Sankyo

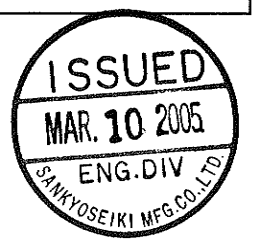
DRIVER SPECIFICATION

Type Name :

ICT3K5-6290DLL

Spec. No.	ASL-NP-07652-52

Notes :



Distribution	No. of sheet	Custody
SGH	1	ICT3K5-3R6290 Type Name

Sym.	Revision		Page	Cha. No.	Date	By	Appr.
Design	M.Kimura	Mar. 1. '05					
Det-che	T.Komatsu	Mar. 4. '05					
Appro.	Y.Uemura	Mar.04.2005					

Reference Document

Sankyo Seiki Mfg.Co.,Ltd. JAPAN

REGISTERED MAR.7.2005 ENG.DIV

No. S39Z-032-00-02C

Revision History on the Document

Revision	Date	By	Descriptions
A	March 1, 2005	Masanori.Kimura	Newly issued

Contents

1. Generals

1.1 Scope

This specification document intends to define the functions, contents, and restrictions regarding the dynamic link library: ICT3K5-6290DLL, which controls the Card Reader/Writer: ICT3K5-3R6290 produced by Sankyo Seiki Mfg. Co., Ltd.

You should understand that, in general, any functions and/or matters that are not described in this document cannot be implemented or their operation results are insecure.

1.2 Conventions of Descriptions in This Document

Unless otherwise especially noted; “Card Reader”, “Card Reader/Writer”, “Target”, “Device”, “Target Device” and other equivalents shall all be understood to be the Card Reader/Writer mentioned in “*1.1 Scope*”.

In this document, Dynamic Link Library is simply called “DLL” as commonly abbreviated. You should understand, therefore, that mentioning “the DLL” without any special note points out the dynamic link library mentioned in “*1.1 Scope*”.

“Application Program” or “Application” in short generally means a program(s) that controls the Card Reader/Writer(s) through the DLL. These expressions can be replaced with “Program using the DLL”.

“API” stands for “Application Programming Interface”. By using APIs, the Application Program makes use of the functions, which the DLL serves.
The functions, which the DLL serves, are materialized as “Commands”. The Application Program makes use of the DLL by calling the commands (functions). Professional-wise, such operations may be described sometimes as; “*The DLL exports functions and the Application Program imports the DLL functions.*”
As mentioned above, the substance of API is functions so that API is sometimes called “API functions”.

“Host Computer” or “Host” in short is connected to the Card Reader/Writer(s) with cable, and it is a computer that controls and operates the Card Reader/Writer(s) by executing the DLL or the Application Program using the DLL.

“OS” stands for “Operating System”.

“Installation of the Card Reader/Writer” or “Installing the Card Reader/Writer” means to carry out a series of preparation work such as; installing the Card Reader/Writer, connecting it to the Host Computer, supplying it with electrical power in order that the Host Computer can control the Card Reader/Writer.

“To control the Card Reader/Writer” means to send commands from the Host Computer for making the Card Reader/Writer execute the specified functions and to eventually have the Host Computer receive the corresponding response so that the Host Computer initiatively controls the Card Reader/Writer to implement the Card Reader/Writer functions.

"API call" or "Calling API" means "API execution" or "Executing API", and they are used to mean just the same thing and can be replaced with each other.

When any related publication(s) or article(s) to be referred to is pointed out in any section other than explanatory parts, a rightward arrow "→" is placed at the top of the indication.

1.3 Notice

Sankyo Seiki Mfg. Co., Ltd. is exempt from any responsibilities, for damages on your system including the host computer(s) and network system, which might have been caused due to installing or using the DLL. Before using the DLL, therefore you are requested to check the operation of the DLL and make sure of the operation contents to ensure that there should occur no problem due to operation of the DLL.

2. Development Environment

2.1 DLL

The DLL is built under the development environment described below:

Microsoft Visual C/C++6.0

2.2 Application Programs

The DLL is coded by using the C++ language. It has already been made sure that the DLL is rightly linked to an application program, built under the same development environment as what the DLL has been built under, and it operates properly.

It has not been checked whether or not the DLL can rightly function when an application program developed under any different environment tries to use the DLL.

3. Execution Environment

3.1 Host Computer

IBM PC/AT compatible machine equipped with one or more USB port(s)

3.2 OS (Operating System)

Microsoft Windows XP

3.3 Language

The DLL operates, being irrelevant to the language specifications of the operating system. In other words, the language specifications of the operating system do not affect the basic operation of the DLL.

The DLL is provided with just one type of resource; i.e., English one, and therefore even if the operating system of the execution environment is not English version, every message is given in English.

4. General Functions

4.1 File Structure

4.1.1 Files and their main functions

Files	Main functions
[1] ICT3K5_6290DLL.dll	This file is an execution module of the DLL. It serves each API's execution code. This file is needed when the Application Program is executed. Revision:2650-01B
[2] ICT3K5_6290DLL.lib	This file is needed when the Application Program, to which the DLL is implicitly linked, is created. It serves information on APIs exported by the DLL. It is not needed when linking is made explicitly.
[3] ICT3K5_6290DLL.h	It is a header file to serve API type declarations, error code definitions and so on. It is needed when the Application Program is created.
[4] PrtclUH.dll	It is a DLL file that operates at a lower layer under the file [1] to serve more fundamental functions. It is needed for executing the file [1]. Revision:2566-01C
[5] CollectLogEx.dll	It is a DLL file that operates at a lower layer under the file [4] to create log files. Revision:2509-02A

4.1.2 Use of the files

The DLL files (.dll) are found and loaded at the time of executing the Application Program according to the procedures and rules specified by the operating system. Therefore, the condition on the folders, in which the DLL files are installed, shall meet the specifications of the operating system.

The files required at the time of developing the Application Program shall be used according to the specifications of the development environment.

4.1.3 Kernel driver

The DLL communicates with the Card Reader/Writer by using a class driver bundled in the operating system. Therefore, it is not needed to install any special kernel driver for executing the DLL.

