	PTScanQuality	34
2.4.3.2.v	PTAntispoofCapture	34
2.4.3.3	PerfectTrust Miscellaneous functions	35
2.4.3.3.a	PTInfo	35

2.4.3.3.b	PTDiagnostics .....	35
2.4.3.3.c	PTSetSessionCfgEx .....	35
2.4.3.3.d	PTGetSessionCfgEx .....	36
2.4.3.3.e	PTGetAvailableMemory .....	36
2.4.3.3.f	PTSetAppData.....	37
2.4.3.3.g	PTGetAppData .....	37
2.4.3.3.h	PTSetLED .....	38
2.4.3.3.i	PTGetLED .....	38
2.4.3.3.j	PTSleep.....	38
2.4.3.4	Callback related definitions.....	39
2.4.3.4.a	PT_STD_GUI_STATE_CALLBACK.....	39
2.4.4	Status Codes.....	40
<b>3.</b>	<b>Handling Fingerprint Template.....</b>	<b>42</b>
3.1	Initialize Smart Card to store the fingerprint templates .....	42
3.2	Store the fingerprint template to the Smart Card.....	43
3.3	Verify the fingerprint in the TFM .....	43
3.4	Registering Callback Function.....	43
3.5	GUI Message Codes .....	45
<b>4.</b>	<b>Interface Function Prototypes (Smart Card) [Proprietary Driver Only] .....</b>	<b>46</b>
4.1	AC_Open .....	46
4.2	AC_Close.....	47
4.3	AC_StartSession .....	47
4.4	AC_EndSession.....	49
4.5	AC_ExchangeAPDU.....	49
4.6	AC_GetInfo .....	50
4.7	AC_SetOption .....	51
<b>5.</b>	<b>Interface Function Prototypes (EEPROM) [Proprietary Driver Only] .....</b>	<b>53</b>
5.1	AC_ReadEEPROM.....	53
5.2	AC_WriteEEPROM.....	54
	<b>Appendix A: Table of Error Codes .....</b>	<b>55</b>

# 1. Introduction

This manual describes the use of TFM.dll interface software to program the AET63 BioTRUSTKey. It is a set of library functions implemented for the application programmers to operate the AET63 and the inserted smart cards. Currently, it is supplied in the form of window 32-bit DLL (for Windows 98/2k/XP). It can be programmed using the popular development tools like Visual C/C++, Visual Basic, Delphi, FoxPro, etc.

The AET63 BioTRUSTKey is connected to the PC via the USB interface.

Even though the hardware communication interface can be different, application programs can still be using the same API (Application Programming Interface) for operating the smart card readers. Actually, the purpose of using the TFM library is to provide the programmer with a simple and consistent interface over all possible hardware. It is the responsibility of the TFM.dll library to handle the communication details, parameter conversions and error handling. The architecture of the TFM library can be visualized as the following diagram:

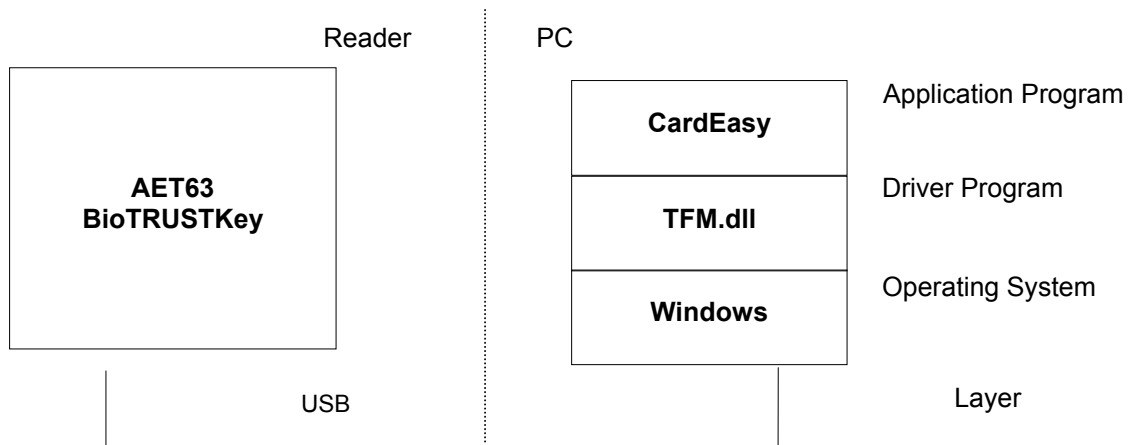


Figure 1.1

## 2. TFM.DLL

### 2.1 Overview

TFM.dll is a set of high-level functions provided for the application software to use. It provides a consistent application programming interface (TFM API) for the application to operate on the card reader and the corresponding inserted card. TFM.dll communicates with the AET63 BioTRUSTKey via the communication port facilities provided by the operating system. TFM.dll is supposed to be platform independent provided that there is a minor modification on the communication module of the TFM.dll to adapt to different operating environments.

### 2.2 Communication Speed

The TFM.DLL library controls the communication speed between the reader and the PC. The communication speed for the USB type of connection is running at 1.5Mbps.

### 2.3 TFM API – Smart Card Reader

The TFM.DLL Application Programming Interface (API) defines a common way of accessing the AET63 BioTRUSTKey. Application programs invoke TFM.DLL through the interface functions and perform operations on the inserted card using ACI commands. The header files **TFMAPI.H**, **TFMERROR.H**, **TFMTYPES.H** and **ACSR20.H**, which contains all the function prototypes and macros described below, are available for the program developer.

#### 2.3.1 Interface Data Structure

The TFM.DLL API makes use of several data structures to pass parameters between application programs and library driver. These data structures are defined in the header file acsr20.h and they are discussed below:

##### 2.3.1.1 AC\_APDU

```
typedef struct {
    BYTE          CLA;
    BYTE          INS;
    BYTE          P1;
    BYTE          P2;
    INT16        Lc;
    INT16        Le;
    BYTE          DataIn[256];
    BYTE          DataOut[256];
    WORD16       Status;
} AC_APDU;
```

The AC\_APDU data structure is used in the AC\_ExchangeAPDU function for the passing of commands and data information into the smart card. For MCU card (T=0, T=1) operation, these values are specific to the smart card operating system. You must have the card reference manual before you can perform any valid

operations on the card. Please notice that Lc represents the data length going into the card and Le represents the data length expecting from the card.

Name	Input/Output	Description
CLA	I	Instruction Class
INS	I	Instruction Code
P1	I	Parameter 1
P2	I	Parameter 2
Lc	I	Length of command data (DataIn)
Le	I/O	Length of response data (DataOut)
DataIn	I	Command data buffer
DataOut	O	Response data buffer
Status	O	Execution status of the command

### 2.3.1.2 AC\_SESSION

```
typedef struct {
    BYTE    CardType;           // Card type selected
    BYTE    SCModule;          // Selected security module.
                                // Use only when card type = AC_SCModule
    BYTE    ATRLen;            // Length of the ATR
    BYTE    ATR[128];          // ATR string
    BYTE    HistLen;           // Length of the Historical data
    BYTE    HistOffset;        // Offset of the Historical data
                                // from the beginning of ATR
    INT16   APDULenMax;        // Max. APDU supported
} AC_SESSION;
```

The AC\_SESSION data structure is used in the AC\_StartSession function call for the retrieval of ATR information from the smart card. Before calling AC\_StartSession, the program needs to specify the value of CardType. After calling the function, the ATR string can be found in ATR field and the length is stored in ATRLen.

Name	Input/Output	Description
CardType	I	The card type selected for operation (refer to Appendix C for CardType)
SCModule	I	The security module selected for operation. (The value is used only when card type = AC_SCModule)
ATRLen	O	Length of the ATR string
ATR	O	Attention to reset (ATR) string
HistLen	O	Obsolete field – not used anymore
HistOffset	O	Obsolete field – not used anymore
APDULenMax	O	Obsolete field - not used anymore

**2.3.1.3 AC\_INFO**

```
typedef struct {
    INT16      nMaxC;           // Maximum number of command data bytes
    INT16      nMaxR;           // Maximum number of data bytes that
                                // can be requested in a response
    INT16      CType;           // The card types supported by the reader
    BYTE       CStat;           // The status of the card reader
    BYTE       CSel;            // The current selection of card type
    BYTE       szRev[10];       // The 10 bytes firmware type and
                                // revision code
    INT16      nLibVer;         // Library version
    Long       lBaudRate;       // Current Running Baud Rate
} AC_INFO;
```

The AC\_INFO data structure is used in the AC\_GetInfo function call for the retrieval of reader-related information. Their meanings are described as follows:

Name	Input/Output	Description
nMaxC	O	The maximum number of command data byte (DataIn) that can be accepted in the ExchangeAPDU command
nMaxR	O	The maximum number of response data byte (DataOut) that will be appeared in the ExchangeAPDU command
CType	O	The card types supported by the reader
Cstat	O	The status of the card reader Bit0 = card present (1) or absent (0) Bit1 = card powered up (1) or powered down (0)
szRev[10]	O	The firmware revision code
nLibVer	O	Library version (e.g. 310 is equal to version 3.10)

## 2.4 TFM API – Fingerprint Scanner

### 2.4.1 Type Declarations

TFM uses type declarations for convenient handling of the application source code.

Signed byte

```
typedef      char      PT_CHAR;
```

Unsigned byte

```
typedef      unsigned char  PT_BYTE;
```

Signed 2 bytes

```
typedef      short     PT_SHORT;
```

Unsigned 2 bytes

```
typedef      unsigned short PT_WORD;
```

Signed 4 bytes

```
typedef      long      PT_LONG;
```

Unsigned 4 bytes

```
typedef      unsigned long  PT_DWORD;
```

Boolean value (zero, non-zero)

```
typedef      unsigned long  PT_BOOL;
```

Return status of functions

```
typedef      PT_LONG       PT_STATUS;
```

Handle to a connection to the TFM. This is the connection between proxy and the physical TFM.

```
typedef      PT_DWORD       PT_TFM;
```

Handle to a connection to a proxy

```
typedef      PT_DWORD       PT_CONNECTION;
```

Prototype of memory allocating function

```
typedef      void* (PTAPI *PT_MALLOC) (PT_DWORD Size);
```

Prototype of memory freeing function

```
typedef      void (PTAPI *PT_FREE) (void *Memblock);
```

### 2.4.2 Interface Data Structure

The TFM API makes use of several data structures to pass parameters between application programs and the library driver. These data structures are defined in the header file **TFMTYPES.H** and they are discussed below:

#### 2.4.2.1 PT\_GLOBAL\_INFO

The global information about this PerfectTrust implementation, especially the version info.

```
typedef struct pt_global_info {
    PT_DWORD  ApiVersion;
    PT_DWORD  Functionality;
    PT_DWORD  Flags;
} PT_GLOBAL_INFO;
```



Name	Input/Output	Description
ApiVersion	O	Version of TFM API. Highest byte = major version, second highest byte = minor version, low word = subversions.
Functionality	O	Bit mask, defining which blocks of functions are supported (see PT_GIFUNC_xxxx).
Flags	O	Additional flags (e.g. encryption strength), see PT_GIFLAGS_xxxx.

