Serial CCID and Application Layer Interface Library Kit

-

Programmer's Addendum to 'GBP Interface Library Kit' document

-

Version 2.1

TABLE OF CONTENTS

1 Introduction

1.1 Object

This document describes the different interface levels available to develop an application driving a Gemplus device.

1.2 Scope

The initial project GBP Interface Library Kit targets Gemcore based Gemplus devices.
This update of the library has the following goals:

- keep compatibility with existing layers,
- deliver Entry Points to manage:
  - Serial CCID Gemplus devices,
  - USB CCID Gemplus devices (CCID layer is generic, lower layers are delivered for a demonstration purpose only, based on the use of a Windows XP compliant Windriver USB driver)
- deliver a generic upper API.

1.3 Reference documents

The reference documents are:

Ref Nb	**Title**
[R1]	GBP Interface Library Kit, Programmer's Guide, Version 1.0
[R2]	USB Device Class Specification for USB Chip/Smart Card Interface Devices
[R3]	Serial communication on Combo Reader

2 Architecture

2.1 Schema

PSEUDO ISO LAYERS	GEMCORE based devices	CCID based devices		Common
APPLICATION		**AL**: Application entry points		
				AL_APDU_TPDU: APDU mapper to TPDU
PRESENTATION	**PreGEMCORE**: GEMCORE interface	**PreLCCID**: CCID interface (partial)		
TRANSPORT	**TL**: GBP and TLP protocols	**TLCCIDGe**: Generic CCID protocol		**ApiSystm**
		TLSeCCID: Serial CCID protocol	**TLCCID**: CCID protocol	
PHYSICAL	**VP**: Virtual Port		RawWinDv.lib	
	PP: Physical Port Layer		Windriver	

COLOR	*Description*
BOLD	**Associated files names**
GREEN	Miscellaneous system API
BLUE	Dedicated CCID & Application newly implemented files
BLUE	Dedicated CCID & Application newly implemented files
PINK	Sample USB implementation Windriver based under Windows XP
RED	Not yet existing files that could be useful to the project

This table associates communication layers with already existing and newly implemented interfaces.

2.2 Files organization

The three main directories of the delivery are:

- ‘**Doc**’: include this document,
- ‘**Windriver**’: include required information how to install this sample driver with a targeted USB CCID device, Windows 32bit specific,
- ‘**Generic**’: include the two main directories of this project, organized as follow:
    - files to generate the CILK Library: ‘CILK’,
    - files delivered as an application sample using this library: ‘ApSample’.

2.2.1 CILK directory

The directories are arranged as follow:

- Source and Header files: ‘Code’,
- Project and compiled files: ‘Project’,
- Library sample USB Windriver based: ‘Library’.

2.2.1.1 Code directory

This directory is composed of the Source and Header files.
Almost all of them are platform and OS independent.
The exception is for the Physical Layer files:

- ‘PP.c’,
- ‘PP.h’,

which must be chosen and possibly adapted depending on the target platform/OS.
Four implementations are delivered as sample located in the sub-directories:

- ‘WIN_32’: implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 6.0, so in 32bit environment,
- ‘WIN_16’: implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 1.52, so in 16bit environment,
- ‘LINUX’: delivered but not tested,
- ‘TEMPLATE’: reduced to upper layer entry points, to implement according to target environment.

Modus operandi:

First Step:

Copy the files:

- **‘PP.c’,**
- **‘PP.h’,**

from the specific targeted platform implementation sub-directory (example: ‘WIN_32’) to the same directory as the other Source and Header files.

Second Step:

Under another than the WIN_32 environment, the sample Windriver USB management is not supported, so:

- **set in comments** the line at the beginning of the file **‘TLCCID.h’**:

#define WIN32_USB_WINDRIVER

while the corresponding code is not supported,

- **remove from the CILK project the associated library file: ‘RawWinDv.lib’**.

Third Step:

For Debug Serial communication purpose only, in the file 'VP.h', it is possible to activate communication traces, printf() based using default file names, while removing the comment for:

//#define PRINTF_DEBUG_TRACES

Fourth Step:

In the case you are not interested in the Secure functions, you can set in comments in the file 'PreLCCID.h':

#define USING_SECURE_FUNCTION

Fifth Step:

Depending on your target, you have to update the single define that must used within the file 'ApiSystm.c':

#define WIN_32_API
#define WIN_16_API

And eventually create a new specific one –together with the associated code- adapted to your system API used (see code inside).