4.2 Specifications

4.2.1 The number of Card Reader/Writers that can be controlled at the same time

It is possible to boot multiple threads from a process, i.e., the Application Program, for which the DLL is mapped, and to make each thread control each different target.
In the specifications of the DLL, there is no restriction on the number of Card Reader/Writers that can be controlled at the same time.

The resource, which is needed to control a Card Reader/Writer, is dynamically secured as required so that there is theoretically no restriction on the number of Card Reader/Writers that can be controlled. However, practically there appears some restriction due to the specifications of the USB interface and/or the restrictions of resource of the host computer.

4.2.2 Data definition

This section describes the original data types used in the DLL.
The data types, which this section describes, are all defined in the header file (.h); one of the configuring files.
→ Refer to the file [3].

DLL_INFORMATION

```
typedef  struct
{
    struct
    {
        CHAR   szFilename[ _MAX_FNAME];
        CHAR   szRevision[ 32];
    }
    upperDll;

    struct
    {
        CHAR   szFilename[ _MAX_FNAME];
        CHAR   szRevision[ 32];
    }
    lowerDll;
}
DLL_INFORMATION, *LPDLL_INFORMATION;
```

`DLL_INFORMATION` is used when `GetDllInformation` is executed. Each member stores the information described below:

Members		Descriptions
`upperDll`	`szFilename[]`	Character string to indicate the filename of the file [1].
	`szRevision[]`	Character string to indicate the revision number of the file [1].
`lowerDll`	`szFilename[]`	Character string to indicate the filename of the file [4].
	`szRevision[]`	Character string to indicate the revision number of the file [4].

→ Refer to `GetDllInformation`.

COMMAND

```
typedef  struct
{
    BYTE      bCommandCode;
    BYTE      bParameterCode;

    struct
    {
        DWORD   dwSize;
        LPBYTE  lpbBody;
    }
    Data;
}
COMMAND, *LPCOMMAND;
```

COMMAND is used for the purpose of giving the materials needed to compose a command message when calling ExecuteCommand, i.e., a command code, a parameter code, and data.

Members		Descriptions
bCommandCode		Command code
bParameterCode		Parameter code
Data	dwSize	Size of the data (bytes) If there exists no data to give, set dwSize to be 0.
	lpbBody	Pointer to the memory area where the data is saved If dwSize is 0, then lpbBody is ignored.

→ Refer to ExecuteCommand, ExecuteCommand2.

→ Regarding the structure of a command message, refer to the specifications of Card Reader/Writer Interface.

REPLY

```
typedef struct
{
    REPLY_TYPE     replyType;

     union
    {
        POSITIVE_REPLY      positiveReply;
        NEGATIVE_REPLY      negativeReply;
    }
    message;
}
REPLY, *LPREPLY;
```

REPLY is used for the purpose of saving the execution results and contents of the command executed by calling ExecuteCommand.

Members		Descriptions
ReplyType		Execution results of the command → Refer to REPLY_TYPE.
message	PositiveReply	Contents of the execution results in the case of "replyType = PositiveReply" → Refer to POSITIVE_REPLY.
	NegativeReply	Contents of the execution results in the case of "replyType = NegativeReply" → Refer to NEGATIVE_REPLY.

→ Refer to REPLY_TYPE, POSITIVE_REPLY, NEGATIVE_REPLY, ExecuteCommand, ExecuteCommand2,UpdateFirmware, ICCardTransmit, SAMTransmit.

REPLY_TYPE

```
typedef  enum
{
    PositiveReply,
    NegativeReply,
    ReplyReceivingFailure,
    CommandCancellation,
    ReplyTimeout,
}
REPLY_TYPE, *LPREPLY_TYPE;
```

REPLY_TYPE is used for member declaration of the data definition: REPLY, and indicates the execution results of the command notified by response message.

Definitions	Descriptions
PositiveReply	Positive response has been returned.
NegativeReply	Negative response has been returned.
ReplyReceivingFailure	Response receiving has failed.
CommandCancellation	In the condition of waiting for response receiving, CancelCommand has been executed and command execution has been canceled.
ReplyTimeout	In the condition of waiting for response receiving, time-out has occurred.

→ Refer to REPLY.

POSITIVE_REPLY

```
typedef  struct
{
    BYTE   bCommandCode;
    BYTE   bParameterCode;

    struct
    {
        BYTE   bSt1;
        BYTE   bSt0;
    }
    StatusCode;

    struct
    {
        DWORD    dwSize;
        BYTE     bBody[ MAX_DATA_ARRAY_SIZE];
    }
    Data;
}
POSITIVE_REPLY, *LPPOSITIVE_REPLY;
```

POSITIVE_REPLY is used for member declaration of the data definition: REPLY, and it is used for storing the execution results of the command when a positive response is received.

Members		Descriptions
bCommandCode		Command code
bParameterCode		Parameter code
StatusCode	bSt1	Status code: st1
	bSt0	Status code: st0
Data	dwSize	Size of the data (bytes) given by the response message.
	bBody[]	Stores the data body given by the response message.

→ Refer to REPLY and NEGATIVE_REPLY.
→ Regarding the structure of a positive reply message, refer to the specifications of Card Reader/Writer Interface.

NEGATIVE_REPLY

```
typedef  struct
{
    BYTE   bCommandCode;
    BYTE   bParameterCode;

    struct
    {
        BYTE   bE1;
        BYTE   bE0;
    }
    ErrorCode;

    struct
    {
        DWORD    dwSize;
        BYTE     bBody[ MAX_DATA_ARRAY_SIZE];
    }
    Data;
}
NEGATIVE_REPLY, *LPNEGATIVE_REPLY;
```

NEGATIVE_REPLY is used for member declaration of the data definition: REPLY, and it is used for storing the execution results of the command when a negative response is received.

Members		Descriptions
bCommandCode		Command code
bParameterCode		Parameter code
ErrorCode	bE1	Error code: e1
	bE0	Error code: e0
Data	dwSize	Size of the data (bytes) given by the response message.
	bBody[]	Stores the data body given by the response message.

→ Refer to REPLY and POSITIVE_REPLY.
→ Regarding the structure of a negative reply message, refer to the specifications of Card Reader/Writer Interface.

