APPLICATION NOTE

LIBRARY REFERENCE
FOR TDA8030

Release 1.0

ISO 7816 and EMV 3.1.1 compliant

AN/01012

C51 KEIL Compiler

Philips
Semiconductors

PHILIPS

ABSTRACT

The single chip USB smart card device TDA8030 integrates in one chip both an smart card and an Usb interfaces. Two independent libraries driving these two interfaces are proposed to ease the development of Usb smart card reader applications.

This document describes the libraries references of the smart card and the Usb interfaces.

APPLICATION NOTE

LIBRARY REFERENCE FOR TDA8030

C51 KEIL Compiler

Author(s):
Jean-Luc LUONG
System & Applications
Business Unit Identification – Business Line RIC Caen
France

Keywords
TDA8030
ISO 7816-3
EMV 3.1.1
Library
USB

Date: 26 April 2001

TABLE OF CONTENTS

1 General considerations

1.1 Assignment of micro-controller pins

In order to control and to communicate with these two interfaces, a number of micro-controller pins are used. The table below indicates the assignment of the micro-controller pins.

SMART CARD INTERFACE (SCI)			
UP Pins	**Type**	**Linked to**	**Function description**
P00-P07	I/O	SCI Low Address/Data Bus	Low address/Data bus
P20-P27	I/O	SCI high address bus	High address bus
P27	O	SCI Chip SelectS	SCI Selection
P32	I	SCI Interrupt	SCI Interrupt
P36	O	SCI WR strobe	WR strobe
P37	O	SCI RD strobe	RD strobe
USB INTERFACE (UI)			
P00-P07	I/O	UI Low Address/Data Bus	Low address/Data bus
P20-P27	I/O	UI high address bus	High address bus
P10	O	Mask_Interrupt_Usb	Used in suspend mode to mask usb interrupt
P11	O	Usb_Softconnect	Links the 1,5K pull-up resistor to the cable voltage
P12	I	Usb_Mp_Ready	Indicate the ready/not-ready states of the UI
P13	I	Usb_Clk_Enable_N	Indicates the exit from suspend mode
P14	O	Usb_Reset_N	Reset UI
P25	O	UI Command/data selection	Command/Data Selection
P26	O	UI Chip Select	UI Selection
P33	I	Usb_Intr_N	UI Interrupt
P34	I	Usb_Suspend	Indicates suspend/ not suspend states of the UI
P35	O	Usb_WakeUp_N	????
P36	O	UI WR strobe	WR strobe
P37	O	UI RD strobe	RD strobe
AVAILABLE PINS FOR CUSTOMER APPLICATIONS			
P15	I/O		
P16	I/O		
P17	I/O		
P30	I/O		
P31	I/O		

1.2 ISO 7816 mode and EMV 3.11 mode

The choice between these two modes is made during the activation of the card (see power up function). At each activation it is possible to choose the mode.

For the EMV 3.11 mode, refer to the document:

- EMV 96 Integrated Circuit Card Specification For Payment Systems Version 3.1.1 May 31, 1998

The asynchronous library has been validated according to the Europay document:

- ICC Terminal Type Approval: Test Bench Description Executable Tests February 11th, 2000

1.3 Compilation

The libraries are compiled in **small model** (refer to the compiler manual, and for the 83/87C51RB+, this micro-controller has **512 bytes of additional auxiliary RAM** addressable as an external memory.

1.4 Auxiliary RAM

The library uses a **buffer in auxiliary RAM** to handle the communications between the host and the card reader and between the smart card and the card reader. The buffer length is limited by the auxiliary RAM size. The exchange block length between the card reader and the card is limited at 266 bytes. The start address of the buffer is 0 in auxiliary RAM. It is possible to configure the size of the buffer.

1.5 Interrupts

The interrupt **INT0** is used by the smart card library. Its interrupt service routine uses the **register bank 1. INT0** has a low priority level**;** all functions use the **register bank 0**. The register bank 2 and the register bank 3 are available for the software application.

The interrupt **INT1** is only used by the USB library to wake up from suspend mode. This interrupt, when it happens, uses register bank 0 and does not alter any register in bank 0. Otherwise, the interrupt **INT1** is not activated and consequently not used.

1.6 Timer

The TDA8030's timers are used for all the timing issues whatever the protocol (T=0 or T=1) is.Work Waiting Time, Extra Guard Time, Character Waiting Time, Block Waiting Time, Block Guard Time, Block Waiting Time extension are fully supported whatever the baud rate is on the I/O line.

The timers are also used for counting the 40.000 cycles (or more) during the ATR phase and also for counting the maximum duration of the ATR (19200 etu) in case of EMV standard.

1.7 System initialisation

Before calling any function of the libraries, it is **mandatory to call the function init_system.** It must be the **first instructions** of the application software. Refer to chapter 2.1 for more details about these functions.

1.8 Library function calls

Any function must be called from **the register bank 0**.
The maximum stack level used by the library is 8. The minimum depth of the stack is 16 bytes (1 level uses 2 bytes of stack.

1.9 Data exchange with the host

The libraries do not contain the software communication with the host. The software communication depends on the protocol used by the application.

1.10 Linker

In the command line of the link one has to specify both the internal RAM size (256) and the start address of the indirect memory area (80h).
Command line example to generate the object code for the emulation:

MyAppli.obj, edphl.obj, asynch.lib, usb.lib
RAMSIZE(256)
IDATA(80H)

If the start address of indirect memory area is not specify, the indirect data can be located in direct memory area. It is mandatory to specify the RAM size.

1.11 Variables location

All variables used by the libraries are located in the 256 bytes of the internal RAM.

1.12 Code and ram sizes of the libraries

The table blow gives some indications about the ROMand the RAM sizes of the libraries:

Name	Rom size	Data size	Idata size	Bit size
Async.lib	9 Kbytes	33	29	35
Usb.lib	1,6 Kbytes	3	20	0
Edphl.c	409 bytes	3	0	6

The figures given above is get with the level 9 of optimisation of the Keil C compiler.

2 Library of smart card interface

2.1 DESCRIPTION OF THE LIBRARY FUNCTIONS

All the library functions are using bank 0 and should be used in the same bank.