#### 2.4.2.2 PT\_DATA

This structure is used to associate any arbitrary long data block with the length information.

```
typedef struct pt_data {
    PT_DWORD Length;
    PT_BYTE  Data[1];
} PT_DATA;
```

Name	Input/Output	Description
Length	I/O	Length of the Data field in bytes
Data	I/O	The data itself, variable length

#### 2.4.2.3 PT\_MEMORY\_FUNCS

This structure is used to hand over to PerfectTrust the set of memory allocation/freeing routines, which will be then used for all dynamic memory management.

```
typedef struct pt_memory_funcs {
    PT_MALLOC pfnMalloc;
    PT_FREE   pfnFree;
} PT_MEMORY_FUNCS;
```

Name	Input/Output	Description
PfnMalloc	I	Memory allocating function
PfnFree	I	Memory freeing function

#### 2.4.2.4 PT\_BIR\_HEADER

The header of the BIR (Biometric Identification Record). This type is exactly equivalent to BioAPI's BioAPI\_BIR\_HEADER. All the integer values in the header are little-endians to ensure portability between different computers.

```
typedef struct pt_bir_header {
    PT_DWORD Length;
    PT_BYTE  HeaderVersion;
    PT_BYTE  Type;
    PT_WORD  FormatOwner;
    PT_WORD  FormatID;
    PT_CHAR  Quality;
    PT_BYTE  Purpose;
    PT_DWORD FactorsMask;
} PT_BIR_HEADER;
```

### 2.4.2.5 PT\_BIR

A container for biometric data. BIR (Biometric Identification Record). It can be an enrolled template, audit data etc. BIR consists of a header, followed by the opaque data and optionally by a signature. This type is binary compatible with BioAPI's BioAPI\_BIR. The only difference is, that in BioAPI\_BIR the data is divided into four separate memory blocks, while PT\_BIR keeps all the data together.

```
typedef struct pt_bir {
    PT_BIR_HEADER    Header;
    PT_BYTE          Data[1];
} PT_BIR;
```

### 2.4.2.6 PT\_INPUT\_BIR

A structure used to input a BIR to the API. Such input can be either the actual BIR data, or one of the predefined handles to the template cache.

```
typedef struct pt_input_bir {
    PT_BYTE    byForm;
    union {
        PT_BIR    *pBIR;
        PT_LONG    lSlotNr;
        PT_BYTE    abyReserved[20];
    } InputBIR;
} PT_INPUT_BIR;
```

Name	Input/Output	Description
ByForm	I	Form of the input BIR
PBIR	I	Used when byForm = PT_FULLBIR_INPUT
LSlotNr	I	Used when byForm = PT_SLOT_INPUT
abyReserved	I	For future use

### 2.4.2.7 PT\_INFO

A structure used by PTInfo to return status-independent information about TFM.

```
typedef struct pt_info {
    PT_DWORD    FwVersion;
    PT_DWORD    FwMinNextVersion;
    PT_DWORD    FwVariant;
    PT_DWORD    FwFunctionality;
    PT_DWORD    FwConfig;
    PT_DWORD    Id;
    PT_DWORD    AuthenticateId;           // Should not be used
    PT_DWORD    Usage;
    PT_DWORD    SensorType;
    PT_WORD    ImageWidth;
    PT_WORD    ImageHeight;
    PT_DWORD    MaxGrabWindow;           // Should not be used
    PT_DWORD    CompanionVendorCode;
} PT_INFO;
```

Name	Input/Output	Description
FwVersion	O	Version of the TFM's firmware. Highest byte = major version, second highest byte = minor version,

		low word = subversions/revisions.
FwMinNextVersion	O	Minimal version required for future firmware update
FwVariant	O	Variant of firmware - see PT_FWVARIANT_xxxx (E.g. variant with USB, variant with SIO etc.)
FwFunctionality	O	Blocks of functionality included in firmware. See PT_FWFUNC_xxxx.
FwConfig	O	FW's configuration flags, set up during manufacturing. See PT_FWCFG_xxxx.
Id	O	TFM ID. If used, allows to assign unique ID to every TFM piece. Otherwise 0.
AuthenticateId	O	ID of the Authenticate group. Every TFM with the same Authenticate code belongs to the same Authenticate group. The PTAAuthenticate function is not implemented in this version; therefore this parameter should not be used or you may set AuthenticateId == 0.
Usage	O	Type of the reader in which the TFM is used. 0 = unspecified usage.
SensorType	O	Type and version of sensor.
ImageWidth	O	Sensor image width
ImageHeight	O	Sensor image height (for strip sensor max. height)
MaxGrabWindow	O	Max. guaranteed length of the output data for PTGrabWindow. The PTGrabWindow function is not implemented in this version; therefore this parameter should not be used.
CompanionVendorCode	O	Companion vendor code

#### 2.4.2.8 PT\_SESSION\_CFG

```
typedef struct pt_session_cfg {
    PT_SECURITY_LEVEL    SecuritySetting;
    PT_ANTISPOOFING_LEVEL AntispoofingLevel;
    PT_BOOL              MultipleEnroll;
    PT_BOOL              LatentDetect;
    PT_BOOL              SubSample;
    PT_BOOL              SensorDisabling;
    PT_CALLBACK_LEVEL   CallbackLevel;
    PT_BOOL              WakeUpByFinger;
    PT_DWORD             WakeUpByFingerTimeout;
    PT_BYTE              SubWindowArea;
    PT_BOOL              WffUseHwDetection;
    PT_WORD              WffFingerDownDelay;
    PT_WORD              WffFingerUpDelay;
    PT_BOOL              RecUseHwDetection;
    PT_WORD              RecFingerDownDelay;
    PT_WORD              RecFingerUpDelay;
    PT_WORD              RecTerminationPolicy;
    PT_BOOL              RecRemoveTopdown;
    PT_BOOL              RecRemoveBottomup;
    PT_BOOL              NavUseHwDetection;
    PT_WORD              NavFingerDownDelay;
    PT_WORD              NavFingerUpDelay;
    PT_WORD              NavClickTimeMin;
    PT_WORD              NavClickTimeMax;
    PT_WORD              NavMovementDelay;
    PT_DWORD             NavClickAllowedMovement;
    PT_WORD              NavNavigationType;
    PT_WORD              BioEnrollInputType;
    PT_WORD              BioVerifyInputType;
    PT_WORD              EnableScanQualityQuery;
} PT_SESSION_CFG;
```

Name	Input/Output	Description
SecuritySetting	I	Security level for templates matching
AntispoofingLevel	I	Level of anti-spoofing
MultipleEnroll	I	Indicates whether or not enrollment should use multiple finger images
LatentDetect	I	Indicates whether or not latent detection should be activated
SubSample	I	Indicates whether or not extraction should sub-sample images
SensorDisabling	I	Indicates whether or not put sensor into sleep mode after each biometric operation
CallbackLevel	I	Indicates what amount of GUI callbacks is received during biometric operations
WakeUpByFinger	I	If PT_TRUE, FM can be woken up from deep sleep by a finger on sensor
WakeUpByFingerTimeout	I	Timeout for returning to standby after wake-up by finger (in milliseconds)
SubWindowArea	I	Area of the subwindow in percents of the full area used for extractor
WffUseHwDetection	I	Use Hardware finger detection
WffFingerDownDelay	I	Timing for finger touch
WffFingerUpDelay	I	Timing for finger lift
RecUseHwDetection	I	Use Hardware finger detection
RecFingerDownDelay	I	Timing for finger touch
RecFingerUpDelay	I	Timing for finger lift
RecTerminationPolicy	I	Defines the way, how end of finger scan is determined
RecRemoveTopdown	I	Remove striation from top
RecRemoveBottomup	I	Remove striation from bottom
NavUseHwDetection	I	Use Hardware finger detection
NavFingerDownDelay	I	Timing for finger touch
NavFingerUpDelay	I	Timing for finger lift
NavClickTimeMin	I	Minimum elapsed time to detect a click
NavClickTimeMax	I	Maximum elapsed time to detect a click
NavMovementDelay	I	Delay before movement detection
NavClickAllowedMovement	I	Maximum allowed movement for click
NavNavigationType	I	Navigation type
BioEnrollInputType	I	Reconstruction type for enrollment
BioVerifyInputType	I	Reconstruction type for verification
EnableScanQualityQuery	I	Bitmask of scan quality modes

#### 2.4.2.9 PT\_FINGER\_LIST

```
typedef struct pt_finger_list {
    PT_DWORD          NumFingers;
    struct {
        PT_LONG       SlotNr;
        PT_DWORD      FingerDataLength;
        PT_BYTE       FingerData[PT_MAX_FINGER_DATA_LENGTH];
    };
};
```

```

    } List[1];
} PT_FINGER_LIST;

```

Name	Input/Output	Description
NumFingers	O	Number of fingers in the list
SlotNr	O	Number of slot, where the finger is stored
FingerDataLength	O	Length of data associated with the finger
FingerData	O	Data associated with the finger

#### 2.4.2.10 PT\_NAVIGATION\_CALLBACK

The navigation data passed to the PT\_NAVIGATION\_CALLBACK.

```

typedef struct pt_navigation_data
{
    PT_SHORT    dx;
    PT_SHORT    dy;
    PT_WORD     signalBits;
} PT_NAVIGATION_DATA;

```

Name	Input/Output	Description
Dx	I	Delta X since the last navigation data
Dy	I	Delta Y since the last navigation data
SignalBits	I	Information bitmask, see PT_NAVIGBITS_xxxx

## 2.4.3 Functions

### 2.4.3.1 Application General Functions

The Application General Functions allow initializing the library, creating and closing logical connections to TFM, setting callbacks and performing other general operations.

#### 2.4.3.1.a PTInitialize

The **PTInitialize** function initializes the API library. It must be called before any other function.

```
LONG PTInitialize(  
    PT_MEMORY_FUNCS    *pMemoryFuncs  
);
```

##### Parameters

*pMemoryFuncs*

Structure of pointers to the memory allocation and deallocation routines.

##### Return Values

Status code

#### 2.4.3.1.b PTTerminate

The **PTTerminate** function terminates the API library. It must not be called while any connection is still open. Usually, there is no need to call this function.

```
LONG PTTerminate(  
    void  
);
```

##### Parameters

None

##### Return Values

Status code

#### 2.4.3.1.c PTGlobalInfo

The **PTGlobalInfo** returns information about the API version and other global information independent on any connection.

```
LONG PTGlobalInfo(  
    PT_GLOBAL_INFO **ppGlobalInfo  
);
```

##### Parameters

*ppGlobalInfo*

Address of a pointer, which will be set to point to a global info block. The global info block is dynamically allocated by PerfectTrust and must be freed by the application.

### Return Values

Status code

#### 2.4.3.1.d PTOpen

The **PTOpen** function opens the communication channel with the TFM.

```
PT_STATUS PTOpen(
    IN PT_CHAR *pszDsn,
    OUT PT_CONNECTION *phConnection
);
```

#### Parameters

*pszDsn*

Zero-terminated ASCII string describing the FM connection parameters.

1. For opening the ACS TFM reader through **ACS Proprietary driver**, with smart card with **transparent** file type, this should be used: "ACR30U=0 filetype=transparent".
2. For opening the ACS TFM reader through **ACS Proprietary driver**, with smart card with **record** file type, this should be used: "ACR30U=0 filetype=record".
3. For opening the ACS TFM reader through **PCSC system**, with smart card with **transparent** file type, this should be used: "PCSC=0 sharemode=shared filetype=transparent".
4. For opening the ACS TFM reader through **PCSC system**, with smart card with **record** file type, this should be used: "PCSC=0 sharemode=shared filetype=record".

*phConnection*

Resulting connection handle. At the end of the connection, it should be closed using **PTClose**. To close local connection you should call **PTClose**).

#### Return Values

Status code

#### 2.4.3.1.e PTClose

The **PTClose** function closes a connection previously opened by **PTOpen**).

```
LONG PTClose(
    PT_CONNECTION hConnection
);
```

#### Parameters

*hConnection*

Connection handle of the connection to be closed.

#### Return Values

Status code

#### 2.4.3.1.f PTSetGUICallbacks

The **PTSetGUICallbacks** function sets the address of the callback routine to be called if any called function involves displaying a biometric user interface. The callback functionality is described below.

```
LONG PTSetGUICallbacks(
    PT_CONNECTION hConnection,
```

```

        PT_GUI_STREAMING_CALLBACK pfnGuiStreamingCallback,
        void *pGuiStreamingCallbackCtx,
        PT_GUI_STATE_CALLBACK pfnGuiStateCallback,
        Void *pGuiStateCallbackCtx
    );

```

### Parameters

*hConnection*

Connection handle.

*pfnGuiStreamingCallback*

A pointer to an application callback to deal with the presentation of biometric streaming data. Reserved for future use, currently not implemented. Use NULL for this parameter.

*pGuiStreamingCallbackCtx*

A generic pointer to context information provided by the application that will be presented on the callback. Reserved for future use, currently not implemented. Use NULL for this parameter.

*pfnGuiStateCallback*

A pointer to an application callback to deal with GUI state changes.

*pGuiStateCallbackCtx*

A generic pointer to context information provided by the application that will be presented on the callback.