Examples of Command Execution Results to be Stored in REPLY

Example 1) In the case of Positive Response

If the contents of the response message, which returns as the results of executing the command by calling ExecuteCommand, are as described below; ···→

Type of response		Positive response
Command code		12h
Parameter code		34h
Status code	st1	35h (= '5')
	st0	36h (= '6')
Data		A1h, A2h, A3h, A4h, A5h (5 bytes)

···→ contents of the response message are saved into the REPLY type data specified by ExecuteCommand.

.replyType				PositiveReply
.message				
	.positiveReply			
		.bCommandCode		12h
		.bParameterCode		34h
		.StatusCode		
			.bSt1	35h
			.bSt0	36h
		.Data		
			.dwSize	5
			.bBody[ ]	A1h, A2h, A3h, A4h, A5h

Example 2) In the case of Negative Response

If the contents of the response message, which returns as the results of executing the command by calling ExecuteCommand, are as described below; ···→

Type of response		Negative response
Command code		12h
Parameter code		34h
Error code	e1	35h (= '5')
	e0	36h (= '6')
Data		None

···→ contents of the response message are saved into the REPLY type data specified by API.

.replyType				NegativeReply
.message				
	.negativeReply			
		.bCommandCode		12h
		.bParameterCode		34h
		.ErrorCode		
			.bE1	35h
			.bE0	36h
		.Data		
			.dwSize	0
			.bBody[ ]	No data storing

→ Refer to ExecuteCommand, ExecuteCommand2.

4.2.3 APIs

GetDllInformation

```
DWORD   GetDllInformation(
            LPDLL_INFORMATION       lpDllInformation
            );
```

Function

Obtains information on the DLL.

Argument & Detailed Function

lpDllInformation

Information on the DLL is saved in the data set of `DLL_INFORMATION` type to get notified in the format.
→ Refer to `DLL_INFORMATION`.

This API can be executed anytime, regardless of completion of `ConnectDevice`. Even if this API is executed, no communication transaction between the Host Computer and Card Reader/Writer (such as “Command sending – Response receiving”) is caused.

Return Value

_NO_ERROR

The return value of this API is always `_NO_ERROR`.

Example

```
DLL_INFORMATION   dllInformation;
GetDllInformation(
    &dllInformation
    );
```

ConnectDevice

```
DWORD   ConnectDevice(
            LPCSTR   lpszSerialNumber,
            LPSTR    lpszCurrentSerialNumber
            );
```

Function

Makes preparations for controlling the Card Reader/Writer(s). The key operations are as described below:

① To set up communication between the Host Computer and Card Reader/Writer
② To secure and initialize the resource for controlling the objective Card Reader/Writer

→ Refer to `DisconnectDevice`.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the Card Reader/Writer for which communication is set up. The serial number shall be a character string having a NULL at the end.

If only one Card Reader/Writer exists, you might as well specify it with a NULL instead of the serial number.
If a NULL is specified even though two or more Card Reader/Writers exist, it becomes impossible to identify which Card Reader/Writer is the target one so that operation may not work on the Card Reader/Writer you want but work on another one. In such a case, therefore it is prohibited to specify a NULL.

lpszCurrentSerialNumber

Once the API implementation succeeds, the serial number of the Card Reader/Writer, for which communication has been set, gets returned. The serial number to be returned at the time is expressed as a character string having a NULL at its end. Then, the serial number is saved in the address specified with `lpszCurrentSerialNumber`, and it is reported to the Application Program.
For the serial number storage, it is recommended to secure a memory area of 16 bytes or greater. Regarding the serial number and the size of the memory area required for saving the serial number, refer to the specifications of Card Reader/Writer Interface.

Even if communication is set with `lpszSerialNumber = NULL`, the serial number of the objective Card Reader/Writer can be noted by using `lpszCurrentSerialNumber`.
To execute any API such as `ExecuteCommand and CancelCommand`, which requires specifying a serial number, set the serial number obtained in this manner.

For execution of any API operation accompanied by communication transaction, it is needed to complete `ConnectDevice` in advance.

Return Value

_NO_ERROR

Normally completed

_CANNOT_CREATE_OBJECT_ERROR

Failed in creating the object

_CANNOT_OPEN_DRIVER_ERROR

Failed in opening the driver

This error may be caused in the cases described below:

① The specified Card Reader/Writer is already in operation.
② The specified Card Reader/Writer is not installed yet.
③ The communication driver cannot get opened due to any other reason.

_FAILED_TO_BEGIN_THREAD_ERROR

Failed in creating or invoking a thread.

_DEVICE_ALREADY_CONNECTED_ERROR

`ConnectDevice` is already completed.

Example

E.g. 1)

```
static char    szSerialNumber[ ] = “12345678”;
char    szSerialNumberDetected[ 16];
DWORD    dwResult = ConnectDevice(
                                    szSerialNumber,
                                    szSerialNumberDetected
                                    );

if( dwResult == _NO_ERROR){
    // Communications between the Host Computer and the Card Reader/Writer was established
    // successfully.    In this case the same character string as the one given by szSerialNumber is
    // saved in szSerialNumberDetected.
    ...
    ...
}
```

E.g. 2)

```
char    szSerialNumberDetected[ 16];
DWORD   dwResult = ConnectDevice(
                                NULL,
                                szSerialNumberDetected
                                );

if( dwResult == _NO_ERROR){
    // Communications between the Host Computer and the Card Reader/Writer was established
    // successfully.   In this case the target device's serial number is returned and then saved in
    // szSerialNumberDetected.
    ...
    ...
}
```

DisconnectDevice

```
DWORD   DisconnectDevice(
            LPCSTR   lpszSerialNumber
            );
```

Function

Quits controlling the Card Reader/Writer. The key operations are as described below:

① To quit communication

② To release the resource secured for controlling the objective Card Reader/Writer

→ Refer to `ConnectDevice`.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.

→ Refer to `ConnectDevice`.

Once this API is executed, it becomes impossible to control the objective Card Reader/Writer. When it becomes necessary to control the Card Reader/Writer, execute `ConnectDevice` again.