LIBRARY ASYNC.LIB			
Function Name	**Syntax**	**Description**	**Stack Level**
init_system	void init_system (unsigned int **size**)	The init_system function defines the **size** of the **data exchange buffer in auxiliary Ram.** The data exchange buffer is used for data communication between the smart card and the card reader. Its length is limited to the auxiliary RAM size. This function initializes the variables system of the libraries. The maximum length of the data exchange buffer is 256 bytes **It is mandatory to call this function at the beginning of the main application** The data exchange buffer is used by the functions below: • XmitAPDU function • power_up function • negotiate function • Send_Sblock_IFSD function	3
XmitAPDU	Unsigned int XmitAPDU (unsigned int **offset,** unsigned int **length**)	The XmitAPDU function is used for the transportation of the APDU in T=0 protocol or T=1 protocol according to **ISO 7816-3 and 7816-4**. **offset** contains the offset address of the data exchange buffer where the first byte APDU is stored (data exchange buffer address + offset). **Length** is APDU length **(256 bytes max)** The characters received from the card are stored at the data exchange buffer address + offset, the command is overwritten. **Return value**: Number of bytes returned by the card, offset -> received bytes from the card in the data exchange buffer. The command is overwritten. This function returns zero when an error occured, in this case the error type can be obtained by the read_error() function (see Error list chapter 4)	8
power_up	unsigned char power_up(unsigned int **ptr**, unsigned char **voltage**)	The power_up function: • switches the libraries in 7816 mode or in EMV 3.11 mode depending of the content of the Data Exchange Buffer + ptr. If '0' the 7816 mode will be selected, if other value the EMV 3.1.1 mode will be selected. This memory location will be overwritten during the ATR, • asserts VCC (**3 or 5 volts** depending of the **voltage**) on the card. Only 2 values voltage are possible 3 or 5 • sets Fcard = F_{Xtal} /4 • sets ETU = Fcard/372 during the Answer To Reset • sets Fµc = F_{Xtal} • sets UART prescaler = /31 • activate the card, • stores the **A**nswer **T**o **R**eset into the Data Exchange Buffer in the auxiliary RAM, **ptr** contains the offset address of the Data Exchange Buffer, • decodes the protocol T=0 or T=1, • memorizes WI, CWI, BWI which are used to control the communications with the card, • Initializes the Guard Time Register GTR, • In specific mode (TA2 is present) automatically applies the protocol indicated in TA2 and uses Fi/Di parameters given in ATR. • **Return value:** Number of bytes of the Answer To Reset (< 32 bytes). • 0h = error, in this case the error type is sent back by the read_error() function, see Error list chapter 4. • FFh = In EMV mode, a SIFS_REQUEST is automatically perform after a right A.T.R. and power_up() function will return FFh in case of error.	6

		Example: init_system (256);The auxiliary RAM = 256 bytes length status = power_up(0, 3); The ATR will be stored at the first of the Data Exchange Buffer. The Vcc card is 3 Volts.	
power_down	void power_down(void)	The power_down function deactivates the card.	2
negotiate	unsigned char negotiate (unsigned int **ptr**, unsigned char **protocol**, unsigned char **FiDi**)	The negotiate function executes the Protocol Parameters Selection **PPS**. **ptr** contains the offset address of the Data Exchange buffer, the negotiate function uses 6 bytes from this address. **protocol** is used to select the protocol and **Fi/Di** is used to select the new Fi/Di parameters The coding of **protocol** is the following: • 00H for T=0 protocol • 01H for T=1 protocol The coding of **FiDi** is the coding of the TA1 parameter (see ISO7816-3). If the card is in negotiable mode (TA2 is not present) and answers in its ATR a Fi/Di different from the default value (Fi/Di = 372) or proposes 2 different protocols (T=0 and T=1), a PPS command can be given by using the negotiate command. This negotiate command automatically generates a PPS command by using the the selected Fi/Di parameters and the selected protocol**.** **Return value:** 1 = Successful 0 = Error, in this case the error type is sent back by the read_error() function, see Error chapter 4.	6
send_Sblock_IFSD	unsigned char send_Sblock_IFSD (unsigned char IFSD)	This function performs an IFSD negotiation with the card. An S(IFS)request is sent to the card with the value of the **IFSD** parameter given as an input. **Return value:** 1 successful 0 IFSD value not supported. (read error function)	6
set_clock_card	bit set_clock_card (unsigned char **divider**)	The set_clock_card function selects the clock to be sent to the card: • 0 RFU • 1 F_{Xtal} /2 • 2 F_{Xtal} /4 • 3 F_{Xtal} /8 • 4 Fint/2(~1.25Mhz) **Return value:** 1 = Successful 0 = Divider not supported or not compatible with Fi. The error type is sent back by the read_error() function, see Error list in chapter 4.	2
set_baud_rate	void set_baud_rate (unsigned char **FiDi,** unsigned char **CKU**)	The set_baud_rate function programs the ETU of the ISO uart according the **FiDi** value (see ISO 7813-3) and the **CKU** parameter value. If **CKU** is set to 0, the baud rate is the one defined by **FiDi**; if **CKU** is set to one, the baud rate on I/O line is the double of the baud rate defined by **FiDi** (the ETU value is divided by 2). **CKU** shall not be set if $F_{Clk} = F_{Xtal.}$ **Return value:** 0 = FiDi not accepted. The error type is sent back by the read_error() function, see Error list chapter 4. 1 = FiDi accepted	2
clock_stop	void clock_stop (unsigned char clock_level)	This function stops the card clock card at the required level: • 0 Fint/2 • 1 Clock stopped in low state • 2 Clock stopped in high state.	2
idle_mode	void idle_mode (unsigned char clock_level)	This function set the card clock at the required state and then puts the controller in idle mode: • 0 Fint/2 • 1 Clock stopped in low state • 2 Clock stopped in high state.	3
read_alarm	unsigned char read_alarm (void)	This function returns 0 when there is no alarm. If the returned value is 1, the alarm problem comes from card 1; the alarm type can be obtained by calling read_error()function. **It's mandatory to use this function to check if an alarm has occured**. **Example**: Alarm is activated when : the card is inserted or removed, and when the TDA8030 has been deactivated due to a	1