### Return Values

Status code

### Remarks

Application has three basic options:

Use *pfnGuiStateCallback* == NULL. In this case, no user interface will be displayed.

Use *pfnGuiStateCallback* == PT\_STD\_GUI\_STATE\_CALLBACK. This will display the standard PerfectTrust built-in user interface. In this variant *pGuiStateCallbackCtx* can have the value of the window handle to the window, which should serve as the parent for the UI. It could be NULL if the UI windows should have no parent.

Use *pfnGuiStateCallback* == your own callback. In this case your callback will be responsible for displaying the user interface.

The default settings before the first call are:

*pfnGuiStateCallback* == PT\_STD\_GUI\_STATE\_CALLBACK

and *pGuiStateCallbackCtx* == NULL.

### Example Code

```

HWND hWnd;
PT_STATUS status;

hWnd = GetActiveWindow ();

```



```
status = PTSetGUICallbacks(hConnection, NULL, NULL,
PT_STD_GUI_STATE_CALLBACK, (void *)hWnd);
```

### 2.4.3.1.g PTFree

The **PTFree** function frees memory block using deallocation function passed to API by PTInitialize() call. This function may be used for releasing structures allocated by other API functions.

```
VOID PTFree(
void *memblock
);
```

#### Parameters

*memblock*  
Supplies the memory block to be released.

#### Return Values

None

### 2.4.3.2 PerfectTrust Biometric functions

This section has been strongly inspired by the BioAPI standard. The function calls are practically the same, except for different naming of the parameter types. BioAPI uses its own type definitions, which are too specialized to be used in PerfectTrust. Also some parameters were simplified and some handles replaced by pointers to binary data blocks.

#### 2.4.3.2.a PTCapture

The **PTCapture** function scans the live finger and processes it into a template. The last template obtained through **PTCapture** will be remembered throughout the session and can be used by biometric matching functions. In addition, it can be optionally returned to the caller. This function can call GUI callbacks.

```
LONG PTCapture(
PT_CONNECTION hConnection,
PT_BYTE byPurpose,
PT_BIR **ppCapturedTemplate,
PT_LONG lTimeout,
PT_BIR **ppAuditData,
PT_DATA *pSignData,
PT_DATA **ppSignature
);
```

#### Parameters

*hConnection*  
Handle of the connection to TFM.

*byPurpose*  
Purpose of the enrollment. Use one of the **PT\_PURPOSE\_XXXX** values.

*ppCapturedTemplate*  
Address of the pointer, which will be set to point to the resulting template (BIR). The template has to be discarded by a call to **PTFree()**. If the template should be only remembered for use of next functions, leave this parameter NULL

*lTimeout*

Timeout in milliseconds. "-1" means default timeout.

*ppAuditData*

Optional address of the pointer, which will be set to point to the resulting audit data (BIR). The audit data has to be discarded by a call to **PTFree()**. The resulting value can be also **PT\_UNSUPPORTED\_BIR** (audit operation not supported) and **PT\_INVALID\_BIR** (no audit data available). The audit data contains the ID of the TFM, the image of the finger used during enrollment and other information. Depending on the settings of the TFM, the fingerprint image part of the audit data may be encrypted using the public key **KAUDITE1**, **KAUDITE2**, or by both. Use functions **PTAuditKey()** and **PTAuditData()** to get the plaintext fingerprint image audit data.

*pSignData*

Optional data to be signed together with the audit data (see *ppSignature*). It is recommended to supply unique sign data (e.g. a time stamp) for every sign operation to prevent a replay attack.

*ppSignature*

When not NULL, it represents the address of the pointer, which will be set to point to the resulting signature. The signature has to be discarded by a call to **PTFree()**. The signature is the digital signature of the *AuditData* concatenated with the *SignData* created using the TFM's private signing asymmetric key **KSIGN**. The signature can be verified anytime using the **PTVerifySignature** function.

**Return Values**

Status code

**2.4.3.2.b PTEenroll**

The **PTEenroll** function scans the live finger once or several times, depending on the session settings, and combines the images into one enrollment template. The last template obtained through **PTEenroll** will be remembered throughout the session and can be used by biometric matching functions. This function can call GUI callbacks.

```
LONG PTEenroll(
    PT_CONNECTION  hConnection,
    PT_BYTE        byPurpose,
    PT_INPUT_BIR   *pStoredTemplate,
    PT_BIR         **ppNewTemplate,
    PT_LONG        *plSlotNr,
    PT_DATA        *pPayload,
    PT_LONG        lTimeout,
    PT_BIR         **ppAuditData,
    PT_DATA        *pSignData,
    PT_DATA        **ppSignature
);
```

**Parameters**

*hConnection*

Handle of the connection to TFM

*byPurpose*

Purpose of the enrollment. Use one of the **PT\_PURPOSE\_xxxx** values.

*pStoredTemplate*

Template to be adapted. Reserved for future use. Currently not implemented. Always use NULL.

*ppNewTemplate*

Address of the pointer, which will be set to point to the resulting template (BIR). The template has to be discarded by a call to **PTFree()**. If the template should be stored only in TFM's non-volatile memory, leave this parameter NULL.

*pSlotNr*

Pointer to a variable which receives slot number (0..N-1) where the template was stored. If the value is NULL, template is not stored on TFM. *pPayload* Data to be embedded into the resulting template. Payload data is an output parameter from **PTVerify** and **PTVerifyEx** when successful match is achieved.

*ITimeout*

Timeout in milliseconds. "-1" means default timeout.

*ppAuditData*

Optional address of the pointer, which will be set to point to the resulting audit data (BIR). The audit data has to be discarded by a call to **PTFree()**. The resulting value can be also **PT\_UNSUPPORTED\_BIR** (audit operation not supported) and **PT\_INVALID\_BIR** (no audit data available). The audit data contains the ID of the TFM, the image of the finger used during enrollment and other information. Depending on the settings of the TFM, the fingerprint image part of the audit data may be encrypted using the public key **KAUDITE1**, **KAUDITE2**, or by both. Use functions **PTAuditKey()** and **PTAuditData()** to get the plaintext fingerprint image audit data.

*pSignData*

Optional data to be signed together with the audit data (see *ppSignature*). It is recommended to supply unique sign data (e.g. a time stamp) for every sign operation to prevent a replay attack.

*ppSignature*

When not NULL, it represents the address of the pointer, which will be set to point to the resulting signature. The signature has to be discarded by a call to **PTFree()**. The signature is the digital signature of the *AuditData* concatenated with the *SignData* created using the TFM's private signing asymmetric key **KSIGN**. The signature can be verified anytime using the **PTVerifySignature()** function.

**Return Values**

Status code

### 2.4.3.2.c PTEnrollSC

The **PTEnrollSC** function scans the live finger once depending on the session settings and store it to the smart card. The last template obtained through **PTEnrollSC** will be remembered throughout the session and can be used by biometric matching functions. This function can call GUI callbacks.

```
LONG PTEnrollSC(
    PT_CONNECTION          hConnection,
    PT_BYTE                recordNo,
    PT_BYTE                byPurpose,
    PT_INPUT_BIR           *pStoredTemplate,
    PT_DATA                *pPayload,
    PT_LONG                lTimeout,
    PT_BIR                 **ppAuditData,
    PT_DATA                *pSignData,
    PT_DATA                **ppSignature
);
```

#### Parameters

Please refer to **PTEnroll()** function above.

#### Return Values

Status Code

### 2.4.3.2.d PTEnrollSC3

The **PTEnrollSC3** function scans the live finger three times, combines the images into one enrollment template and stores the template into smart card.

```
PT_STATUS PTEnrollSC3 (
    IN PT_CONNECTION  hConnection,
    IN PT_BYTE        SCRecordNo,
    IN PT_BYTE        byPurpose,
    IN PT_INPUT_BIR  *pStoredTemplate,
    IN PT_DATA        *pPayload,
    IN PT_LONG        lTimeout,
    OUT PT_BIR        **ppAuditData,
    IN PT_DATA        *pSignData,
    OUT PT_DATA        **ppSignature
)
```

#### Parameters

*hConnection*

Handle to the connection to TFM

*SCRecordNo*

The record number to be stored in the smart card. It starts with value 0.

*ByPurpose*

Purpose of the enrollment. Use one of the PT\_PURPOSE\_XXXX values

*PStoredTemplate*

Template to be adapted. Reserved for future use. Currently not implemented. Always use NULL.

*Ppayload*

Data to be embedded into the resulting template. Payload data is an output parameter when successful match is achieved.

**Ltimeout**

Timeout in milliseconds. "-1" means default timeout. Timeout is used to limit the waiting for acceptable finger; it does not include the time needed for further image and template processing.

**PpAuditData**

Reserved, use NULL.

**pSignData**

Reserved, use NULL

**ppSignature**

Reserved, use NULL

**Return Values**

Status code

**2.4.3.2.e PTVerifyMatch**

The **PTVerifyMatch** function matches the supplied captured template against the supplied enrollment template. This function does not scan live finger and therefore does not call GUI callbacks.

```

LONG PTVerifyMatch(
    PT_CONNECTION    hConnection,
    PT_LONG          *plMaxFARRequested,
    PT_LONG          *plMaxFRRRequested,
    PT_BOOL          *pboFARPrecedence,
    PT_INPUT_BIR    *pCapturedTemplate,
    PT_INPUT_BIR    *pStoredTemplate,
    PT_BIR           **ppAdaptedTemplate,
    PT_BOOL          *pboResult,
    PT_LONG          *plFARAchieved,
    PT_LONG          *plFRRAchieved,
    PT_DATA          **ppPayload
);

```

**Parameters****hConnection**

Handle of the connection to TFM

**plMaxFARRequested**

Max. FAR requested by the caller

**plMaxFRRRequested**

Max. FRR requested by the caller. Optional, can be NULL.

**pboFARPrecedence**

If both FAR and FRR are provided, this parameter decides which of them takes precedence: PT\_TRUE -> FAR, PT\_FALSE -> FRR.

**pCapturedTemplate**

The template to verify - BIR data or one of the predefined handles. If NULL, the result of the last PTCapture or PTEnroll will be used.

**pStoredTemplate**

The template to be verified against - BIR data or one of the predefined handles.

**ppAdaptedTemplate**

Address of the pointer, which will be set to point to a template created by adapting the *pStoredTemplate*. Reserved for future use, currently not implemented. Always use NULL.

**pboResult**

The result: Match/no match

**plFARAchieved**

The value of FAR achieved

*pIFRRAchieved*

The value of the FRR achieved

*ppPayload*

Address of the pointer, which will be set to point to the payload data, originally embedded in the *pStoredTemplate*. Payload data is available only when successful match is achieved.

**Return Values**

Status code

**2.4.3.2.f PTVerify**

The **PTVerify** function scans the live finger or uses the last captured finger data and tries to match it against the supplied enrollment template. If the function scans a live finger, the template obtained will be remembered throughout the session and can be used by other biometric matching functions. This function can call GUI callbacks (unless *boCapture* is FALSE);

```
LONG PTVerify(
    PT_CONNECTION hConnection,
    PT_LONG *plMaxFARRequested,
    PT_LONG *plMaxFRRRequested,
    PT_BOOL *pboFARPrecedence,
    PT_INPUT_BIR *pStoredTemplate,
    PT_BIR **ppAdaptedTemplate,
    PT_BOOL *pboResult,
    PT_LONG *plFARAchieved,
    PT_LONG *plFRRAchieved,
    PT_DATA **ppPayload,
    PT_LONG lTimeout,
    PT_BOOL boCapture,
    PT_BIR **ppAuditData,
    PT_DATA *pSignData,
    PT_DATA **ppSignature
);
```

**Parameters***hConnection*

Handle of the connection to TFM.

*plMaxFARRequested*

Max. FAR requested by the caller.

*plMaxFRRRequested*

Max. FRR requested by the caller. Optional, can be NULL.

*pboFARPrecedence*

If both FAR and FRR are provided, this parameter decides which of them takes precedence: PT\_TRUE -> FAR, PT\_FALSE -> FRR.

*pStoredTemplate*

The template to be verified against - BIR data or one of the predefined handles.