2.2.1.2 Project directory

This directory intends to contain the project files of the CILK Library.
Each sub-directory is associated with one platform environment. Actually two are implemented:

- 'CILK_32': implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 6.0, so in 32bit environment,
- 'CILK_16': implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 1.52, so in 16bit environment.

These contain the list of the files of the project (located in '..\Code'), the output file name, the makefile, …. and finally the **CILK library**.

2.2.1.3 Library directory

This directory intends to contain the library files of the CILK Library.
Each sub-directory is associated with one platform environment. Actually only one is implemented:

- 'WIN_32': implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 6.0, so in 32bit environment.

It is Windriver Specific and contains the library file and its associated Header file. Included within an upper layer, they enable to communicate with an USB Windriver installed device.

2.2.2 ApSample directory

This directory is only intended to be an application sample using the CILK Library.
The directories are also arranged as follow:

- Source and Header files: 'Code',
- Project and compiled files: 'Project'.

2.2.2.1 Code directory

This directory is composed of the Source and Header files.
The same application code is used on 16-bit and 32-bit environment, thanks to the portage of system functions in a dedicated file 'ApiSystm.c'

2.2.2.2 Project directory

This directory intends to contain the project files of the Sample application.
Each sub-directory is associated with one platform environment. Actually two are implemented:

- 'Sampl_32': implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 6.0, so in 32bit environment,
- 'Sampl_16': implemented and tested under a PC using Windows XP SP2 and with the development suite Visual C++ 1.52, so in 16bit environment.

These contain the list of the files of the project (located in '..\Code'), **the CILK library included in the project**, the output file name, the makefile, …. and finally the **Executable file**, to run according the program parameters (see Usage() function).

2.3 GILK already existing files

2.3.1 Based project

A detailed description of the project, its goal, its delivery is available at the following link:

`http://www.gemplus.com/techno/gbp_libraries/`

2.3.2 File adaptations

The initial release of these files is described in the document [R1].
Followings are only required file updates to integrate new functionalities.

2.3.2.1 PP

- **`PP_OpenComm()`**

TWOSTOPBITS rather than ONESTOPBITS in case upper than 38400 bps baudrate, required by our Serial CCID device.
ReadIntervalTimeout changed from 2 ms to 60 ms.

- **`PP_ReadComm()`**

The parameter 'puiSize' enables to specify the exact length of data to read.
If equal to 'READING_LENGTH_CONTROL_BY_CALLBACK', behaviour unchanged compare to initial file. The length of the frame to receive is analysed byte per byte by the call back function.
If different, only up to the specified length is read, commonly in one operation.
The reason for this change is that in case Serial CCID protocol, the physical layer cannot know the expected length to read.

2.3.2.2 VP

The define:

PRINTF_DEBUG_TRACES

Is added for debug purpose only. If compiled with, raw data exchanged are written in two cyclic files. See coding for details and associated code should be removed for a customer delivery.

- **`VP_Receive()`**

Its own resources are no ore used.
Parameters coming from the upper layer (Transport) are directly used for the lower one (PP), enabling the transport layer to specify the length of the data to read.
This also avoids the end up copy of data.

2.3.2.3 TL

- **`TL_Receive()`**

Unchanged, thanks to the parameter 'usiLenResponse = GBP_SIZE' which enables the use of the call back function as the initial implementation (see PP_ReadComm() for more details).

2.4 CCID, Serial CCID, CCID Generic, CCID Presentation and AL project

2.4.1 ApiSystm

2.4.1.1 Functionality

This file gives to the project an abstract of some system API which implementation may change depending on the target used.

2.4.1.2 Outside world

- **`ApiSystm_Sleep()`**

Wait for a given delay.

- **`ApiSystm_getchar()`**

Wait for a character input

- **`ApiSystm_ getstring()`**

Wait for a string input

- **`ApiSystm_ toupper()`**

Return the upper value of one character.

- **`ApiSystm_ memset()`**
- **`ApiSystm_ memcpy()`**
- **`ApiSystm_ memcmp()`**
- **`ApiSystm_ strlen()`**
- **`ApiSystm_ strcmp()`**

Implement as close as possible the ANSI definition of the associated function.

2.4.1.3 Inside implementation

Only brief overview, details are inside the code.

- **`Local_getstring()`**

Enable to catch an input string with specific parameters.

2.4.2 TLCCID

2.4.2.1 Functionality

This layer is at the same protocol level as the TL files.
It implements the CCID protocol as described in the document [R2].