		hardware problem (overcurrent on VCC overheating, voltage drop on VDD).	
read_error	unsigned char read_error(void)	This function returns the error type when the functions listed below return an error: • Power_up • XmitAPDU • Negotiate • Send_Sblock_IFSD • Set_clock_card This function must be called if the read_alarm() function return 1 **Return value:** Error Type(defined for each function, see Error list in chapter 4)	1
check_pres_card	bit check_pres_card(void)	The check_pres_card function checks if the card is inserted in the card reader. **Return value:** 1 = The card is present 0 = No card	2
read_alarm_card_moved	unsigned char read_alarm_card_moved (void)	This command returns a byte which gives the information on the moving card. The definition of bits in the byte is given below. The application has to test if the card is inserted or removed after a debouncing delay by means of read_register() function. **Return value:** 1 = Card movement 0 = No card movement **It is mandatory to call this function when read_alarm() function is different of zero and when the read_error() function returns zero because in this case the alarm must be due to card movement.**	1
GetCardParam	unsigned int GetCardParam (unsigned int ptr)	This function returns 3 current parameters of the actived card : • FiDi (by example 11h for Fi= 372 and Di = 1) • Clock Card (01h for Fxtal, 02h for Fxtal/2, 04h for Fxtal/4, 08h for Fxtal/8). • Protocol (00h for T=0, 01h for T=1). If the card is active, it returns 3 otherwise 0 and an error message is generated in this case.	
read_card_active	bit read_card_active (unsigned char **slot**)	This function returns 1 if the card selected by **slot** is active. **Return value:** • 1 card active • 0 card inactive. The error type is sent back by the read_error() function, see Error list chapter 4.	5
SetNAD	void SetNAD (unsigned char NewNad)	This command is used to specify a SAD (source address) and a DAD (destination address) for T=1 protocol as defined in ISO7816-3. The default value is 00 and will be kept until the send NAD command has been notified to the TDA8029. Any NAD submission where SAD and DAD are identical (except 00) will be rejected. If bits b4 or b8 of the NAD required are set to 1 (VPP programming) the NAD will be rejected. The NAD shall be initialised before any information exchange with the card using T=1 protocol, otherwise and error message will be generated.	
LOW LEVEL FUNCTIONS			
write_register	void write_register(unsigned char **address**, unsigned char **data**)	This function writes a **data** into any writable register of the TDA8030. The register is selected by the **address** parameter. **Address** of the writable registers: reserved words, see tdalib.h	1
read_register	unsigned char read_register(unsigned char **address**)	This function allow to read all readable registers of the TDA8030. The register is selected by the **address** parameter. **Address** of the readable registers: reserved words, see tdalib.h **Return value:** read value.	1
write_bit	void write_bit (unsigned char reg, unsigned char mapping, bit value)	This function writes **bit_value** in the register located at **reg** address, following the bitmap given by **mapping**.	1
read_bit	bit read_bit (unsigned char reg, unsigned char mapping)	This function gives the value of the bit in the register located at **reg** address, following the bitmap given by **mapping**. Mapping must select only one bit at the same time. **Return value:** value of the bit given by mapping	3

2.2 ERROR LIST

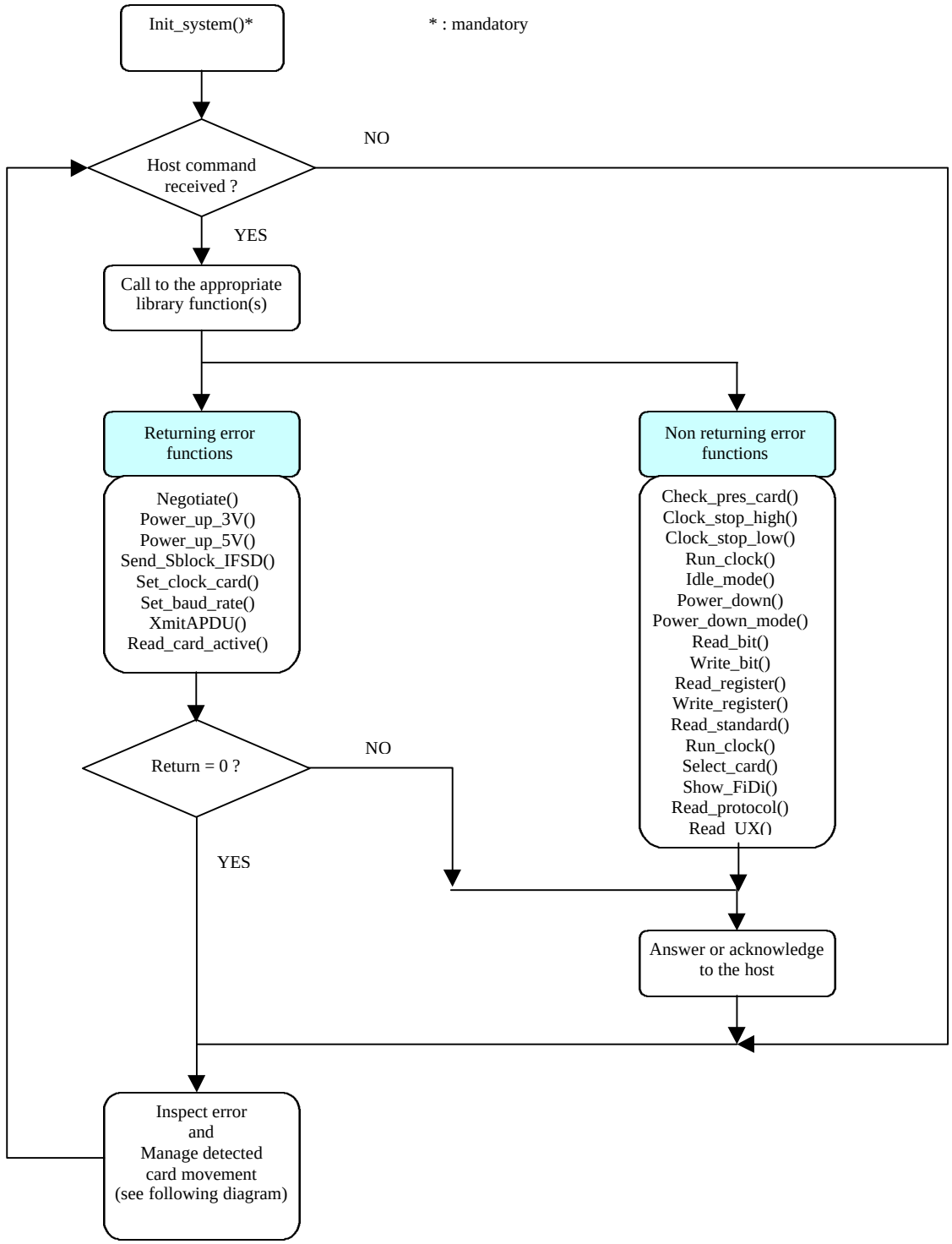
The error list gives the status code identification and a brief signification of the status error code. The error code is returned by the read_error() function. When this function returns 0 there is no error.