When it becomes unnecessary to control the Card Reader/Writer, for example, at the time of quitting the Application Program; be sure to execute `DisconnectDevice` as a quitting operation.

If Plug & Play occurs in the process and it requires you to execute `ConnectDevice` again, execute `DisconnectDevice` first and then execute `ConnectDevice`.

Return Value

_NO_ERROR

Normally completed

_DEVICE_NOT_CONNECTED_ERROR

This error may be caused in the cases described below:

① `ConnectDevice` operation is not yet completed for the specified Card Reader/Writer.

② `DisconnectDevice` operation is already executed for the specified Card Reader/Writer.

Example

```
char    szSerialNumber[ 16];

// Establishes communications between the Host Computer and the Card Reader/Writer
DWORD    dwResult = ConnectDevice(
                                NULL,
                                szSerialNumber
                                );

if( dwResult == _NO_ERROR)
{
    // ConnectDevice completed
    ...
    ...
    // Closes communications
    DisconnectDevice( szSerialNumber);
}
```

CancelCommand

```
DWORD   CancelCommand(
            LPCSTR      lpszSerialNumber
            );
```

Function

Cancels the command being in operation. The key operations are as described below:

① To send “DLE,EOT” control code and receive the response

② To cancel the condition of waiting for the response if there exists `ExecuteCommand` waiting for response to the command

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.

→ Refer to `ConnectDevice`.

Being executed, this API sends “DLE,EOT” control code to the Card Reader/Writer. Once having received “DLE,EOT” control code while executing a command, the Card Reader/Writer interrupts the command execution and returns a response. The API waits for receiving the response, and it returns the control to the Application Program when the response reception is confirmed. Regarding the “DLE,EOT” control code and operation of Card Reader/Writers at the time when the Card Reader/Writer receives the code, refer to the specifications of Card Reader/Writer Interface.

To execute this API, `ConnectDevice` must have already been completed.

→ Refer to `ConnectDevice`.

If there exists `ExecuteCommand` waiting for response to command execution, executing `CancelCommand` from another thread cancels the condition of waiting for response so that the `ExecuteCommand` quits the execution and returns the control to the Application Program.

→ Refer to `ExecuteCommand`.

Return Value

_NO_ERROR

Normally completed

_DEVICE_NOT_CONNECTED_ERROR

`ConnectDevice` operation is not yet completed for the specified Card Reader/Writer.

→ Refer to `ConnectDevice`.

_FAILED_TO_SEND_COMMAND_ERROR

“DLE,EOT” cannot be sent.

When an error has happened in the layer that operates under the DLL to serve further basic functions to the DLL, this error code is returned.

_FAILED_TO_RECEIVE_REPLY_ERROR

An error has happened in the operation of receiving the response for “DLE,EOT” control code. When an error has happened in the layer that operates under the DLL to serve further basic functions to the DLL, this error code is returned.

_REPLY_TIMEOUT

A time-out error has happened in the operation of receiving the response for “DLE,EOT” control code.
The time-out setup is made to be 5 seconds in this API operation.

Example

```
char    szSerialNumber[ 16];

// Establishes communications between the Host Computer and the Card Reader/Writer
ConnectDevice(
        NULL,
        szSerialNumber
        );
...
...
...
// Aborts a command execution having been invoked by calling ExecuteCommand
// in another thread
CancelCommand(
        szSerialNumber
        );
```

ExecuteCommand

```
DWORD   ExecuteCommand(
            LPCSTR              lpszSerialNumber,
            CONST COMMAND       Command,
            CONST DWORD         dwTimeout,
            LPREPLY             lpReply
            );
```

Function

Executes a command, and returns the execution result.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.
→ Refer to `ConnectDevice`.

Command

Provide the materials needed to compose a command message, i.e., a command code, a parameter code, and data.
→ Refer to `COMMAND`.

dwTimeout

Specify the time-out interval for the period from sending the command message until receiving the reply message to it. The interval is specified in millisecond.
If `dwTimeout` is `INFINITE`, the function's time-out interval never elapses.

lpReply

Specify the data save destination for the command execution results, and give a pointer to `REPLY` type data as the save destination.
→ Refer to `REPLY`.

To execute this API, `ConnectDevice` must have already been completed.
→ Refer to `ConnectDevice`.

Regarding the details of the command to be executed by calling this API, refer to the specifications of Card Reader/Writer Interface.

Return Value

_NO_ERROR

Normally completed

_DEVICE_NOT_CONNECTED_ERROR

`ConnectDevice` operation is not yet completed for the specified Card Reader/Writer.
→ Refer to `ConnectDevice`.

_CANCEL_COMMAND_SESSION_ERROR

Since `CancelCommand` is now being executed, it is impossible to execute this API.
→ Refer to `CancelCommand`.

_FAILED_TO_SEND_COMMAND_ERROR

The command cannot be sent.
When an error has happened in the layer that operates under the DLL to serve further basic functions to the DLL, this error code is returned.

_FAILED_TO_RECEIVE_REPLY_ERROR

An error has been caused in response receiving operation for the executed command.
When an error has happened in the layer that operates under the DLL to serve further basic functions to the DLL, this error code is returned.

_COMMAND_CANCELED

The command execution has been canceled.

_REPLY_TIMEOUT

Time-out has been caused in response receiving operation for the executed command.