*ppAdaptedTemplate*

Address of the pointer, which will be set to point to a template created by adapting the *pStoredTemplate*. Reserved for future use, currently not implemented. Always use NULL.

*pboResult*

The result: Match/no match.

*plFARAchieved*

The value of FAR achieved.

*plFRRAchieved*

The value of the FRR achieved.

***ppPayload***

Address of the pointer, which will be set to point to the payload data, originally embedded in the *pStoredTemplate*. Payload data is available only when successful match is achieved.

***lTimeout***

Timeout in milliseconds. "-1" means default timeout.

***boCapture***

If `PT_TRUE`, **PTVerify** at first captures live fingerprint. If `PT_FALSE`, result of the last finger capturing function (e.g. **PTCapture** or **PTEnroll**) will be used.

***ppAuditData***

Optional address of the pointer, which will be set to point to the resulting audit data (BIR). The audit data has to be discarded by a call to **PTFree()**. The resulting value can also be **PT\_UNSUPPORTED\_BIR** (audit operation not supported) and **PT\_INVALID\_BIR** (no audit data available). The audit data contains the ID of the TFM, the image of the live finger used during verification and other information. Depending on the settings of the TFM, the fingerprint image part of the audit data may be encrypted using the public key **KAUDITV1**, **KAUDITV2**, or by both. Use functions **PTAuditKey()** and **PTAuditData()** to get the plaintext fingerprint image audit data.

***pSignData***

Optional data to be signed together with the audit data (see *ppSignature*). It is recommended to supply unique sign data (e.g. a time stamp) for every sign operation to prevent a replay attack.

***ppSignature***

When not NULL, it represents the address of the pointer, which will be set to point to the resulting signature. The signature has to be discarded by a call to **PTFree()**. The signature is the digital signature of the *AuditData* concatenated with the *pSignData* created using the TFM's private signing asymmetric key **KSIGN**. The signature can be verified anytime using the **PTVerifySignature** function.

**Return Values**

Status code

**2.4.3.2.g PTVerifySC**

The **PTVerifySC** function scans the live finger or uses the last captured finger data and try to match it against the fingerprint template stored in the smart card.

```
PT_STATUS PTVerifySC (
    IN PT_CONNECTION hConnection,
    IN PT_LONG *pIMaxFARRequested,
    IN PT_LONG *pIMaxFRRRequested,
    IN PT_BOOL *pboFARPrecedence,
    IN PT_BYTE SCRecordNo,
    OUT PT_BIR **ppAdaptedTemplate,
    OUT PT_BOOL *pboResult,
    OUT PL_LONG *pIFARAchieved,
    OUT PT_LONG *pIFRRAchieved,
    OUT PT_DATA **ppPayload,
    IN PT_LONG lTimeout,
    IN PT_BOOL boCapture,
    OUT PT_BIR **ppAuditData,
    IN PT_DATA *pSignData,
    OUT PT_DATA **ppSignature
)
```

**Parameters*****hConnection***

Handle to the connection to TFM

*pIMaxFARRequested*

Max. FAR requested by the caller.

*pIMaxFRRRequested*

Max. FRR requested by the caller. Optional, can be NULL

*pboFARPrecedence*

If both FAR and FRR are provided, this parameter decides which of them takes precedence.

PT\_TRUE->FAR, PT\_FALSE->FRR.

*SCRecordNo*

The record number to be stored in the smart card. It starts with value 0.

*ppAdaptedTemplate*

Reserved for future use, always use NULL

*pboResult*

The result: Match/No match

*pIFARAchieved*

The value of the FAR achieved.

*pIFRRAchieved*

The value of the FRR achieved.

*ppPayload*

Address of the pointer, which will be set to point to the payload data, originally embedded in the StoredTemplate. Payload data is available only when successful match is achieved.

*ITimeout*

Timeout in milliseconds. "-1" means default timeout. Timeout is used to limit the waiting for acceptable finger; it does not include the time needed for further image and template processing.

*boCapture*

If PT\_TRUE, PTVerifySC at first captures live fingerprint. If PT\_FALSE, result of the last finger capturing function will be used.

*ppAuditData*

Reserved, use NULL.

*pSignData*

Reserved, use NULL.

*ppSignature*

Reserved, use NULL.

**Return Values**

Status code

**2.4.3.2.h PTVerifySCAll**

The **PTVerifySCAll** function scans the live finger or uses the last captured finger data and tries to match it against the entire fingerprint templates stored in the smart card.

```
PT_STATUS PTVerifySCAll (
    IN PT_CONNECTION hConnection,
```



```

    IN PT_LONG *pIMaxFARRequested,
    IN PT_LONG *pIMaxFRRRequested,
    IN PT_BOOL *pboFARPrecedence,
    IN PT_BYTE NumRecord,
    OUT PT_BIR **ppAdaptedTemplate,
    OUT PT_BOOL *pboResult,
    OUT PL_LONG *pIFARAchieved,
    OUT PT_LONG *pIFRRAchieved,
    OUT PT_DATA **ppPayload,
    IN PT_LONG lTimeout,
    IN PT_BOOL boCapture,
    OUT PT_BIR **ppAuditData,
    IN PT_DATA *pSignData,
    OUT PT_DATA **ppSignature,
    OUT PT_BYTE *pRecordNo
)

```

## Parameters

### *hConnection*

Handle to the connection to TFM

### *pIMaxFARRequested*

Max. FAR requested by the caller.

### *pIMaxFRRRequested*

Max. FRR requested by the caller. Optional, can be NULL

### *pboFARPrecedence*

If both FAR and FRR are provided, this parameter decides which of them takes precedence.

PT\_TRUE->FAR, PT\_FALSE->FRR.

### *NumRecord*

The number of records stored in the smart card. It will do the verification from record 0 to record [NumRecord - 1].

### *ppAdaptedTemplate*

Reserved for future use, always use NULL

### *pboResult*

The result: Match/No match

### *pIFARAchieved*

The value of the FAR achieved.

### *pIFRRAchieved*

The value of the FRR achieved

### *PpPayload*

Address of the pointer, which will be set to point to the payload data, originally embedded in the StoredTemplate. Payload data is available only when successful match is achieved.

### *LTimeout*

Timeout in milliseconds. "-1" means default timeout. Timeout is used to limit the waiting for acceptable finger; it does not include the time needed for further image and template processing.

### *boCapture*

If PT\_TRUE, PTVerifySCAll at first captures live fingerprint. If PT\_FALSE, result of the last finger capturing function will be used.

*ppAuditData*

Reserved, use NULL.

*pSignData*

Reserved, use NULL.

*ppSignature*

Reserved, use NULL.

*pRecordNo*

The record number that matched the scanned or last finger captured.

**Return Values**

Status Code

**2.4.3.2.i PTVerifyEx**

The **PTVerifyEx** function scans the live finger or uses the last captured finger data and tries to match it against the set of supplied enrollment templates. If the function scans live finger, the template obtained will be remembered throughout the session and can be used by other biometric matching functions. Return the index of the best matching template or -1 if no match.

This is an extension to BioAPI. Its main purpose is to be able to match user's finger against all his enrolled fingers (i.e. up to 10 fingers) without the necessity to ask user several times to scan his finger.

```

LONG PTVerifyEx(
    PT_CONNECTION hConnection,
    PT_LONG *plMaxFARRequested,
    PT_LONG *plMaxFRRRequested,
    PT_BOOL *pboFARPrecedence,
    PT_INPUT_BIR *pStoredTemplates,
    PT_BYTE _byNrTemplates,
    PT_BIR **ppAdaptedTemplate,
    PT_SHORT *pshResult,
    PT_LONG *plFARAchieved,
    PT_LONG *plFRRAchieved,
    PT_DATA **ppPayload,
    PT_LONG lTimeout,
    PT_BOOL boCapture,
    PT_BIR **ppAuditData,
    PT_DATA *pSignData,
    PT_DATA **ppSignature
);

```

**Parameters***hConnection*

Handle of the connection to TFM.

*plMaxFARRequested*

Max. FAR requested by the caller

*plMaxFRRRequested*

Max. FRR requested by the caller. Optional, can be NULL.

*pboFARPrecedence*

If both FAR and FRR are provided, this parameter decides which of them takes precedence: PT\_TRUE -&gt; FAR, PT\_FALSE -&gt; FRR.

*pStoredTemplates*

An array of templates to be verified against - BIR data or one of the predefined handles.

*byNrTemplates*Number of templates in *pStoredTemplates*

*ppAdaptedTemplate*

Address of the pointer, which will be set to point to a template created by adapting the *pStoredTemplate*. Reserved for future use, currently not implemented. Always use NULL.

*pshResult*

The result: The zero-based index of the best matching template or -1 if no match.

*pIFARAchieved*

The value of FAR achieved.

*pIFFRAchieved*

The value of the FRR achieved.

*ppPayload*

Address of the pointer, which will be set to point to the payload data, originally embedded in the *pStoredTemplate*. Payload data is available only when successful match is achieved.

*ITimeout*

Timeout in milliseconds. "-1" means default timeout.

*boCapture*

If PT\_TRUE, PTVerifyEx at first captures live fingerprint.

If PT\_FALSE, result of the last finger capturing function (e.g. **PTCapture** or **PTEnroll**) will be used.

*ppAuditData*

Optional address of the pointer, which will be set to point to the resulting audit data (BIR). The audit data has to be discarded by a call to **PTFree()**. The resulting value can also be PT\_UNSUPPORTED\_BIR (audit operation not supported) and PT\_INVALID\_BIR (no audit data available). The audit data contains the ID of the TFM, the image of the live finger used during verification and other information. Depending on the settings of the TFM, the fingerprint image part of the audit data may be encrypted using the public key KAUDITV1, KAUDITV2, or by both. Use functions **PTAuditKey()** and **PTAuditData()** to get the plaintext fingerprint image audit data.

*pSignData*

Optional data to be signed together with the audit data (see *ppSignature*). It is recommended to supply unique sign data (e.g. a time stamp) for every sign operation to prevent a replay attack.

*ppSignature*

When not NULL, it represents the address of the pointer, which will be set to point to the resulting signature. The signature has to be discarded by a call to **PTFree()**. The signature is the digital signature of the *pAuditData* concatenated with the *pSignData* created using the TFM's private signing asymmetric key KSIGN. The signature can be verified anytime using the **PTVerifySignature** function.

**Return Values**

Status code

**2.4.3.2.j PTVerifyAll**

The **PTVerifyAll** function scans the live finger or uses the last captured finger data and tries to match it against the templates stored in TFM's non-volatile memory. If the function scans live finger, the template obtained will be remembered throughout the session and can be used by other biometric matching functions. Returns the slot of the best matching template or -1 if no match.

This is an extension to BioAPI. Its main purpose is to be able to match user's finger against all the enrolled templates in the TFM's database, but without the complexity of the heavyweight BioAPI database mechanism. This is not a real one-to-many matching, but one-to-few matching.

```
LONG PTVerifyAll(
    PT_CONNECTION hConnection,
    PT_LONG *plMaxFARRequested,
    PT_LONG *plMaxFRRRequested,
    PT_BOOL *pboFARPrecedence,
    PT_BIR **ppAdaptedTemplate,
```

```

PT_LONG    *plResult,
PT_LONG    *plFARAchieved,
PT_LONG    *plFRRAchieved,
PT_DATA    **ppPayload,
PT_LONG    lTimeout,
PT_BOOL    boCapture,
PT_BIR     **ppAuditData,
PT_DATA    *pSignData,
PT_DATA    **ppSignature
);

```

## Parameters

### *hConnection*

Handle of the connection to TFM

### *plMaxFARRequested*

Max. FAR requested by the caller

### *plMaxFRRRequested*

Max. FRR requested by the caller. Optional, can be NULL.

### *pboFARPrecedence*

If both FAR and FRR are provided, this parameter decides which of them takes precedence: PT\_TRUE -> FAR, PT\_FALSE -> FRR.

### *ppAdaptedTemplate*

Address of the pointer, which will be set to point to a template created by adapting the *pStoredTemplate*. Reserved for future use, currently not implemented. Always use NULL.

### *plResult*

The result: The zero-based index of the best matching template or -1 if no match.