Error code	Meaning of error code
08h	Buffer length too short.
0Ah	3 errors in T=1.
20h	Wrong APDU.
21h	Too short APDU.
22h	Card mute during T=1 exchange.
24h	Bad NAD in a prologue field in T= 1
25h	Bad LRC in T = 1
26h	Card resynchronised in T = 1
27h	Chain aborted.
28h	Bad PCB in a prologue field in T = 1
29h	Card data overflow (512 bytes max)
2Ah	NAD not initialised
30h	Non negotiable mode
31h	Protocol different of T= 0 and T=1.
32h	Negotiation of protocol T=1 failed.
33h	PPS answer different of PPS request.
34h	Error on PCK.
35h	Parameter error
36h	Buffer overflow.
37h	BGT not reached.
38h	TB3 not present.
39h	PPS not accepted (no answer from card).
3Bh	Early answer of the card during activation
40h	Card deactivated
55h	Unknown command
80h	Card mute after power on.
81h	Time out.
82h	3 parity errors in TS.
83h	3 parity errors in reception
84h	3 parity errors in transmission.
85h	ATR too long.
86h	Bad FiDi
87h	Card clock not accepted.
88h	ATR duration longer than 19200 ETUs
89h	CWI not supported.
8Ah	BWI not supported.
8Bh	WI not supported.
8Ch	TC3 not supported.
8Dh	Parity errors during ATR.
8Eh	Error on SW1
90h	3 consecutive parity errors in T=1
91h	SW1 different of 6X and 9X
92h	Specific mode TA2 (b5=1)
93h	TB1 absent during cold reset
94h	TB1 different from 0 during cold reset

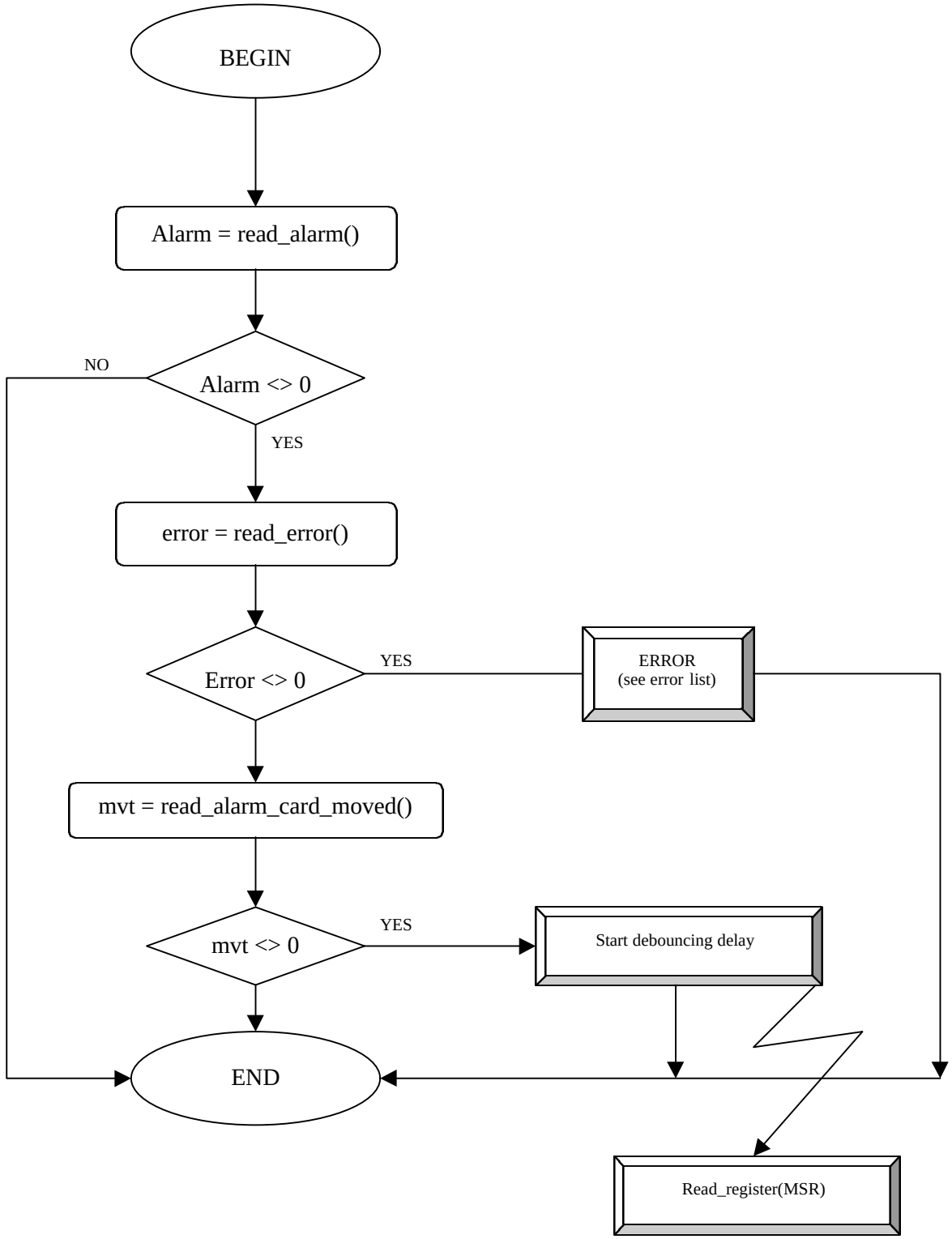
95h	IFSC less than 10h or IFSC equal to FFh
96h	Wrong Tdi
97h	TB2 present in ATR
98h	TC1 not compatible with CWT
99h	IFSD not accepted
A0h	Procedure byte error
B0h	Protected byte.
B1h	Pin error.
B2h	Write error.
B3h	Data length too long.
B4h	Error counter too long.
B5h	Pin code not verified.
B6h	Protected bit already set.
B7h	Verify pin error.
C0h	Card absent.
C1h	Io line locked
C2h	Protocol different T=0 and T=1.
C3h	Checksum error.
C4h	TS neither 3B or 3F.
C5h	Bad FiDi.
C6h	ATR not supported.
C7h	VPP not supported.
C8h	VCC error.
C9h	Card removed.
CAh	Class error without deactivation.
CBh	Class error with deactivation.
D0h	Bad secret code.
D1h	Card locked.
D2h	Write error.
D3h	Max erase time reached.
E0h	Card error
E1h	Bad clock card
E2h	Uart overflow
E3h	Supply voltage dropoff
E4h	Temperature alarm
E5h	Card 1 deactivated
E6h	Card 2 deactivated
E9h	Framing error
F0h	Serial LRC error
F1h	Command not executed (Collision)
FFh	Serial time-out