Example

E.g. 1)

```
#define   _TIMEOUT      (20000)       // milliseconds

CHAR    szSerialNumber[ 16];

// Establishes communications between the Host Computer and the Card Reader/Writer
// and detects a target device's serial number
DWORD    dwResult = ConnectDevice(
                                NULL,
                                szSerialNumber
                                );
 ...
 ...
COMMAND    Command;                    // Command message to send
{
    Command.bCommandCode = 0x31;   // Status request command
    Command.bParameterCode = 0x30;   // Parameter code
    Command.Data.dwSize = 0;                  // Data size
}
REPLY    Reply;                              // Reply message to receive

// Executes Status request command, and then receives a reply for the command
dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );
```

```
if( dwResult == _NO_ERROR && Reply.replyType == PositiveReply)
{
    if( Reply.message.positiveReply.StatusCode.bSt1 == '0'
    && Reply.message.positiveReply.StatusCode.bSt0 == '2')
    {
        // status code="02"
        // Detected a card inside of Card Reader/Writer
        ...
        ...
    }
}
else
{
    // Unexpected situation occurred
    ...
    ...
}
```

E.g. 2)

```
#define   _SERIAL_NUMBER   "12345678"   // Target device's serial number
#define   _TIMEOUT         (20000)      // milliseconds

CHAR   szSerialNumber[ 16];

// Establishes communications between the Host Computer and the Card Reader/Writer
DWORD   dwResult = ConnectDevice(
                              _SERIAL_NUMBER,
                              szSerialNumber
                              );
 ...
 ...

COMMAND     Command;       // Command message to send
{
    Command.bCommandCode = 0x30;      // Initialize command
    Command.bParameterCode = 0x30;    // Parameter code
    BYTE   fm = 0x30;
    BYTE   Pd = 0x30;
    BYTE   Wv = 0x30;
    BYTE   Sh = 0x30;
    BYTE   Ds = 0x30;
    BYTE   Ty = 0x31;
    BYTE   Cp = 0x30;
    BYTE   bData[ ] = { 0x33, 0x32, 0x34, 0x30, fm, Pd, Wv, Sh, Ds, Ty, Cp, };
    Command.Data.dwSize = sizeof( bData) / sizeof( BYTE);    // Data size (bytes)
    Command.Data.lpbBody = bData;    // Start address of region where data is saved
}
REPLY   Reply;       // Reply message to receive
```

```
// Executes Initialize command, and then receives a reply for the command
dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );

if( dwResult == _NO_ERROR)
{
    // Initialize command successfully finished.
    ...
    ...
    if( Reply.replyType == PositiveReply)
    {
        // Received positive reply
        ...
        ...
    }
    else if( Reply.replyType == NegativeReply)
    {
        // Received negative reply
        ...
        ...
    }
    ...
    ...
}
else
{
    // Initialize command failed.
    ...
    ...
}
```

ExecuteCommand2

```
DWORD   ExecuteCommand2(
            LPCSTR          lpszSerialNumber,
            LPCOMMAND       lpCommand,
            CONST DWORD     dwTimeout,
            LPREPLY         lpReply
            );
```

Function

Executes a command, and returns the execution result.

`ExecuteCommand` and `ExecuteCommand2` are identical except for just one point, i.e., in case of `ExecuteCommand` the second parameter, `COMMAND`, is given as an instance while the parameter is specified in the format of a pointer when calling `ExecuteCommand2`.
→Refer to `ExecuteCommand`.

ICCardTransmit

```
DWORD   ICCardTransmit(
                LPCSTR          lpszSerialNumber,
                COSNT DWORD     dwDataSizeToSend,
                LPBYTE          lpbDataToSend,
                CONST DWORD     dwSizeOfDataToReceive,
                LPDWORD         lpdwSizeOfDataReceived,
                LPBYTE          lpbDataReceived,
                LPDWORD         lpdwAdditionalErrorCode,
                LPREPLY         lpReply
                );
```

Function

Carries out a series of data exchange between the Host Computer and IC card.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.
→ Refer to `ConnectDevice`.

dwDataSizeToSend

Specify the size of the data to be sent to the IC card, in bytes.

lpbDataToSend

Specify the top address of the memory area where the data to be sent to the IC card is saved. The data, which starts from the address specified with this argument and whose size is as specified with the argument `dwDataSizeToSend`, is sent to the IC card.

dwSizeOfDataToReceive

Specify the size of the data to be received from the IC card, in bytes.
The specified data size shall be equal to or less than the data receiving buffer size specified by `lpbDataReceived`.
Sometimes the data received actually may be less than the size specified by this argument.

lpdwSizeOfDataReceived

Specify a pointer to the variable to save the size in the case of receiving data from the IC card. The size of the data actually received, in bytes, is saved.

lpbDataReceived

Specify the address of the data save destination for saving the data received from the IC card.

lpdwAdditionalErrorCode

When some type of error occurs, the API returns an additional information.
Specify a pointer to the variable to save that.
→ See “Return Value”.

lpReply

Specify the data save destination for the command execution results, and give a pointer to REPLY type data as the save destination.
→ Refer to REPLY.

As API internal operation, `ExecuteCommand` gets called. `ExecuteCommand` is called repeatedly as many times as required. Then the result(s) of only the command executed by the last call is saved in the destination specified by this argument, and it is reported to the Application Program.
In case any halfway `ExecuteCommand` has got an error, it is determined that the subsequent operation cannot continue anymore and the API itself gets quitted due to the error. Under this situation, returning the result(s) of the command executed by the last call makes it possible to find out the cause the error, with which the API has quitted.
→ Refer to `ExecuteCommand`.

For the purpose of writing or reading a parcel of data into or out of the IC card, it can be done to call `ExecuteCommand` instead as many times as required for causing the same operation effect as this API does. However, since this API includes all the functions to execute chaining the IC card command as the internal processing, it is free from any troublesome factor that chaining the command by manual operation may result in.
Regarding the execution of chaining the IC card control command, refer to the specifications of Card Reader/Writer Interface.

It is forbidden to send a command by executing `ExecuteCommand` to the objective Card Reader/Writer from another thread in the process of this API. If this operation is done, the result is insecure.

To execute this API, `ConnectDevice` must have already been completed.
→ Refer to `ConnectDevice`.

Return Value

_NO_ERROR

Normally completed

_ICC_TRANSMIT_COMMAND_EXECUTION_FAILED_ERROR

`ExecuteCommand` called in the API internal operation failed.
In this case the variable pointed by `lpdwAdditionalErrorCode` receives the return value of `ExecuteCommand`.
→ Refer to `ExecuteCommand`.

_ICC_TRANSMIT_NEGATIVE_REPLY_RECEIVED_ERROR

A command executed in the API internal operation returned a negative response.

_ICC_TRANSMIT_FAILED_ALLOCATE_MEMORY_REGION_ERROR

Failed in securing the memory area required for executing this API.

_ICC_TRANSMIT_ABORT_REQUEST_RECEIVED_ERROR

Received an AB0RT request from the IC card.