2.4.2.2 Outside world

- **`TLCCID_SendReceive()`**

This function exchanges one CCID command/response.

- **`TLCCID_SetupCommand()`**

This function exchanges one specific USB Setup command/response.

2.4.2.3 Inside implementation

Only brief overview, details are inside the code.
No more function implemented.
They are based on the use of the sample Windriver library.

2.4.3 TLSeCCID

2.4.3.1 Functionality

This layer is at the same protocol level as the TL files.
It implements the Serial CCID protocol as described in the document [R3].

2.4.3.2 Outside world

- **`TLSeCCID_SetCommandEcho()`**

Update the dedicated variable which enables to manage two different device behaviours.
GemPCTwin family device echo the command in the reception buffer while GemcorePOSPro do not.
An application must call this function once at the beginning of the program. It enables to set up the communication behaviour with the target device one.

- **`TLSeCCID_GetCommandEcho()`**

This function retrieve the current library behaviour concerning the command echo management in the case an application might need this information.

- **`TLSeCCID_SendReceive()`**

This function exchanges one CCID command/response. It implements the specific protocol error recovery, up to MAXTRY Nack frame exchanges.
It also implements the specific T=1 time extension management, consisting in only waiting for the next frame while updating the timeout according to the time extension request received.

2.4.3.3 Inside implementation

Only brief overview, details are inside the code.

- **`TLSeCCID_Make()`**
- **`TLSeCCID_Unmake()`**

Constructing/extracting the Serial CCID frames.

- **`TLSeCCID_Receive()`**

Equivalent to the initial call back function. Enables to receive the CCID Serial response while previously littering:

Possible card movements notification before command echo,
The echoed command,
Possible card movements, time extensions before the response.

- **`Static TLSeCCID_EmptyReceptionBuffer()`**

Removing remaining lost bytes in the reception buffer in order to resynchronise communication with the device.

- **`Static TLSeCCID_LookingForSynchByte()`**

Removing on the fly possible card movement and T=0 time extension notifications before the beginning of the frame sent by the device.
For each T=0 time extension received, the timeout is restarted, so implicitly managed by design.

- **`TLSeCCID_EndOfReceipt()`**

Do nothing except returning 'NO_ERR'. Only for compatibility with lower layers and initial implementation.

2.4.4 TLCCIDGe

2.4.4.1 Functionality

This layer is at the same protocol level as the TL files, just above two previous one.
Its goal is to select the appropriate lower layer (Serial or USB) depending on the device selected.

2.4.4.2 Outside world

- **`TLCCIDGeneric_SendReceive()`**

This function exchanges one CCID command/response, notwithstanding the kind of device, Serial or USB.

- **`TLCCIDGeneric_SetupCommand()`**

This function exchanges one CCID Setup command, only available for USB devices.

2.4.4.3 Inside implementation

Only brief overview, details are inside the code.
No more function implemented.
They are based on the use of the just lower layer functions.

2.4.5 PreLCCID

2.4.5.1 Functionality

Two main functionalities:

- Construction/extraction of CCID commands/responses according to the document [R2]. List not exhaustive but restricted to the only needs of the upper layer,
- Deliver to the upper layer the associated resources (buffers).

2.4.5.2 Outside world

- **`PreLCCID_SetSlotNb()`**

Update the current Slot Number.

- **`PreLCCID_InitializeSequenceNb()`**

Perform once a random initialization of the internal parameter SequenceNumber.
The goal is not this library based application to start with always the same value for this parameter. Else, it might cause problem in case only one exchange with a device would be performed, and the application is stopped and run again.

- **`PreLCCID_PC_to_RDR_IccPowerOn()`**

Construct a PowerOn command.

- **`PreLCCID_PC_to_RDR_IccPowerOff_GetSlotStatus()`**

Construct a PowerOff or a GetSlotStatus command.

- **`PreLCCID_PC_to_RDR_GetParameters()`**

Construct a GetParameters command.

- **`PreLCCID_PC_to_RDR_SetParameters()`**

Construct a SetParameters command.

- **`PreLCCID_PC_to_RDR_XfrBlock()`**

Construct an XfrBlock command.

- **`PreLCCID_PC_to_RDR_Escape()`**

Construct an Escape command.

- **`PreLCCID_SetupCommand()`**

Construct a Setup command.

- **`PreLCCID_RDR_to_PC_CheckIntegrity()`**

Extract a CCID response, check its integrity and copy optional data.