2.3 PRINCIPE OF UTILISATION OF THE LIBRARY

The two following diagram show the logical use of the library function.



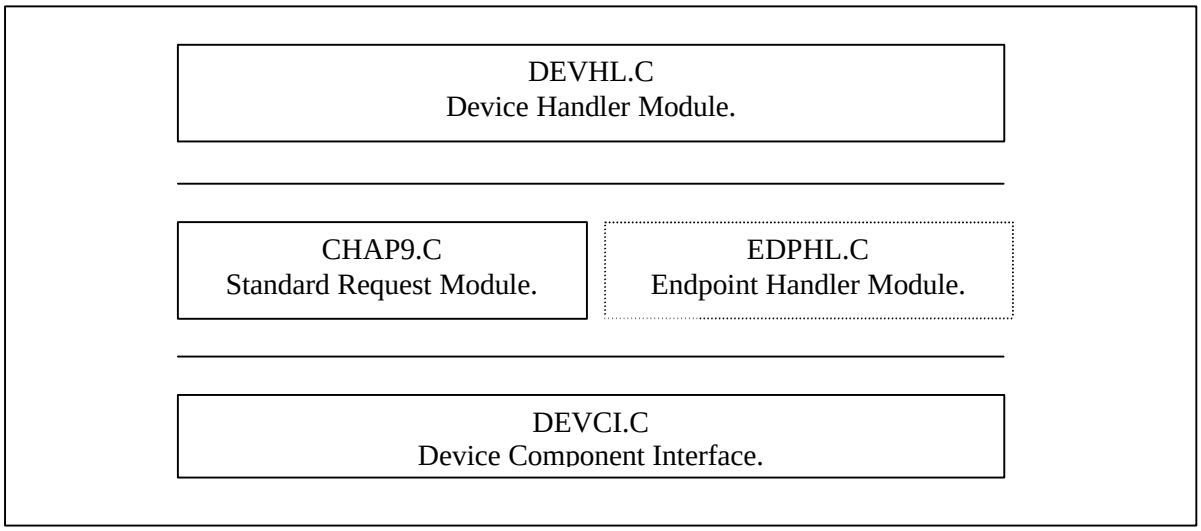
Alarm Monitoring:



3 Library of usb interface: Usb.Lib

3.1 STRUCTURE OF THE LIBRARY

The library of the USB interface consists of 3 modules encapsulated in a library package called USB.LIB and a custom module EDPHL.C. They are organised as follows:



3.1.1 Module Device Handler -- Devhl.c

This part of the code handles interrupts generated by the USB interface. It retrieves data from internal FIFO to CPU memory, and set up proper event flags to inform for processing.
The main loop checks the event flags and passes to appropriate subroutine for further processing

3.1.2 Module Chapter 9 -- Chap9.c

This module handles standard USB device requests defined in the chapter 9 of the USB specifications.

3.1.3 Custom Module Endpoint Handler – Edphl.c

This module contains the handlers of the bulk out, bulk in and interrupt in endpoints. **As the handling of these endpoints are application dependent, this module is not part of the USB library. The customer can consequently modify the endpoints handlers if needed.**

3.1.4 Module Device Component Interface – Devci.c

To further simplify programming with the USB interface, the firmware defines a set of command interfaces, which encapsulate all the functions used to access it.
The following functions are defined as USB command interface to simplify device programming. They are simple implementations of the USB command set, which are defined in the data sheet.