### *plFARAchieved*

The value of FAR achieved

### *plFRRAchieved*

The value of the FRR achieved

### *ppPayload*

Address of the pointer, which will be set to point to the payload data, originally embedded in the *pStoredTemplate*. Payload data is available only when successful match is achieved.

### *lTimeout*

Timeout in milliseconds. "-1" means default timeout.

### *boCapture*

If PT\_TRUE, *PTVerifyEx* at first captures live fingerprint. If PT\_FALSE, result of the last finger capturing function (e.g. **PTCapture** or **PTEnroll**) will be used.

### *ppAuditData*

Optional address of the pointer, which will be set to point to the resulting audit data (BIR). The audit data has to be discarded by a call to **PTFree()**. The resulting value can also be PT\_UNSUPPORTED\_BIR (audit operation not supported) and PT\_INVALID\_BIR (no audit data available). The audit data contains the ID of the TFM, the image of the live finger used during verification and other information. Depending on the settings of the TFM, the fingerprint image part of the audit data may be encrypted using the public key KAUDITV1, KAUDITV2, or by both. Use functions **PTAuditKey()** and **PTAuditData()** to get the plaintext fingerprint image audit data.

### *pSignData*

Optional data to be signed together with the audit data (see *ppSignature*). It is recommended to supply unique sign data (e.g. a time stamp) for every sign operation to prevent a replay attack.

### *ppSignature*

When not NULL, it represents the address of the pointer, which will be set to point to the resulting signature. The signature has to be discarded by a call to **PTFree()**. The signature is the digital signature of the *pAuditData* concatenated with the

*pSignData* created using the TFM's private signing asymmetric key KSIGN. The signature can be verified anytime using the **PTVerifySignature** function.

**Return Values**  
Status code

#### 2.4.3.2.k PTDetectFingerEx

The **PTDetectFingerEx** function verifies if there is a finger on the sensor. If Timeout is nonzero, the function waits for the specified time interval until the required conditions are met.

This function is an extension to BioAPI. BioAPI handles finger detection through MODULE\_EVENT events. However, this mechanism is not suitable for implementation on FM (it requires a possibility to send out asynchronous events, while FM is purely slave device). If needed, a BioAPI-compatible event behavior can be built using the **PTDetectFingerEx** function, but the implementation will be inefficient. **PTDetectFingerEx** is a superset of an obsolete function **PTDetectFinger**, which was used in previous versions of FM. **PTDetectFinger** is equivalent to **PTDetectFingerEx** (hConnection, lTimeout, PT\_DETECT\_ACCEPTABLE | PT\_DETECT\_GUI).

```
LONG PTDetectFingerEx(
    PT_CONNECTION hConnection,
    PT_LONG lTimeout,
    PT_DWORD dwFlags
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM

*lTimeout*

Timeout in milliseconds. "-1" means default timeout. "0" is an acceptable value for this function, it means "test if the current state meets the required conditions".

*dwFlags*

Bit mask determining the behavior of the function and the conditions for which the function waits.

**Return Values**  
Status code

#### 2.4.3.2.l PTStoreFinger

The **PTStoreFinger** function stores given fingerprint template in the selected slot in the non-volatile memory of the TFM. If pTemplate is NULL, it only clears the slot.

This function is an extension to BioAPI. The standard BioAPI interface for this functionality are the database functions (**DbStoreBIR**, **DbGetBIR** etc.). However, this interface is too complicated for the TFM's needs.

**The template cache consists of a predefined number of "slots". The slots are numbered 0..N-1 and accessible through setting byForm field in the PT\_INPUT\_BIR to PT\_SLOT\_INPUT.**

The "N" depends on the model of the TFM and can be found using the **PTInfo()** call. Please note that the real number of templates storable in TFM is further limited by the available memory and can be therefore lower than "N".

```
LONG PTStoreFinger(
    PT_CONNECTION hConnection,
    PT_INPUT_BIR *pTemplate,
    PT_LONG *pSlotNr
);
```

**Parameters***hConnection*

Handle of the connection to TFM

*pTemplate*

Template (BIR) to be stored in the template cache

*pSlotNr*

Pointer to a variable which receives slot number (0..N-1) where the template was stored. If the value is NULL, the template is not stored.

**Return Values**

Status code

**2.4.3.2.m PTDeleteFinger**

The **PTDeleteFinger** function deletes fingerprint template stored in the selected slot in the non-volatile memory of the TFM.

This function is an extension to BioAPI. The standard BioAPI interfaces for this functionality are the database functions (**DbStoreBIR**, **DbGetBIR** etc.). However, this interface is too heavyweight for the TFM's needs.

```
LONG PTDeleteFinger(
    PT_CONNECTION hConnection,
    PT_LONG lSlotNr
);
```

**Parameters***hConnection*

Handle to the connection to TFM

*lSlotNr*

Number of slot to delete (0..N-1)

**Return Values**

Status code

**2.4.3.2.n PTDeleteAllFingers**

The **PTDeleteAllFingers** function deletes all fingerprint templates stored in slots in the non-volatile memory of the TFM.

This function is an extension to BioAPI. The standard BioAPI interfaces for this functionality are the database functions (**DbStoreBIR**, **DbGetBIR** etc.). However, this interface is too complicated for the TFM's needs.

```
LONG PTDeleteAllFingers(
    PT_CONNECTION hConnection
);
```

**Parameters***hConnection*

Handle to the connection to TFM

**Return Values**

Status code

### 2.4.3.2.o PTSetFingerData

The **PTSetFingerData** function assigns an additional application data to a finger template stored in TFM's non-volatile memory.

```
LONG PTSetFingerData(
    PT_CONNECTION hConnection,
    PT_LONG lSlotNr,
    PT_DATA *pFingerData
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM

*lSlotNr*

The slot number of the template to be associated with data.

*pFingerData*

The data to be stored together with the template. If the data length is zero, the application data associated with given fingerprint will be deleted

#### Return Values

Status code

### 2.4.3.2.p PTGetFingerData

The **PTGetFingerData** function reads the additional application data associated with a finger template stored in TFM's non-volatile memory.

```
LONG PTGetFingerData(
    PT_CONNECTION hConnection,
    PT_LONG lSlotNr,
    PT_DATA **ppFingerData
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM

*lSlotNr*

The slot number of the template whose application data should be read.

*ppFingerData*

Address of the pointer, which will be set to point to the application data associated with given fingerprint. If no data are associated with the fingerprint, the result will be a data block with zero length. The data has to be freed by a call to **PTFree**.

#### Return Values

Status code

### 2.4.3.2.q PTListAllFingers

The **PTListAllFingers** function returns list of all fingers stored in the TFM's non-volatile memory together with their associated application data.

```
LONG PTListAllFingers(
    PT_CONNECTION hConnection,
    PT_FINGER_LIST **ppFingerList
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM.

*ppFingerList*

Address of the pointer, which will be set to point to the list of stored fingerprints and their associated data. The data has to be freed by a call to **PTFree**.

**Return Values**

Status code

**2.4.3.2.r PTCalibrate**

The **PTCalibrate** function calibrates the fingerprint sensor to suite best to the given user. The calibration data will be stored in NVM and used for all the following biometric operations, in the current and future connections (communication sessions).

The **PTCalibrate** function is currently used for strip sensors (ESS) only.

The standard calibration (**PT\_CALIB\_TYPE\_STANDARD**) is an interactive operation. The user will be prompted to put, lift or swipe his finger. This feedback will be communicated using the GUI callbacks. The success of this operation is therefore essential to enable and use the GUI callbacks. If the *CallbackLevel* member of the session configuration is set to **CALLBACKS\_NONE**, **PTCalibrate** will directly fail.

The GUI callbacks are also the only way to interrupt this operation. If the host fails to use callbacks, the only way to regain control over FM is to close and reopen the connection (communication session).

```
LONG   PTCalibrate (
        PT_CONNECTION hConnection,
        PT_DWORD      dwType
    );
```

**Parameters**

*hConnection*

Handle of the connection to TFM

*dwType*

Type of calibration operation to be performed.

**Return Values**

Status code

**2.4.3.2.s PTNavigate**

The **PTNavigate** function switches FM to navigation mode (a.k.a. biometric mouse). In this mode, FM will provide navigation info to the host.

```
LONG PTNavigate (
        PT_CONNECTION hConnection,
        PT_LONG      lEventPeriod,
        PT_NAVIGATION_CALLBACK pfnCallback,
        void          *pNavigationCallbackCtx
    );
```

**Parameters**

*hConnection*

Handle of the connection to FM

*lEventPeriod*

Delay in milliseconds between sending navigation data from FM. FM will send one packet per period with all the navigation data accumulated over the period. If *lEventPeriod* is set to "-1", FM will use an on-demand mode.



*pfnCallback*

Callback function, called every time when a navigation data packet arrives to the host.

*pNavigationCallbackCtx*

A generic pointer to context data. This pointer will be supplied to the callback function every time it is called.

**Return Values**

Status Code

**Remarks**

During the **PTNavigate** call, the FM sends periodically packets with navigation data to the host. The arrived packets trigger calling the *pfnCallback* to deliver the data to the application. Please note that a new callback will not be executed until the current returns. If the callback processing takes too long, some navigation data may be lost. Also other communication errors may lead to a lost navigation data - due to the nature of navigation, it makes no sense to use error-correcting protocol.

If *IEventPeriod* = -1, FM will use on-demand mode. In this mode, FM at the beginning sends one navigation data packet and then waits with sending next navigation data until it receives a request from the host. The request is sent every time the *pfnCallback* function on the host returns. This mode is suitable for host applications with slow callback processing. It protects against losing navigation data due to an overrun.

**PTNavigate** will finish when the *pfnCallback* returns with cancel request value. Due to asynchronous communication the host may still receive a few navigation callbacks after the cancel was requested.

**2.4.3.2.t PTClickCalibrate**

The **PTClickCalibrate** function regulates the navigation behavior for the given user, especially the recognition of clicking (tapping). The calibration data will be stored in NVM and used for all the following biometric operations, in the current and future connections (communication sessions).

```
LONG PTClickCalibrate (
    PT_CONNECTION hConnection,
    PT_DWORD      dwType,
    PT_LONG       lEventPeriod,
    PT_NAVIGATION_CALLBACK pfnCallback,
    void          *pNavigationCallbackCtx
);
```

**Parameters***hConnection*

Handle of the connection to FM

*dwType*

The type of the calibration operation. For successful calibration, **PTClickCalibrate** has to be called several times with the specified sequence of types. After the last call of the sequence, the calibration will be completed and effective. To cancel the effect of **PTClickCalibrate** and reset the calibration to the factory default, use value **PT\_CLICKCALIB\_RESET**. In this case, no navigation callbacks are called and the function returns immediately.

*IEventPeriod*

Delay in milliseconds between sending navigation data from FM. FM will send one packet per period with all the navigation data accumulated over the period. If *IEventPeriod* is set to "-1", FM will use an on-demand mode.

*pfnCallback*

Callback function, called every time a navigation data packet arrives to the host.

*pNavigationCallbackCtx*

A generic pointer to context data. This pointer will be supplied to the callback function every time it is called.

**Return Values**

Status Code

**Remarks**

**PTClickCalibrate** behaves very similarly to the **PTNavigate**. It calls navigation callbacks the same way as **PTNavigate**. The only (and main) difference is that **PTClickCalibrate** is used to collect information about given user's behavior.

For this purpose, in some phases (*dwTypes*) could be disabled by either the tapping or the mouse movement functionality.

The host application should, for every phase, prompt the user to perform an action appropriate for that phase and then display a progress feedback. A flag in the navigation data will inform the host when FM collected enough data for the given phase. The host should then finish the operation and progress to the next phase. However, if needed (e.g. for the purpose of nice user interface), the current operation can be continued by the host until the host decides to finish it.

**2.4.3.2.u PTScanQuality**

The **PTScanQuality** function returns scan quality of last finger swipe

```
LONG PTScanQuality(
    PT_CONNECTION hConnection,
    PT_DWORD      *pdwScanQuality
);
```

**Parameters***hConnection*

Handle of the connection to FM

*pdwScanQuality*

Returns scan quality of last finger swipe

**Return Values**

Status code

**2.4.3.2.v PTAntispoofCapture**

The **PTAntispoofCapture** function captures antispoofting data from the sensor. The captured data will be remembered throughout the session and can be used by other biometric commands, which scans the finger. This function can call GUI callbacks.

```
LONG PTAntispoofCapture (
    PT_CONNECTION hConnection,
    PT_LONG      lTimeout,
    PT_DWORD      dwOperation,
    PT_BOOL      *pboLiveFingerDetected
);
```

**Parameters***hConnection*

Handle of the connection to FM

*lTimeout*

Timeout in milliseconds. "-1" means default timeout.