_ICC_TRANSMIT_UNEXPECTED_ERROR

An unexpected error occurred.

Example

```
#define   _TIMEOUT      (20000)       // milliseconds

CHAR   szSerialNumber[ 16];
COMMAND Command;
REPLY  Reply;
DWORD   dwResult;

// Establishes communications between the Host Computer and the Card Reader/Writer
dwResult = ConnectDevice(
                    NULL,
                    szSerialNumber
                    );

if( dwResult != _NO_ERROR)
{
     // ConnectDevice failed
     goto   _EXIT;

}

...
...

// Activates IC card
{
     Command.bCommandCode = 0x49;   // IC card control
     Command.bParameterCode = 0x30;   // Activate
     Command.Data.dwSize = 0;

     dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );

     if( dwResult != _NO_ERROR || Reply.replyType != PositiveReply)
     {
          // Command sending failed or command execution failed
          goto   _EXIT1;
     }
}
```

```
// Exchanges data between the Host Computer and IC card
{
    BYTE    bDataToSend[ ] = { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, .... };
    WORD    dwDataSizeToSend = sizeof( bDataToSend) / sizeof( BYTE);
    BYTE    bData[1024];
    WORD    dwSizeOfDataReceived;
    DWORD    dwErrorCode;

    dwResult = ICCardTransmit(
                    szSerialNumber,
                    dwDataSizeToSend,
                    bDataToSend,
                    512,
                    &dwSizeOfDataReceived,
                    bData,
                    &dwErrorCode,
                    &Reply
                    );

    if( dwResult != _NO_ERROR)
    {
        // ICCardTransmit failed
        ...
        ...
    }
}

// Deactivates IC card
{
    Command.bCommandCode = 0x49;    // IC card control
    Command.bParameterCode = 0x31;    // Deactivate
    Command.Data.dwSize = 0;

    dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );

    if( dwResult != _NO_ERROR || Reply.replyType != PositiveReply)
    {
        // Command sending failed or command execution failed
        ...
        ...
    }
}

_EXIT1:
```

```
// Closes communications between the Host Computer and the Card Reader/Writer
dwResult = DisconnectDevice( szSerialNumber);

_EXIT:
```

SAMTransmit

```
DWORD   SAMTransmit(
            LPCSTR          lpszSerialNumber,
            COSNT DWORD     dwDataSizeToSend,
            LPBYTE          lpbDataToSend,
            CONST DWORD     dwSizeOfDataToReceive,
            LPDWORD         lpdwSizeOfDataReceived,
            LPBYTE          lpbDataReceived,
            LPDWORD         lpdwAdditionalErrorCode,
            LPREPLY         lpReply
            );
```

Function

Carries out a series of data exchange between the Host Computer and SAM.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.
→ Refer to `ConnectDevice`.

dwDataSizeToSend

Specify the size of the data to be sent to the SAM, in bytes.

lpbDataToSend

Specify the top address of the memory area where the data to be sent to the SAM is saved. The data, which starts from the address specified with this argument and whose size is as specified with the argument `dwDataSizeToSend`, is sent to the SAM.

dwSizeOfDataToReceive

Specify the size of the data to be received from the SAM, in bytes.
The specified data size shall be equal to or less than the data receiving buffer size specified by `lpbDataReceived`.
Sometimes the data received actually may be less than the size specified by this argument.

lpdwSizeOfDataReceived

Specify a pointer to the variable to save the size in the case of receiving data from the SAM. The size of the data actually received, in bytes, is saved.

lpbDataReceived

Specify the address of the data save destination for saving the data received from the SAM.

lpdwAdditionalErrorCode

When some type of error occurs, the API returns an additional information.
Specify a pointer to the variable to save that.
→ See “Return Value”.

lpReply

Specify the data save destination for the command execution results, and give a pointer to REPLY type data as the save destination.
→ Refer to REPLY.

As API internal operation, `ExecuteCommand` gets called. `ExecuteCommand` is called repeatedly as many times as required. Then the result(s) of only the command executed by the last call is saved in the destination specified by this argument, and it is reported to the Application Program.
In case any halfway `ExecuteCommand` has got an error, it is determined that the subsequent operation cannot continue anymore and the API itself gets quitted due to the error. Under this situation, returning the result(s) of the command executed by the last call makes it possible to find out the cause the error, with which the API has quitted.
→ Refer to `ExecuteCommand`.

For the purpose of writing or reading a parcel of data into or out of the SAM, it can be done to call `ExecuteCommand` instead as many times as required for causing the same operation effect as this API does. However, since this API includes all the functions to execute chaining the SAM command as the internal processing, it is free from any troublesome factor that chaining the command by manual operation may result in.
Regarding the execution of chaining the SAM control command, refer to the specifications of Card Reader/Writer Interface.

It is forbidden to send a command by executing `ExecuteCommand` to the objective Card Reader/Writer from another thread in the process of this API. If this operation is done, the result is insecure.

To execute this API, `ConnectDevice` must have already been completed.
→ Refer to `ConnectDevice`.

Return Value

_NO_ERROR

Normally completed

_SAM_TRANSMIT_COMMAND_EXECUTION_FAILED_ERROR

`ExecuteCommand` called in the API internal operation failed.
In this case the variable pointed by `lpdwAdditionalErrorCode` receives the return value of `ExecuteCommand`.
→ Refer to `ExecuteCommand`.

_SAM_TRANSMIT_NEGATIVE_REPLY_RECEIVED_ERROR

A command executed in the API internal operation returned a negative response.

_SAM_TRANSMIT_FAILED_ALLOCATE_MEMORY_REGION_ERROR

Failed in securing the memory area required for executing this API.

_SAM_TRANSMIT_ABORT_REQUEST_RECEIVED_ERROR

Received an AB0RT request from the IC card.

_SAM_TRANSMIT_UNEXPECTED_ERROR

An unexpected error occurred.