3.2 FLOWCHART OF MAIN USB HANDLER ROUTINES

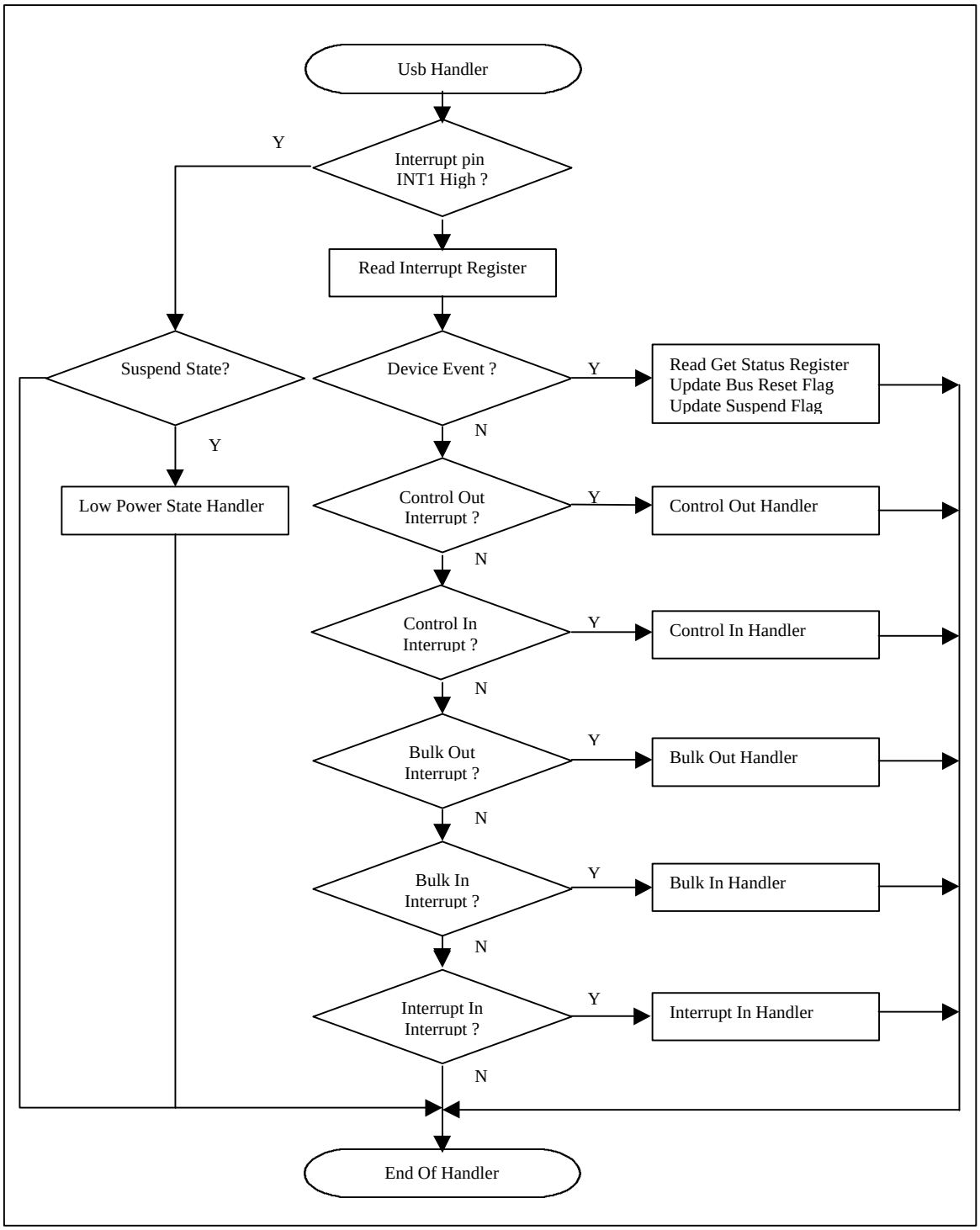
3.2.1 Usb event handler

UsbHandler is the service routine which handles all the USB events reported by the USB interface. The flow of the function *UsbHandler* is straightforward and is shown on the next page.

Everytime an USB event occurs – USB bus reset, suspend change or messages sent by the host – it is detected by the USB interface which puts low the interrupt pin INT1.

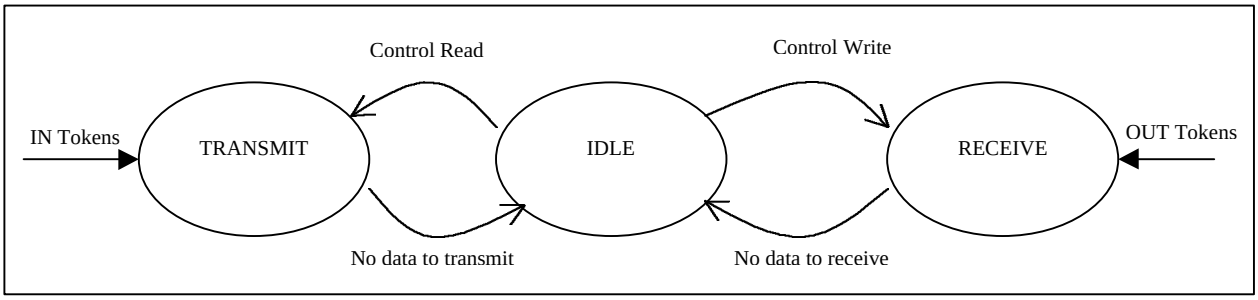
At the entrance of *UsbHandler*, the firmware uses the function *ReadInterruptRegister* to determine the origin of the interrupt source and then to dispatch it to the appropriate subroutines for processing.

When no USB activities are observed, the signal SUSPEND, normally in low state, goes to high state. The TDA8030 can be put in low power state by the function *LowPowerStateHandler*.


Usb Handler
Interrupt pin INT1 High ?
Y
Read Interrupt Register
Suspend State?
Device Event ?
Y
Read Get Status Register
Update Bus Reset Flag
Update Suspend Flag
N
Y
Low Power State Handler
Control Out Interrupt ?
Y
Control Out Handler
N
Control In Interrupt ?
Y
Control In Handler
N
Bulk Out Interrupt ?
Y
Bulk Out Handler
N
Bulk In Interrupt ?
Y
Bulk In Handler
N
Interrupt In Interrupt ?
Y
Interrupt In Handler
N
End Of Handler

3.2.2 Control endpoint handlers

Control transfer always begins with the SETUP stage and then followed later by an optional DATA stage. It then ends with the STATUS stage. The diagram below shows the various states of transitions on the control endpoints. The firmware uses these 3 states to handle control transfers correctly.



When the SETUP packet is received by the USB interface, it will generate an interrupt to the MCU. The micro-controller will need to service this interrupt by reading the interrupt register to determine whether the packet is send to Control Endpoint or the Generic Endpoint. If the packet is for the Control Endpoint, MCU will need to further determine whether the data is a SETUP packet through reading the Read Last Transaction Status Register. For the Get_Descriptor device request, the first packet will have to be the SETUP packet.