This parameter is meaningful only for interactive operations.

*dwOperation*

Operation to be performed

*pboLiveFingerDetected*

Returns PT\_TRUE if live finger was presented to the sensor

**Return Values**  
Status code

### 2.4.3.3 *PerfectTrust Miscellaneous functions*

#### 2.4.3.3.a **PTInfo**

The **PTInfo** function returns a set of information about the connection and the TFM, including the version of TFM.

```
LONG PTInfo(
    PT_CONNECTION hConnection,
    PT_INFO **ppInfo
);
```

**Parameters**

*hConnection*

Handle of the connection to TFM

*ppInfo*

Returned structure with information about the connection and the TFM

**Return Values**

Status code

#### 2.4.3.3.b **PTDiagnostics**

The **PTDiagnostics** function is primarily targeted for use in manufacturer's diagnostic programs.

```
LONG PTDiagnostics(
    PT_CONNECTION hConnection,
    PT_DATA *pInData,
    PT_DATA **ppOutData
);
```

**Parameters**

*hConnection*

Handle of the connection to TFM

*pInData*

Input data block

*ppOutData*

Pointer to a variable, which will receive the address of the output data block. The data has to be freed by host.

**Return Values**

Status code

**Remarks**

This function can be called directly after opening a communication session. **PTDiagnostics** is also exempted from the need of host authentication, which may be required in future API versions.

#### 2.4.3.3.c **PTSetSessionCfgEx**

The **PTSetSessionCfgEx** function sets the session parameters of the TFM. The parameters influence especially the behavior of the biometric functions - e.g. should we use the advanced or the standard templates etc.

The change of parameters is valid only for the current session. Each new session starts with the default settings.

**PTSetSessionCfgEx** is an extension of the now obsolete function **PTSetSessionCfg**.

```
LONG PTSetSessionCfgEx(
    PT_CONNECTION hConnection,
    PT_WORD wCfgVersion,
    PT_WORD wCfgLength,
    PT_SESSION_CFG *pSessionCfg
);
```

#### Parameters

*hConnection*  
Handle of the connection to TFM

*wCfgVersion*  
Version of the configuration data. Use the constant **PT\_CURRENT\_SESSION\_CFG**

*wCfgLength*  
Length of the configuration data

*pSessionCfg*  
Session configuration to be set

#### Return Values

Status code

#### 2.4.3.3.d PTGetSessionCfgEx

The **PTGetSessionCfgEx** function gets the current session parameters of the TFM. The parameters influence especially the behavior of the biometric functions - e.g. should we use the advanced or the standard templates etc.

**PTGetSessionCfgEx** is an extension of the now obsolete function **PTGetSessionCfg**.

```
LONG PTGetSessionCfgEx(
    PT_CONNECTION hConnection,
    PT_WORD wCfgVersion,
    PT_WORD *pwCfgLength,
    PT_SESSION_CFG **ppSessionCfg
);
```

#### Parameters

*hConnection*  
Handle of the connection to TFM

*wCfgVersion*  
Requested version of the configuration data

*pwCfgLength*  
Pointer to the length of the received configuration data

*ppSessionCfg*  
Returned session configuration

#### Return Values

Status code

#### 2.4.3.3.e PTGetAvailableMemory

The **PTGetAvailableMemory** function returns the size in bytes of the remaining EEPROM memory on the TFM available for application's usage.

```
LONG PTGetAvailableMemory(
    PT_CONNECTION hConnection,
```

```

        PT_DWORD          dwType,
        PT_DWORD          *pdwAvailableMemory
    );

```

**Parameters***hConnection*

Handle of the connection to TFM

*dwType*Requested type of memory (see values **PT\_MEMTYPE\_xxxx**)*pdwAvailableMemory*

Returned size of remaining EEPROM memory

**Return Values**

Status code

**2.4.3.3.f PTSetAppData**

The **PTSetAppData** function allows the application to store a block of arbitrary binary data in TFM's non-volatile memory. There is only one block shared by all applications.

```

LONG PTSetAppData (
    PT_CONNECTION hConnection,
    PT_DWORD      dwArea,
    PT_DATA      *pAppData
);

```

**Parameters***hConnection*

Handle of the connection to TFM

*dwArea*Area to write. One of the **PT\_AREA\_xxx** values.*pAppData*

The data to be stored in NVM. If the data length is zero, the application data will be deleted.

**Return Values**

Status code

**2.4.3.3.g PTGetAppData**

The **PTGetAppData** function reads the application data stored in TFM's non-volatile memory.

```

LONG PTGetAppData (
    PT_CONNECTION hConnection,
    PT_DWORD      dwArea,
    PT_DATA      **ppAppData
);

```

**Parameters***hConnection*

Handle of the connection to TFM

*dwArea*Area to read. One of the **PT\_AREA\_xxx** values.*ppAppData*Address of the pointer, which will be set to point to the application data. If no data is stored in NVM, the result will be a data block with zero length. The data has to be freed by a call to **PTFree**.**Return Values**

Status code

### 2.4.3.3.h PTSetLED

The **PTSetLED** function allows the application to control the state and behavior of the two-user interface LEDs, which can be optionally connected to the TFM.

```
LONG PTSetLED(
    PT_CONNECTION hConnection,
    PT_DWORD      dwMode,
    PT_DWORD      dwLED1,
    PT_DWORD      dwLED2
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM

*dwMode*

Mode of the LEDs. Different modes define different behavior of the LEDs during specific operations, especially the biometrics. See **PT\_LED\_MODE\_xxxx**.

*dwLED1*

Parameter defining the detailed behavior of the first LED. This parameter is mode-specific.

*dwLED2*

Parameter defining the detailed behavior of the second LED. This parameter is mode-specific.

#### Return Values

Status code.

### 2.4.3.3.i PTGetLED

The **PTGetLED** function allows the application to query the state and behavior of the two-user interface LEDs, which can be optionally connected to the TFM.

```
LONG PTGetLED(
    PT_CONNECTION hConnection,
    PT_DWORD      *pdwMode,
    PT_DWORD      *pdwLED1,
    PT_DWORD      *pdwLED2
);
```

#### Parameters

*hConnection*

Handle of the connection to TFM

*pdwMode*

Mode of the LEDs. See **PTSetLED** for details.

*pdwLED1*

Parameter defining the detailed behavior of the first LED. This parameter is mode-specific. See **PTSetLED** for details.

*pdwLED2*

Parameter defining the detailed behavior of the second LED. This parameter is mode-specific. See **PTSetLED** for details.

#### Return Values

Status code

### 2.4.3.3.j PTSleep

The **PTSleep** function switches FM to deep sleep or standby mode. In this mode, FM's CPU is stopped to minimize power consumption. FM can be woken up from the sleep mode either by the host or by another event (e.g. when a finger is detected on FM's sensor).

When FM wakes up, the PTSleep function completes and returns the cause of wake up to the caller.

```
LONG PTSleep (
    PT_CONNECTION hConnection,
    PT_DWORD      dwSleepMode,
    PT_IDLE_CALLBACK pfnCallback,
    void          *pIdleCallbackCtx,
    PT_DWORD      *pdwWakeupCause
);
```

#### Parameters

*hConnection*

Handle to the connection to FM.

*dwSleepMode*

Sleep mode to be used. Possible values are **PT\_SLEEP\_MODE\_DEEPSLEEP** (fingerprint sensor is powered down (finger detect is not active)), **PT\_SLEEP\_MODE\_STANDBY** (finger detect is active).

*pfnCallback*

Callback function, called periodically all the time FM is sleeping. Optional, can be NULL.

*pIdleCallbackCtx*

A generic pointer to context data. This pointer will be supplied to the callback function every time it is called.

*pdwWakeupCause*

The cause of wakeup. Currently the following causes are possible:

**PT\_WAKEUP\_CAUSE\_HOST** (signal from the Host),

**PT\_WAKEUP\_CAUSE\_FINGER** (a finger was detected).

#### Return Values

Status code

### 2.4.3.4 Callback related definitions

#### 2.4.3.4.a PT\_STD\_GUI\_STATE\_CALLBACK

The **PT\_STD\_GUI\_STATE\_CALLBACK** function is a standard PerfectTrust GUI callback. This callback is the default until other callback is specified by **PTSetGUICallbacks**