Example

```
#define   _SERIAL_NUMBER     "12345678"

#define   _TIMEOUT           (20000)       // milliseconds

CHAR   szSerialNumber[ 16];
COMMAND Command;
REPLY  Reply;
DWORD   dwResult;

// Establishes communications between the Host Computer and the Card Reader/Writer
dwResult = ConnectDevice(
                    _SERIAL_NUMBER,
                    szSerialNumber
                    );

if( dwResult != _NO_ERROR){
     // ConnectDevice failed
     goto   _EXIT;
}

...
...

// Selects SAM
{
     Command.bCommandCode = 0x49;      // SAM control
     Command.bParameterCode = 0x50;    // Select SAM
     BYTE   Sel = 0x30;                // SAM#1
     Command.Data.dwSize = 1;          // Data size
     Command.Data.lpbBody = &Sel;      // Data

     dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );
```

```
    if( dwResult != _NO_ERROR || Reply.replyType != PositiveReply)
    {
        // Command sending failed or command execution failed
        goto   _EXIT1;
    }
}

// Activates SAM
{
    Command.bCommandCode = 0x49;   // SAM control
    Command.bParameterCode = 0x40;   // Activate
    Command.Data.dwSize = 0;

    dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );

    if( dwResult != _NO_ERROR || Reply.replyType != PositiveReply)
    {
        // Command sending failed or command execution failed
        goto   _EXIT1;
    }
}


// Exchanges data between the Host Computer and SAM
{
    BYTE   bDataToSend[ ] = { 0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17, .... };
    DWORD   dwDataSizeToSend = sizeof( bDataToSend) / sizeof( BYTE);
    BYTE   bData[1024];
    DWORD   dwSizeOfDataReceived;
    DWORD   dwErrorCode;

    dwResult = SAMTransmit(
                    szSerialNumber,
                    dwDataSizeToSend,
                    bDataToSend,
                    512,
                    &dwSizeOfDataReceived,
                    bData,
                    &dwErrorCode,
                    &Reply
                    );
```

```
    if( dwResult != _NO_ERROR)
    {
        // SAMTransmit failed
        ...
        ...
    }
}

// Deactivates SAM
{
    Command.bCommandCode = 0x49;   // SAM control
    Command.bParameterCode = 0x41;   // Deactivate
    Command.Data.dwSize = 0;

    dwResult = ExecuteCommand(
                    szSerialNumber,
                    Command,
                    _TIMEOUT,
                    &Reply
                    );

    if( dwResult != _NO_ERROR || Reply.replyType != PositiveReply)
    {
        // Command sending failed or command execution failed
        ...
        ...
    }
}

_EXIT1:

// Closes communications between the Host Computer and the Card Reader/Writer
dwResult = DisconnectDevice( szSerialNumber);

_EXIT:
```

UpdateFirmware

```
DWORD   UpdateFirmware(
            LPCSTR              lpszSerialNumber,
            LPCSTR              lpszFilename,
            CONST BOOL          fCheckRevision,
            CALL_BACK_FUNCTION  fnFunction,
            LPDWORD             lpdwAdditionalErrorCode
            );
```

Function

Updates the firmware of the Card Reader/Writer.

Argument & Detailed Function

lpszSerialNumber

Specify the serial number of the objective Card Reader/Writer. The serial number shall be a character string having a NULL at the end.

If only one Card Reader/Writer exists, you might as well specify it with a NULL instead of the serial number.
If a NULL is specified even though two or more Card Reader/Writers exist, it becomes impossible to identify which Card Reader/Writer is the target one so that operation may not work on the Card Reader/Writer you want but work on another one. In such a case, therefore it is prohibited to specify a NULL.

→ Refer to `ConnectDevice`

lpszFilename

Specify the name of the file to be downloaded. The file name shall be a character string having a NULL at the end, and it can be accompanied by its path.

fCheckRevision

Determines whether or not to compare the revision of the file specified by *lpszFilename* with the one of firmware installed in the Card Reader/Writer before updating.
In case of TRUE, updating is done only when the revisions are not identical. If the value is FALSE, without comparing the revisions the file is installed replacing a current firmware.

fnFunction

Specify the callback function that is to be called for indicating the percentage of the total process. If you do not need any service like that, specify it with a NULL.

The framework of the callback function has to be the following:

```
VOID WINAPI Func(
    WPARAM    wParam,   // Indicates the percentage: 0,1,2,...,100
    LPARAM    lParam      // No function is assigned
)
{
    ...
    ...
}
```

The percentage is provided as a value of `wParam` varying from 0 to 100, and the callback function is called only when the value of the percentage increased.
There is no function assigned to `lParam`.

lpdwAdditionalErrorCode

When some type of error occurs, the API returns an additional information.
Specify a pointer to the variable to save that.
→ See “Return Value”.

When this API is executed, operation is carried out in due order as the following list describes:

Order of operation	Purpose	What is executed	Descriptions
①	To commence communication	`ConnectDevice`	
②	To distinguish Supervisor mode & User mode	`Initialize command`	If it is notified to be Supervisor mode, the operation jumps to step ⑥. If it is User mode, the operation switches to Supervisor mode by steps ③～⑤.
③	To switch to Supervisor mode	`Switch command`	By executing this command, Plug & Play operation occurs.
④	To terminate communication	`DisconnectDevice`	Since Plug & Play operation occurs, communication-commencing operation is carried out again.
⑤	To commence communication	`ConnectDevice`	
⑥	To initialize the Card Reader/Writer	`Initialize command`	
⑦	To transmit download data	`Download command`	Step ⑦ gets repeated to download all data.
⑧	To switch to User mode	`Switch command`	By executing this command, Plug & Play operation occurs.
⑨	To terminate communication	`DisconnectDevice`	Since Plug & Play operation occurs, communication-commencing operation is carried out again.
⑩	To commence communication	`ConnectDevice`	
⑪	To make sure that the firmware has been updated correctly	`Initialize command`	It is checked that a positive response gets returned to this command, to make sure that the “User Program Code Area” is operating.
⑫	To terminate communication	`DisconnectDevice`	

As `ConnectDevice` and `DisconnectDevice` are executed in the internal operation of this API, you do not need to execute `ConnectDevice` to establish communications between the Host Computer and the objective Card Reader/Writer before executing this API. If `ConnectDevice` has finished, execute `DisconnectDevice` first and then execute this API.