MCU will need to extract the content of the SETUP packet through Select Control Out Endpoint to determine whether this endpoint is full or empty. If the control endpoint is full, MCU will then read the content from the buffer and save the content to its memory. After that, it will validate the host device's request from the memory. If it is a valid request, MCU must send the ACKNOWLEDGE SETUP command to the Control Out endpoint to re-enable the next SETUP phase.

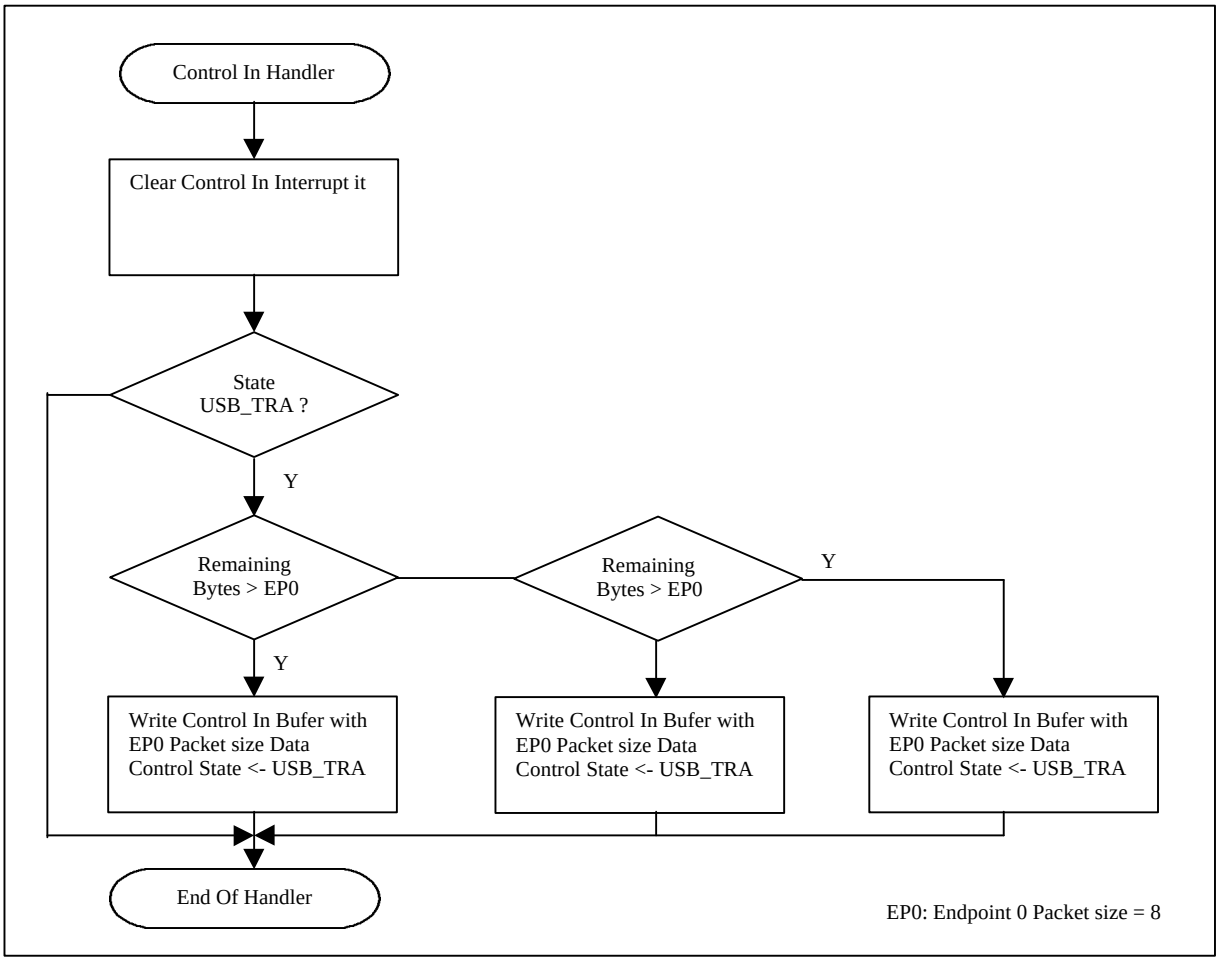
Next, MCU will need to verify whether the control transfer is a Control Read/Write. This can be achieved through reading the 8th bit of the bmRequestType from the SETUP packet. In our case, the control transfer is a Control Read type, that is, the device will need to send data packet back to the host in the coming data phase.