- **`PreLCCID_PC_to_RDR_Secure()`**

Construct a Secure command based on the CCID specifications and taking into account the Gemplus reader GemPCPinpad specific choices (see code). Two mains parameters are implemented:

SECURE_PIN_VERIFICATION,
SECURE_PIN_MODIFICATION.

2.4.5.3 Inside implementation

Only brief overview, details are inside the code.
Implementation of the listed above functions.

- **`PreLCCID_GetSlotNb()`**
- **`PreLCCID_SetSequenceNb()`**
- **`PreLCCID_GetSequenceNb()`**

In case an application would need to update these two global parameters CCID specific.

- **`Static PreLCCID_InitResources()`**

Allocating resources for the upper layer according to the CCID (Header always 10 bytes) and application (max short APDU length) requirements.

- **`Static PreLCCID_InitCCIDHeader()`**

Default init of a CCID Header.

- **`Static PreLCCID_InitXfrBlockEscape()`**

Common coding to construct an XfrBlock or an Escape command.

2.4.6 AL

2.4.6.1 Functionality

Almost the one and only one interface to be used by the customer.
Application error codes are also defined.

2.4.6.2 Outside world

- **`InitCallBackDisplayMsg()`**

Usage dedicated with the Windriver library.

It enables the application to rewrite its own display messages function and assign the right callback function pointer to the lower layers. The default implementation is printf() based.

- **`ExchangeAPDU()`**

Performs an ICC APDU exchange.

- **`GetFirmwareVersion()`**

Returns the firmware version. In case an application would like to get this information for any purpose.

- **`SetGetReaderMode()`**

Setter and Getter for the APDU or TPDU device mode.
It is advised to set it in APDU mode after the card activation and before the first card data exchange.In that case, APDU are exchanged in one command/response.
If the application needs to manage the card Transport Layer (T=0 GetResponse, T=1 chainings for example), choose TPDU.

- **`OpenAndInit()`**

Kept for compatibility and dedicated to Serial CCID devices. A generic application should not use this API but rather the following one, which internally refers to this one.
Standard communication parameters update.
One specific escape command is also sent to change the default reader behaviour:

  - Feature Card switch synchronous: card movement notifications will only be provided together with the next response to a command by the reader.

- **`Close()`**

Reciprocal of the previous function: kept for compatibility, advised not an application call it but the following one which internally refers to it.
It returns reader in its default behaviour and closes the Serial port communication.

- **`OpenReaderComm()`**

This is the first API that an application should call to communicate with a CCID device, Serial or USB.
Depending on the entry parameters, the communication with the associated kind of device is opened.

- **`CloseReaderComm()`**

Close the the communication with the previously opened device.

- **`IccPowerOn()`**

Performs an ICC activation. Voltage, NAD and PPS management values are parameters IN.
Sequence of operation:

  - Set reader in TPDU mode avoiding rejecting cards not Mastercard compliant in APDU mode
  - Perform card activation
  - Analyse ATR: initialising parameters to set to device depending on card declaration like specific mode, protocol, timeouts ans so on…
  - Set Parameters to the device according to the previous analysis result.

In the case an automatic PPS is requested, a PPS exchange is generated according to the following rules:

- The ATR declares the Negotiable Mode,
- A TA1 different from the default value or more than one communication protocol is declared (avoid useless PPS exchange),
- In the case the TA1 declared might result in a slower or equal baudrate than the default one, it is also selected for the PPS exchange. The reason for this behaviour is that typically should not happen. Filtering this case would require to report in this library the ISO 7816-3 FI DI tables to get the resulting baudrate for each TA1 value, increasing code almost never used,
- In the case both communication protocols T=0 and T=1 are declared, T=1 is selected for the PPS exchange.

In the case no PPS is requested, default parameters are in use or the Specific Mode is set up.

- **`IccPowerOff()`**

Performs an ICC deactivation.

- **`GetStatus()`**

Update `fCardInserted` and `fCardPowered` parameters according to the card state.

- **`SetupCommand()`**

Send an USB specific Setup command.

- **`SecureMessageSelectList()`**

Select internal or flash memory list messages.

- **`SecureMessageUpdate()`**

Update one external message list.

- **`SecureCommand()`**

Perform an ICC Secure APDU exchange. Two mains parameters are implemented:
SECURE_PIN_VERIFICATION,
SECURE_PIN_MODIFICATION.
Specific code is implemented to update on the fly reader command timeouts together with the requested one for the PIN entry (verify/modify, number max of digit, inter-character delay, …).