```
LONG PT_STD_GUI_STATE_CALLBACK (
    void          *pGuiStateCallbackCtx,
    PT_DWORD      dwGuiState,
    PT_BYTE      *pbyResponse,
    PT_DWORD      dwMessage,
    PT_BYTE      byProgress,
    void          *pSampleBuffer,
    PT_DATA      *pData
);
```

#### Parameters

*pGuiStateCallbackCtx*

A generic pointer to context information that was provided by the original requester and is being returned to its originator.

*dwGuiState*

A bitmask indicating the current GUI state plus an indication of what other parameters are available. It can be combined from values **PT\_SAMPLE\_AVAILABLE**, **PT\_MESSAGE\_PROVIDED** and **PT\_PROGRESS\_PROVIDED**. In the current implementation, only **PT\_MESSAGE\_PROVIDED** is used.

*pbyResponse*

The response from the application back to the PerfectTrust Proxy API on return from the callback. Can be one of values PT\_CANCEL or PT\_CONTINUE. Other values are reserved for future use.

*dwMessage*

The number of a message to display to the user. For message numbers see PT\_GUIMSG\_XXXX. GuiState indicates if a Message is provided; if not the parameter is 0.

*byProgress*

A value that indicates (as a percentage) the amount of progress in the development of a Sample/BIR. The value may be used to display a progress bar. GuiState indicates if a sample Progress value is provided in the call. Otherwise, the parameter is 0. This parameter is reserved for future use. Currently, it is always 0.

*pSampleBuffer*

The current sample buffer for the application to display. GuiState indicates if a sample Buffer is provided; if not the parameter is NULL. This parameter is reserved for future use. Currently, it is always NULL. The buffer is allocated and controlled by PerfectTrust, it must not be freed.

*pData*

Optional data, which may be available for some GUI message codes. If no data is provided, the parameter is NULL. The data is allocated and controlled by PerfectTrust, it must not be freed.

### Return Values

Status code

### Remarks

It is guaranteed that if an operation displays GUI, the first GUI message will be "Begin GUI" and the last one "End GUI". At least one more call with message "End GUI" will be sent even in the case when the application used *pbResponse* = PT\_CANCEL.

## 2.4.4 Status Codes

Code	Value	Description
PT_STATUS_OK	0	Success return status
PT_STATUS_GENERAL_ERROR	-1001	General or unknown error status. It is also possible that the function only partially succeeded, and that the device is in an inconsistent state.
PT_STATUS_API_NOT_INIT	-1002	PerfectTrust API wasn't initialized
PT_STATUS_API_ALREADY_INITIALIZED	-1003	PerfectTrust API has been already initialized
PT_STATUS_INVALID_PARAMETER	-1004	Invalid parameter error
PT_STATUS_INVALID_HANDLE	-1005	Invalid handle error
PT_STATUS_NOT_ENOUGH_MEMORY	-1006	Not enough memory to process given operation
PT_STATUS_MALLOC_FAILED	-1007	Failure of extern memory allocation function
PT_STATUS_DATA_TOO_LARGE	-1008	Passed data are too large
PT_STATUS_NOT_ENOUGH_PERMANENT_MEMORY	-1009	Not enough permanent memory to store data
PT_STATUS_MORE_DATA	-1010	There is more data to return than the supplied buffer can contain
PT_STATUS_FUNCTION_FAILED	-1033	Function failed



<b>PT_STATUS_INVALID_INPUT_BIR_FORM</b>	-1036	Invalid form of PT_INPUT_BIR structure
<b>PT_STATUS_WRONG_RESPONSE</b>	-1037	TFM has returned wrong or unexpected response
<b>PT_STATUS_NOT_ENOUGH_TFM_MEMORY</b>	-1038	Not enough memory on TFM to process given operation
<b>PT_STATUS_ALREADY_OPENED</b>	-1039	Connection is already opened
<b>PT_STATUS_CANNOT_CONNECT</b>	-1040	Cannot connect to TFM
<b>PT_STATUS_TIMEOUT</b>	-1041	Timeout elapsed
<b>PT_STATUS_BAD_BIO_TEMPLATE</b>	-1042	Bad biometric template
<b>PT_STATUS_SLOT_NOT_FOUND</b>	-1043	Requested slot was not found
<b>PT_STATUS_ANTISPOOFING_EXPORT</b>	-1044	Attempt to export antispoofing info from TFM
<b>PT_STATUS_ANTISPOOFING_IMPORT</b>	-1045	Attempt to import antispoofing info to TFM
<b>PT_STATUS_ACCESS_DENIED</b>	-1046	Access to operation is denied
<b>PT_STATUS_NO_TEMPLATE</b>	-1049	No template was captured in current session
<b>PT_STATUS_BIOMETRIC_TIMEOUT</b>	-1050	Timeout for biometric operation has expired
<b>PT_STATUS_CONSOLIDATION_FAILED</b>	-1051	Failure of template consolidation
<b>PT_STATUS_BIO_OPERATION_CANCELED</b>	-1052	Biometric operation canceled
<b>PT_STATUS_AUTHENTICATION_FAILED</b>	-1053	Authentication failed
<b>PT_STATUS_UNKNOWN_COMMAND</b>	-1054	Unknown command
<b>PT_STATUS_GOING_TO_SLEEP</b>	-1055	Power off attempt failed
<b>PT_STATUS_NOT_IMPLEMENTED</b>	-1056	Function or service is not implemented
<b>PT_STATUS_COMM_ERROR</b>	-1057	General communication error
<b>PT_STATUS_SESSION_TERMINATED</b>	-1058	Session was terminated
<b>PT_STATUS_TOUCH_CHIP_ERROR</b>	-1059	Touch chip error occurred
<b>PT_STATUS_I2C_EEPROM_ERROR</b>	-1060	I2C EEPROM error occurred
<b>PT_STATUS_INVALID_PURPOSE</b>	-1061	Purpose parameter =or BIR's purpose is invalid for given operation
<b>PT_STATUS_SWIPE_TOO_BAD</b>	-1062	Finger swipe is too bad for image reconstruction
<b>PT_STATUS_NOT_SUPPORTED</b>	-1063	Value of parameter is not supported
<b>PT_STATUS_CALIBRATION_FAILED</b>	-1064	Calibration failed
<b>PT_STATUS_ANTISPOOFING_NOT_CAPTURED</b>	-1065	Antispoofing data were not captured
<b>PT_STATUS_LATCHUP_DETECTED</b>	-1066	Sensor latch-up event detected
<b>PT_STATUS_DIAGNOSTICS_FAILED</b>	-1067	Diagnostics failed
<b>PT_STATUS_IMAGE_INCONSISTENCE</b>	-1100	Enroll Image Inconsistence

### 3. Handling Fingerprint Template

#### 3.1 Initialize Smart Card to store the fingerprint templates

1. Choose a smart card that can have enough size to store the fingerprint templates. The fingerprint template can be 540 bytes maximum in size.
2. Initialize the smart card and create files in the smart card to store the fingerprint template. The initialization can be done using the standard smart card application tools.
3. List down the APDUs to access to the file storing the fingerprint template. For each fingerprint template, there should be one list of APDUs for enrollment and another list of APDUs for verification. Store the lists of APDUs to the EEPROM. The lists of APDUs are stored in the EEPROM in the Tag-Length-Value (TLV) format.

Tag	Length	Description
0x80	0x00	To reset the smart card
0xA0	Length of the APDU	To send the APDU to the smart card. The APDU is specified in the "Value" field.
0x03	Length of the APDU	This is the last APDU to read/write the fingerprint template in the smart card. The APDU is specified in the "Value" field. This APDU should either be the READ_BINARY or WRITE_BINARY command. The first byte is the CLA. The second byte is the INS. The third and forth byte specified the address to be read/written.

**Example 1**

The data to be written to EEPROM for enrollment can be:

```
80 00      A0 06 00 A4 02 00 02 33 33 00      03 05 00 D6 00 00 00
Reset      Select file APDU                      Write Binary APDU
```

**Example 2**

The data to be written to EEPROM for verification can be:

```
80 00      A0 06 00 A4 02 00 02 33 33 00      03 05 00 B0 00 00 00
Reset      Select file APDU                      Read Binary APDU
```

The EEPROM can store lists of APDUs for up to 5 fingerprint templates. For each fingerprint template, there is one list of APDUs for enrollment and one list of APDUs for verification. The address mapping of the EEPROM is shown in figure 1.

Address 0x0000		Enroll (256 bytes max)
Address 0x0100	----- <b>RECORD 0</b> -----	Verify (256 bytes max)
Address 0x0200		Enroll (256 bytes max)
Address 0x0300	----- <b>RECORD 1</b> -----	Verify (256 bytes max)
Address 0x0400		Enroll (256 bytes max)
Address 0x0500	----- <b>RECORD 2</b> -----	Verify (256 bytes max)
Address 0x0600		Enroll (256 bytes max)
Address 0x0700	----- <b>RECORD 3</b> -----	Verify (256 bytes max)
Address 0x0800		Enroll (256 bytes max)
Address 0x0900	----- <b>RECORD 4</b> -----	Verify (256 bytes max)

**Figure 1 Address Mapping of the EEPROM**

### 3.2 Store the fingerprint template to the Smart Card

1. Before doing any operation, you should call the **PTOpen()** to open the port (connection) to our reader.
2. Use the API called **PTEnrollSC3()** to save the fingerprint template to the smart card.
3. AET63 uses the APDUs stored in the EEPROM to access to the specified smart card file.

### 3.3 Verify the fingerprint in the TFM

1. Use the API called **PTVerifySC()** to verify the fingerprint with the template stored in smart card.
2. Use another API called **PTVerifySCAll()** to verify the fingerprint with all the templates stored in smart card.
3. AET63 uses the APDUs stored in the EEPROM to access to the specified smart card file. The **PT\_DATA** structure is used to associate any arbitrary long data block with the length information.

### 3.4 Registering Callback Function

Besides depending on our library to handle the GUI callback, user can register a callback function to receive these GUI callback message and handle them.

```
PT_STATUS PTSetGUICallbacks (
    IN PT_CONNECTION hConnection,
    IN PT_GUI_STREAMING_CALLBACK pfnGuiStreamingCallback,
    IN void *pGuiStreamingCallbackCtx,
    IN PT_GUI_STATE_CALLBACK pfnGuiStateCallback,
    IN void *pGuiStateCallbackCtx
)
```

This is a type of the callback function that an application can supply to enable itself to display GUI state information to the user.

```
typedef PT_STATUS (PTAPI *PT_GUI_STATE_CALLBACK) (
    IN void *pGuiStateCallbackCtx,
    IN PT_DWORD dwGuiState,
    OUT PT_BYTE *pbyResponse,
    IN PT_DWORD dwMessage,
    IN PT_BYTE byProgress,
    IN void *pSampleBuffer,
    IN PT_DATA *pData
)
```

Description	A type of the callback function that an application can supply to enable itself to display GUI state information to the user.	
Parameters	pGuiStateCallbackCtx	A generic pointer to context information that was provided to the PTSetGUICallbacks function and now is returned in every callback.
	dwGuiState	A bitmask indicating the current GUI state plus an indication of what other parameters are available. It can be combined from values PT_SAMPLE_AVAILABLE, PT_MESSAGE_PROVIDED and PT_PROGRESS_PROVIDED. In the current implementation, only PT_MESSAGE_PROVIDED is used.
	pbyResponse	The response from the application back to the PTAPI on return from the callback. Can be one of values PT_CANCEL (cancel the operation in progress) or PT_CONTINUE (continue with the operation in progress). Other values are reserved for future use.
	dwMessage	The number of a message to display to the user. For message numbers see PT_GUIMSG_XXXX. DwGuiState indicates if a message is provided; if no, this parameter should be ignored.
	byProgress	Reserved for future use.
	pSampleBuffer	Reserved for future use.
	pData	Reserved for future use.
Return Value	PT_STATUS	Result code. PT_STATUS_OK (0) means success

PT_MESSAGE_PROVIDED	0x1	dwMessage parameter is valid
PT_CANCEL	0x1	Cancel the operation
PT_CONTINUE	0x0	Continue the operation

### 3.5 GUI Message Codes

PT_GUIMSG_GOOD_IMAGE	0	Image with acceptable quality was just scanned.
PT_GUIMSG_NO_FINGER	1	No finger detected.
PT_GUIMSG_TOO_LIGHT	2	Finger image is too light.
PT_GUIMSG_TOO_DRY	3	Finger is too dry.
PT_GUIMSG_TOO_DARK	4	Finger image is too dark.
PT_GUIMSG_TOO_HIGH	5	Finger is too high.
PT_GUIMSG_TOO_LOW	6	Finger is too low.
PT_GUIMSG_TOO_LEFT	7	Finger is too left.
PT_GUIMSG_TOO_RIGHT	8	Finger is too right.
PT_GUIMSG_TOO_SMALL	9	Finger image is too small.
PT_GUIMSG_TOO_STRANGE	10	Finger image is too strange.
PT_GUIMSG_BAD_QUALITY	11	Finger has bad quality.
PT_GUIMSG_PUT_FINGER	12	Put finger 1st time.
PT_GUIMSG_PUT_FINGER2	13	Put finger 2nd time.
PT_GUIMSG_PUT_FINGER3	14	Put finger 3rd time.
PT_GUIMSG_REMOVE_FINGER	15	Remove finger.
PT_GUIMSG_CONSOLIDATION_FAIL	16	Multiple enrollment failed.
PT_GUIMSG_CONSOLIDATION_SUCCESS	17	Multiple enrollment succeed.
PT_GUIMSG_CLEAN_SENSOR	18	Clean the sensor.
PT_GUIMSG_KEEP_FINGER	19	Keep finger on the sensor.
PT_GUIMSG_START	20	GUI starts now.
PT_GUIMSG_VERIFY_START	21	GUI starts now for verify operation.
PT_GUIMSG_ENROLL_START	22	GUI starts now for enroll operation.
PT_GUIMSG_FINGER_DETECT_START	23	GUI starts now for detect finger operation.
PT_GUIMSG_GUI_FINISH	24	GUI ends now.
PT_GUIMSG_GUI_FINISH_SUCCEED	25	GUI ends now after success.
PT_GUIMSG_GUI_FINISH_FAIL	26	GUI ends now after failure.
PT_GUIMSG_CALIB_START	27	GUI starts now for sensor calibration.
PT_GUIMSG_TOO_FAST	28	Too fast finger swipe.
PT_GUIMSG_TOO_SKEWED	29	Too skewed finger swipe.
PT_GUIMSG_TOO_SHORT	30	Too short finger swipe.
PT_GUIMSG_TOUCH_SENSOR	31	Touch the sensor. Used on strip sensor to distinguish from PUT_FINGER = swipe finger.
PT_GUIMSG_PROCESSING_IMAGE	32	Finger image passed preliminary check and now is about to enter template extraction, which takes approx. 1 second.
PT_GUIMSG_SWIPE_IN_PROGRESS	33	Finger was detected and is about to be scanned. The application must react very quickly to this message, otherwise a part of finger image may get lost.

Note: Please note that not all of the defined GUI codes are really used. Some of them are defined for future use or used connection with other sensors only.

## 4. Interface Function Prototypes (Smart Card) [Proprietary Driver Only]

Generally, a program is required to call `AC_Open` first to obtain a handle. The handle is required for subsequent calls to `AC_StartSession`, `AC_ExchangeAPDU`, `AC_EndSession` and `AC_Close`. The inserted card can be powered up by using the `AC_StartSession` function and card commands can be exchanged with the inserted card using the `AC_ExchangeAPDU` function. Moreover, `AC_SetOptions` and `AC_GetInfo` are two commands that can be used to set and read the various information of the reader.

### 4.1 AC\_Open

This function opens a port and returns a valid reader handle for the application program.

#### Format:

```
INT16 AC_DECL AC_Open (INT16 ReaderType, INT16 ReaderPort);
```

#### Input Parameters:

The table below lists the parameters for this function (you can refer to TFM.H for the corresponding value):

Parameters	Definition / Values
ReaderType <sup>3</sup>	AET63 = AET63 BioTRUSTKey
ReaderPort	The port that is connected with the reader. <i>AC_USB</i> = Using the USB communication port

#### Returns:

The return value is negative and contains the error code when the function encounters an error during operation. Otherwise, it returns a valid reader handle. Please refer to appendix A for the detailed description and meaning of the error codes.

#### Examples:

```
// Open a port to an AET63 connected to USB
INT16 hReader;

hReader = AC_Open(AET63, AC_USB);
```

#### Remarks:

When the application wants to access the security module, it needs to open (use the `AC_Open` command) the reader for the second time to get a different handler for the handling of the security module session.

## 4.2 AC\_Close

This function closes a previously opened reader port.

### Format:

```
INT16 AC_DECL AC_Close (INT16 hReader);
```

### Input Parameters:

The table below lists the parameters for this function

Parameters	Definition / Values
HReader	A valid reader handle previously opened by AC_Open()

### Returns:

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

### Examples:

```
// Close a previously opened port
INT16 RtnCode;
RtnCode = AC_Close(hReader);
```

## 4.3 AC\_StartSession

This function starts a session with a selected card type and updates the session structure with the values returned by the card Answer-To-Reset (ATR). A card reset starts a session and it is ended by either another card reset, a power down of the card or the removal of a card from the reader. Note that this function will power up the card and perform a card reset.

### Format:

```
INT16 AC_DECL AC_StartSession (INT16 hReader, AC_SESSION FAR *Session);
```

### Input Parameters:

The table below lists the parameters for this function:

Parameters	Definition / Values
hReader	A valid reader handle returned by AC_Open()
Session.CardType	The selected card type for this session (AC_T0 for T=0 card, AC_T1 for T=1 card, and "0" for auto detect MCU smart card)
Session.SCModule	The selected security module number

Parameters	Definition / Values
	(Required only when card type =AC_SModule)

### Output Parameters:

The table below lists the parameters returned by this function:

Parameters	Definition / Values
Session.ATR	Answer to Reset returned by the card
Session.ATRLen	Length of the answer to reset
Session.HistLen	Length of the historical data
Session.HistOffset	Offset of the historical data
Session.APDUlenMax	Maximum length of APDU supported

### Returns:

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

### Examples:

```
// Prepare Session structure for T=0 smart card
INT16 RtnCode,i;
AC_SESSION Session;

Session.CardType = AC_T0; // Card type : T=0

//Start a session on previously opened port
RtnCode = AC_StartSession(hReader, &Session);

// Print the card ATR
printf("Card Answer to Reset : ");
for (i = 0; i < (INT16) Session.ATRLen; i++)
    printf(" %02X",Session.ATR[i]);
```

### Remarks:

When no card type is selected (i.e. Session.CardType = 0), the reader will try to detect the inserted card type automatically. However, while the reader can distinguish the T=0 card, T=1 card and synchronous memory card, it cannot distinguish different types of memory card.



## 4.4 AC\_EndSession

This function ends a previously started session and powers off the card.

### Format:

```
INT16 AC_DECL AC_EndSession (INT16 hReader);
```

### Input Parameters:

The table below lists the parameters for this function:

Parameters	Definition / Values
HReader	A valid reader handle returned by AC_Open()

### Returns:

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

### Examples:

```
//End session on a previously started session
RtnCode = AC_EndSession(hReader);
```

## 4.5 AC\_ExchangeAPDU

This function sends an APDU command to a card via the opened port and returns the card's response.

### Format:

```
INT16 AC_DECL AC_ExchangeAPDU (INT16 hReader, AC_APDU FAR *Apdu);
```

### Input Parameters:

The table below lists the parameters for this function:

Parameters	Definition / Values
hReader	A valid reader handle returned by AC_Open()
Apdu.CLA	Instruction Class
Apdu.INS	Instruction Code
Apdu.P1	Parameter 1
Apdu.P2	Parameter 2
Apdu.DataIn	Data buffer to send
Apdu.Lc	Number of bytes in Apdu.DataIn to be sent
Apdu.Le	Number of bytes expected to receive

**Output Parameters:**

The table below lists the parameters returned by this function:

Parameters	Definition / Values
Apdu.DataOut	Data buffer containing the card response
Apdu.Le	Number of bytes received in Apdu.DataOut
Apdu.Status	Status bytes SW1, SW2 returned by the card

**Returns:**

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

**Examples:**

```
// Read 8 bytes of data from T=0 card
INT16 RtnCode,i;
AC_APDU Apdu;

Apdu.CLA = 0x00; // Instruction Class
Apdu.INS = 0xB0; // INS = Read File
Apdu.P1 = 0x00; // MSB of starting address
Apdu.P2 = 0x00; // LSB of starting address
Apdu.Lc = 0x00; // No input data for this command
Apdu.Le = 0x08; // Read 8 bytes data

//Exchange APDU with AET63
RtnCode = AC_ExchangeAPDU(hReader, &Apdu);

if (RtnCode >= 0)
{
    // print the data
    printf("Data : ");
    for (i=0; i < (INT16) Apdu.Le; i++)
        printf(" %02X",Apdu.DataOut[i]);
    // print the status bytes
    printf("Card Status (SW1 SW2)=%04X",Apdu.Status);
}
}
```

**4.6 AC\_GetInfo**

This function retrieves information related to the currently selected reader.

**Format:**

```
INT16 AC_DECL AC_GetInfo (INT16 hReader, AC_INFO FAR *Info);
```

**Input Parameters:**

The table below lists the parameters for this function:

Parameters	Definition / Values
Hreader	A valid reader handle returned by AC_Open()

**Output Parameters:**

The table below lists the parameters returned by this function:

Parameters	Definition / Values
Info.szRev	Revision code for the selected reader.
Info.nMaxC	The maximum number of command data bytes.
Info.nMaxR	The maximum number of data bytes that can be requested to be transmitted in a response
Info.CType	The card types supported by this reader
Info.CStat	The current status of the reader 00 <sub>H</sub> = no card inserted 01 <sub>H</sub> = card inserted, not powered up 03 <sub>H</sub> = card powered up
Info.CSel	The currently selected card type
Info.nLibVer	Current library version (e.g. 310 is equal to version 3.10)
Info.lBaudRate	The current running baud rate

**Returns:**

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

**Examples:**

```
// Get the revision code of the currently selected reader
INT16 RtnCode;
AC_INFO Info;

RtnCode = AC_GetInfo(hReader, &Info);
printf("Reader Operating System ID : %s", Info.szRev);
```

**4.7 AC\_SetOption**

This function sets various options for the reader.

**Format:**

INT16 AC\_DECL AC\_SetOptions (INT16 hReader, WORD16 Type, WORD16 Value);

**Input Parameters:**

The table below lists the parameters for this function

Parameters	Definition / Values
HReader	A valid reader handle returned by AC_Open()

Parameters	Definition / Values
	(except for the ACO_RESET_READER option)
Type	Type of option that is going to set
Value	Value parameter for the selected option type

**Returns:**

The return value is zero if the function is successful. Otherwise, it returns a negative value meaning that the option setting is not available.

**Options:**

Options	Type	Value
<u>Enable</u> the reader to issue the <u>GET_RESPONSE</u> command <u>automatically</u> (only valid for the MCU card)	ACO_ENABLE_GET_RESPONSE	SW1 + "00" (GET_RESPONSE will be issued automatically when this SW1 is returned from the card)
<u>Disable</u> the <u>automatic</u> issue of the <u>GET_RESPONSE</u> command (this is the default option of the reader)	ACO_DISABLE_GET_RESPONSE	0
Check the reader is supporting the "eject card" option or not*	ACO_GET_READER_CAPABILITIES	0
Enable / Disable card insertion / removal notification message	ACO_SET_NOTIFICATION	1 =enable notification 2 =disable notification

\* Function return 0 when that option is supported, otherwise it is not supported

**Examples:**

```
// Set the AET63 to disable the automatic issue of the GET_RESPONSE command
INT16 RtnCode;
```

```
RtnCode = AC_SetOption(hReader, ACO_DISABLE_GET_RESPONSE, 0);
if (RtnCode < 0)
    printf("Set option failed\n");
```

## 5. Interface Function Prototypes (EEPROM) [Proprietary Driver Only]

There are two functions for user to read and write the EEPROM insider the reader, namely *AC\_ReadEEPROM* and *AC\_WriteEEPROM*. Similar to the functions mentioned in previous section, the handle for these two functions should be obtained by *AC\_Open* and released by *AC\_Close*.

### 5.1 AC\_ReadEEPROM

This function reads out the content from the EEPROM.

#### Format:

```
DLLAPI INT16 AC_DECL AC_ReadEEPROM( INT16          hReader,
                                   UINT16   addr,
                                   UINT16   len,
                                   UINT8*    buf,
                                   UINT16*   pDataLen);
```

#### Input Parameters:

The table below lists the parameters for this function:

Parameters	Definition / Values
hReader	A valid reader handle returned by <i>AC_Open()</i>
addr	EEPROM address
len	Number of bytes to read in
buf	Data buffer for storing the content
pDataLen	Pointer to UINT16 variable for storing the total number of bytes read in

#### Returns:

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

#### Examples:

```
// Read in EEPROM content from address 0x100, 10 bytes
UINT8    dataBuf[10];
UINT16   dataLen;
INT16    RtnCode;
int      i;

RtnCode = AC_ReadEEPROM(hReader, 0x100, 10, dataBuf, &dataLen);
if (RtnCode >= 0)
{
    // print the data
    printf("Data : ");
    for (i=0; i < 10; i++)
    {
        printf(" %02X",dataBuf[i]);
    }
}
```

## 5.2 AC\_WriteEEPROM

This function writes out the content to the EEPROM.

### Format:

```
DLLAPI INT16 AC_DECL AC_WriteEEPROM(INT16          hReader,
                                     UINT16   addr,
                                     UINT16   len,
                                     UINT8*   buf);
```

### Input Parameters:

The table below lists the parameters for this function:

Parameters	Definition / Values
hReader	A valid reader handle returned by AC_Open()
addr	EEPROM address
len	Number of bytes to write out
buf	Data buffer for storing the content

### Returns:

The return value is zero if the function is successful. Otherwise, it returns a negative value containing the error code. For the detailed meaning of the error code, please refer to appendix A.

### Examples:

```
// Write out 10 bytes data to EEPROM address 0x100
UINT8   dataBuf[10];
INT16   RtnCode;

RtnCode = AC_WriteEEPROM(hReader, 0x100, 10, dataBuf);
if (RtnCode < 0)
{
    printf("Write EEPROM failed(%d)\n", RtnCode);
}
```

## Appendix A: Table of Error Codes

Code	Meaning
-603	Error in the reader handle
-600	Session parameter is null
-108	No free handle left for allocation
-100	Selected port is invalid
-101	Selected reader is invalid
-102	Selected port is occupied
-1001	No card type selected
-1002	No card is inserted
-1003	Wrong card type
-1004	Card not powered up
-1005	INS is invalid
-1006	Card failure
-1007	Protocol error
-1008	Card type not supported
-1009	Incompatible command
-1010	Error in address
-1011	Data length error
-1012	Error in response length
-1013	Secret code locked
-1014	Invalid SC module number
-1015	Incorrect password
-1050	Error in CLA
-1051	Error in APDU parameters
-1052	Communication buffer is full
-1053	Address not align with word boundary
-1080	Protocol frame error
-1081	No response from reader
-1082	Error found in the calling function's parameters
-1083	Specified function not supported
-1084	Connector short circuit
-1085	Unexpected internal error
-1086	A required DLL file is missing
-1099	Unknown response
-2000	USB internal error
-2001	Error in memory allocation

-2002	Error in linking USB library
-2003	Error in locating window system directory
-3000	Error found in PCSC smart card manager