It is forbidden to access to or to control the objective Card Reader/Writer of this API in the process by executing any API such as ConnectDevice, DisconnectDevice, ExecuteCommand, and CancelCommand. If this operation is done, the result is insecure.

Return Value

_NO_ERROR

Normally completed

_UPDATE_FIRMWARE_CONNECT_DEVICE_FAILED_ERROR

`ConnectDevice` called in the API internal operation failed.
In this case the variable pointed by `lpdwAdditionalErrorCode` receives the return value of `ConnectDevice.`
→ Refer to `ConnectDevice`.

_UPDATE_FIRMWARE_DISCONNECT_DEVICE_FAILED_ERROR

`DisconnectDevice` called in the API internal operation failed.
In this case the variable pointed by `lpdwAdditionalErrorCode` receives the return value of `DisconnectDevice.`
→ Refer to `DisconnectDevice`.

_UPDATE_FIRMWARE_UNKNOWN_FILE_TYPE_ERROR

Unknown type of file was specified. This API does not know how to handle the file.

_UPDATE_FIRMWARE_CANNOT_OPEN_FILE_ERROR

The file cannot get opened.

_UPDATE_FIRMWARE_FAILED_TO_ALLOCATE_MEMORY_REGION_ERROR

Failed in securing the memory area for saving the download file.

_UPDATE_FIRMWARE_CANNOT_READ_FILE_ERROR

An error has happened at the time of reading the file.

_UPDATE_FIRMWARE_UNEXPECTED_FILE_CONTENTS_ERROR

The file cannot get interpreted.

_UPDATE_FIRMWARE_DEVICE_ALREADY_CONNECTED_ERROR

Communications between the Host Computer and the objective Card Reader/Writer has been established.

_UPDATE_FIRMWARE_COMMAND_EXECUTION_FAILED_ERROR

`ExecuteCommand` called in the API internal operation failed.
In this case the variable pointed by `lpdwAdditionalErrorCode` receives the return value of `ExecuteCommand.`
→ Refer to `ExecuteCommand`.

_UPDATE_FIRMWARE_NEGATIVE_REPLY_RECEIVED_ERROR

A command executed in the API internal operation returned a negative response.

_UPDATE_FIRMWARE_IDENTICAL_REVISION_ERROR

The revision number of the file specified by `lpszFilename` is identical with the one of the current firmware.

_UPDATE_FIRMWARE_UNEXPECTED_ERROR

An unexpected error occurred.

Example

E.g. 1)

```
static CHAR    szSerialNumber[ ] = “12345678”;
DWORD    dwErrorCode;

// Updates firmware
DWORD    dwResult = UpdateFirmware(
                                szSerialNumber,     // Objective Card Reader/Writer’s serial number
                                “1234-56A.DWL”,   // File to download into the Card Reader/Writer
                                TRUE,                 // Compares revision numbers
                                NULL,                 // No information about progress is required
                                &dwErrorCode       // Additional error information is returned if needed
                                );
```

E.g. 2)

```
BOOL    g_fInTheProcess = FALSE;

// Callback function
void WINAPI Func( WPARAM    wParam, LPARAM    lParam)
{
    if( wParam == 0)
    {
        // Firmware updating process has just begun
        g_fInTheProcess = TRUE;
    }
    else if( wParam == 100)
    {
        // Firmware updating process has ended
        g_fInTheProcess = FALSE;
    }
    ...
    ...
}
```

```
void    Sample( void)
{
      static CHAR    szFilename[ ] = “c:\\users\\systemfiles\\1234-56A.dwl”;
      DWORD    dwErrorCode;

      // Updates firmwate
      DWORD    dwResult = UpdateFirmware(
                                    NULL,          // Just one Card Reader/Writer is installed
                                    szFilename,
                                    FALSE,
                                    ( CALL_BACK_FUNCTION)Func,
                                    &dwErrorCode
                                    );
      ...
      ...
}
```

4.2.4 Log file

`CollectLogEx.dll` is the DLL to create a log file.
The features of `CollectLogEx.dll` and the log file created by the DLL are the followings:

Item	Features
[1] File format	Format: Text file Symbol of line breaks: CR,LF (linefeed-carriage return character pairs) Symbol of an end of file: Not used
[2] File name	$LogEx.txt The following registry enables you to modify the setting: Registry: `HKEY_LOCAL_MACHINE\SOFTWARE\SankyoCollectLogEx` Key: `LogFileName (REG_SZ)`
[3] Folder to create a log file	Current directory The following registry enables you to modify the setting: Registry: `HKEY_LOCAL_MACHINE\SOFTWARE\SankyoCollectLogEx` Key: `LogFileFolder (REG_SZ)`
[4] Maximum number of lines	10,000 The following registry enables you to alter the setting of the value: Registry: `HKEY_LOCAL_MACHINE\SOFTWARE\SankyoCollectLogEx` Key: `MaxNumOfLines (REG_DWORD)` Range of the value: from 1 to 100,000 (If the value set is out of range, 10,000 is applied.) Note: When the line number exceeds the value specified by “maximum number of lines”, it returns to the first line and keeps recording logs putting new lines over existing ones.
[5] Number of characters per line	100 The following registry enables you to alter the setting of the value: Registry: `HKEY_LOCAL_MACHINE\SOFTWARE\SankyoCollectLogEx` Key: `MaxNumOfCharactersPerLine (REG_DWORD)` Range of the value: from 100 to 256 (If the value set is out of range, 100 is applied.)
[6] Maximum size of the file	The following calculation gives the size in bytes: (maximum number of lines)x{(number of characters per line)+2}
[7] Capability of disabling the DLL to create a log file	If there is no `CollectLogEx.dll` installed, no log file is to be created. Note: Even without installation of `CollectLogEx.dll`, the DLL can still work. However, under such condition, no log file is created. When log file creating operation itself seems to cause a security problem because any created log file may drain away by mistake to disclose the contents of transaction, it might be better not to install `CollectLogEx.dll` or to uninstall it if it is already installed.
[8] Identification of the latest recorded line	The marker line, which is filled with ‘~’, is made to show the newest line in a log file, and it facilitates searching the position of the line.

The end of this document.