2.4.6.3 Inside implementation

Only brief overview, details are inside the code.
Implementation of the listed above functions. Whole of them are based on the same skeleton:

- check parameters,
- construct the CCID associated command,
- perform the Serial CCID exchange,
- check the CCID response,
- update optional data,
- return execution status (OK or Application defined error code).

- **`Static CheckSendReceiveErrorStatus()`**

In case of no response to a command, the application must be aware of this to take the good decision towards the card. This function only distinguishes this case from the other communication errors.

- **`Static ExchangeDATA()`**

Generic function for both ExchangeAPDU and ExchangeESCAPE commands.

- **`Static ExchangeESCAPE()`**

Perform a device Escape command exchange.
Only implicitly required by the customer API. This generic function enables the specific following implementations…

- **`Static SetCardMovementNotification()`**

Select the Serial card movement notification as Synchronous (at most one –current card presence- just before the next device response) or asynchronous (as many as real card movement, any when even during command reception which might cause loss of the command; unwise while the goal of the application is not to specifically check the card movements).

- **`Static InitializeCCIDSequenceNb()`**

While opening the device communication, on the first use of the library, the CCID Sequence Number parameter is randomly initialized avoiding always starting with the same value.

- **`Static SetGetParameters()`**

Setter and Getter for the device Parameters. Specific for T=0 and T=1 card protocol.

- **`Static AnalyseATRInitParameters()`**

ATR analysis to initialise parameters for the previous function.

- **`Static ExchangePPS()`**

Perform a PPS exchange.

- **`Static GenericIccPowerOffGetStatus()`**

Generic function for both IccPowerOff and GetStatus commands.

2.5 Identified missing files out of the scope for this project

2.5.1 AL_APDU_TPDU

APDU management that could be performed in this API, same way as drivers do.
Actually not planned.

2.5.2 PreGEMCORE

Same as constructing CCID commands/responses, but applied to Gemcore API.

May become easier for customers to use that kind of library.
Actually not planned.

2.6 Sample application program

2.6.1 Actual release

Files to include:
#include "..\..\CILK\Code\TLSeCCID.h"
#include "..\..\CILK\Code\AL.h"

And file to insert into the project:
CILK_32.lib or CILK_16.lib.

Using the command line parameters (see Program Usage()), if whole of them are available, this program opens the communication with the chosen device, displays these parameters, the current firmware release and performs a loop displaying on the fly the current available menu. The default behaviour is the following sequence once:

- GetStatus()
- IccPowerOn()
- SetGetReaderMode(SET_APDU_MODE)
- ExchangeAPDU()
- IccPowerOff() one loop out of two

and displays immediate results on the associated console application.

3 Customers updating from GILK to CILK

There is an ascending compatibility when updating from GILK (GBP protocol) to CILK (Serial CCID protocol). This means that after a CILK project has replaced a GILK one, physical layers VP and PP are compatible with both layers TL, TLSeCCID and upper ones.

But how updating a customer GILK based project to a new CILK one?
Following are the different tasks to perform:

- start from a new CILK project,
- choose the closest ‘PP.c’ file to the project configuration (WIN16, WIN32, …) and copy it as explained before (see **Modus operandi**) in the same directory as the other files of the project,
- update this ‘PP.c’ file according with the already adaptation performed within this same file coming from the GILK project. Take care that the older one cannot directly replace the newest one while there are some minor API and functionalities changes,
- the other files of the CILK project should not change,
- by the application side, older Gemcore commands like ‘0x12’, ‘0x15’ must be updated using the new high level offered API IccPowerON(),ExchangeAPDU()…

In résumé, firstly the lowest layer ‘PP.c’ of a new CILK project requires to be adapted as previously done for a GILK project.

Secondly, the customer Application side has to use the new set of high level API, written to be easy to use and offering parameters as close as possible to the Standard CCID functions.

4 Limitations

Take care with 115200 bps baud rate. At Application Layer, a 16-bit compatible define is offered but lower layers require the use of this value 115200 which does not stand on a 16-bit value, the upper value being 0xFFFF=65535.
It is suggested to use the just lower standard value 57600 bps to avoid this specific limitation.

There is no dynamic allocation of the resources. Initially there is not with the GILK project. It could be useful while length of CCID frames is coded on 4 bytes. It is considered as useless for us while the longer CCID frames used will only be short APDU.

CCID lengths more than a 16-bit environment could manage (65535) are not implemented and considered as error.