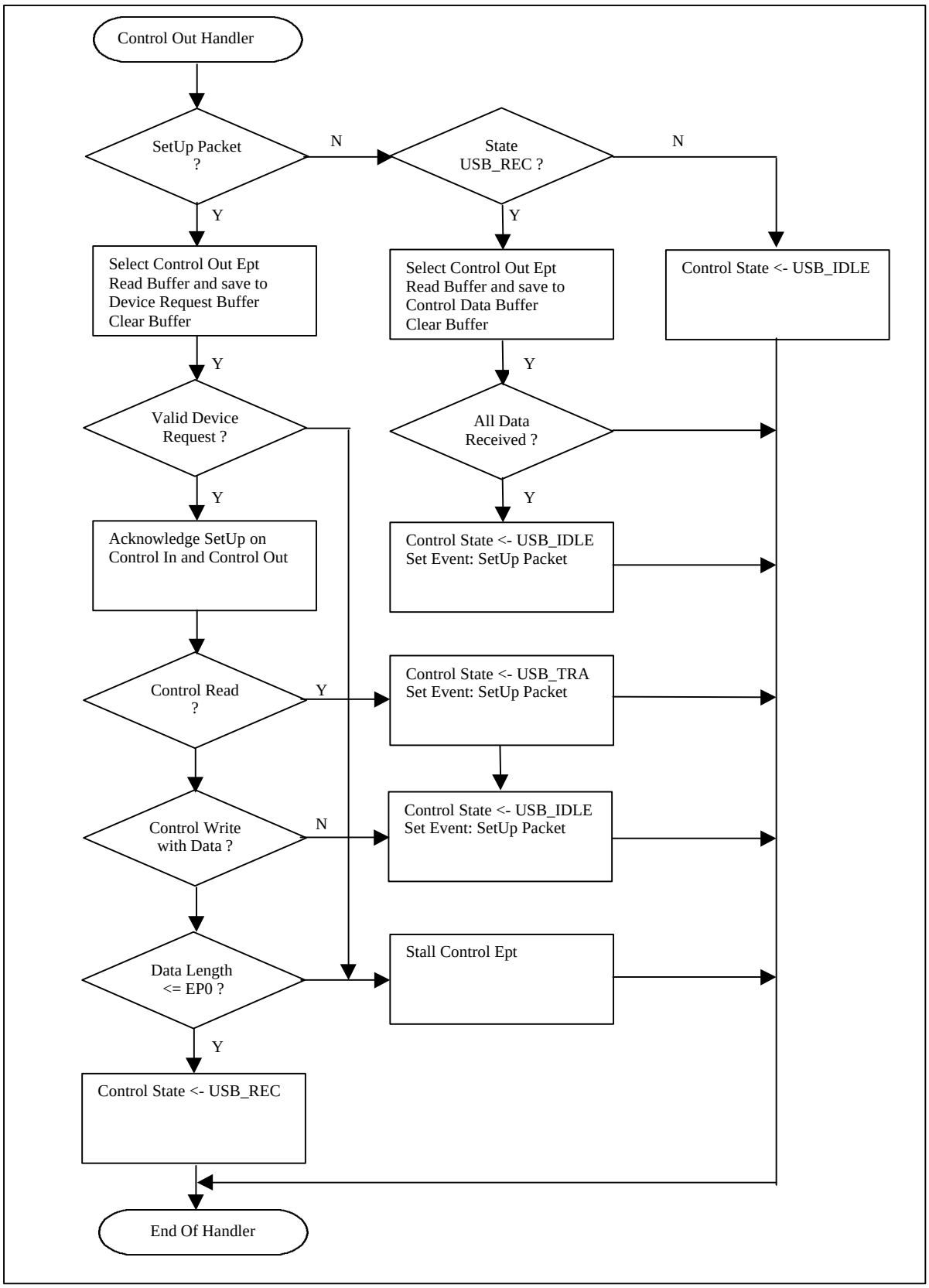
MCU will need to set a flag to indicate that the USB device is now in the transmit mode, that is, ready to send its data upon the host's request.

After the Setup stage is finished, the host will execute the data phase. The USB interface will expect to receive the Control_In packet. The process is shown in the next flowchart, "Control_In Handler". Again, MCU will first need to clear the Control_In interrupt Bit on the device by reading its Read Last Transaction Register. Then, MCU will proceed to send the data packet after verifying that the device is in the appropriate mode, that is, the Transmit mode.

As the control endpoint has only 16 bytes FIFO, MCU will have to control the amount of data during the transmitting phase if the requested length is more than 16 bytes. As indicated on the flowchart, MCU will have to check its current and remaining size of the data to be sent to the host. If the remaining bytes are greater than16 bytes, MCU will send the first 16 bytes and then subtract the reference length (requested length) by 16.
When the next Control_In token comes, MCU will determine whether the remaining bytes is zero. If there is no more data to send, MCU will need to send an empty packet to inform the host that there will be no more data to be sent over.
If the SETUP packet is Set_Descriptor request, then the control transfer in the SETUP packet will

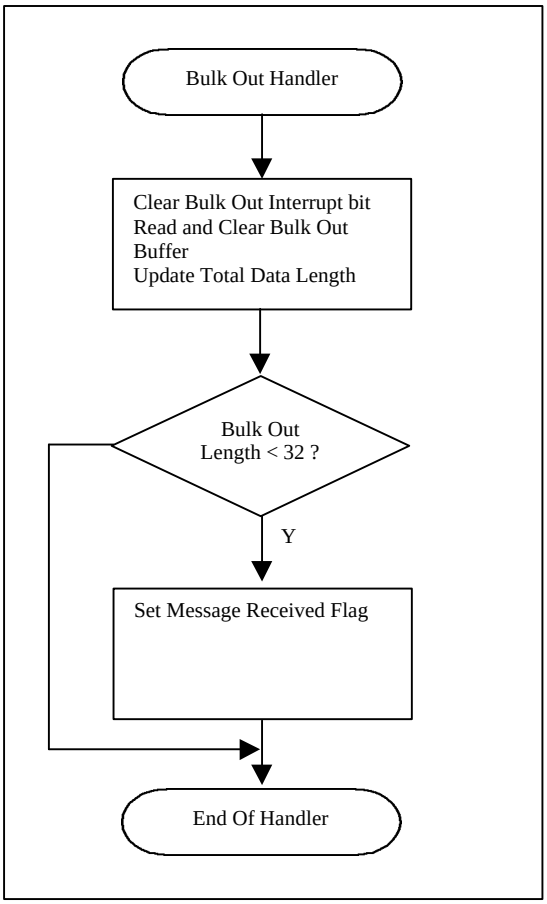
indicate that it is a Control Write type. After executing the procedure for the Set_Descriptor request, MCU will wait for the data phase. The host will send a Control Out token and MCU will have to extract the data from the device buffer. The flow will now be on the right side of the Control Out Handler flow sequence. MCU will have to first verify whether the device is in the USB_Receive mode. Then, MCU will have to verify whether the buffer is full by checking the Select Control Out Endpoint and read the data out from the buffer.




Control Out Handler
SetUp Packet ?
N
State USB_REC ?
N
Y
Y
Select Control Out Ept Read Buffer and save to Device Request Buffer Clear Buffer
Select Control Out Ept Read Buffer and save to Control Data Buffer Clear Buffer
Control State <- USB_IDLE
Y
Y
Valid Device Request ?
All Data Received ?
Y
Y
Acknowledge SetUp on Control In and Control Out
Control State <- USB_IDLE Set Event: SetUp Packet
Control Read ?
Y
Control State <- USB_TRA Set Event: SetUp Packet
Control Write with Data ?
N
Control State <- USB_IDLE Set Event: SetUp Packet
Data Length <= EP0 ?
Stall Control Ept
Y
Control State <- USB_REC
End Of Handler

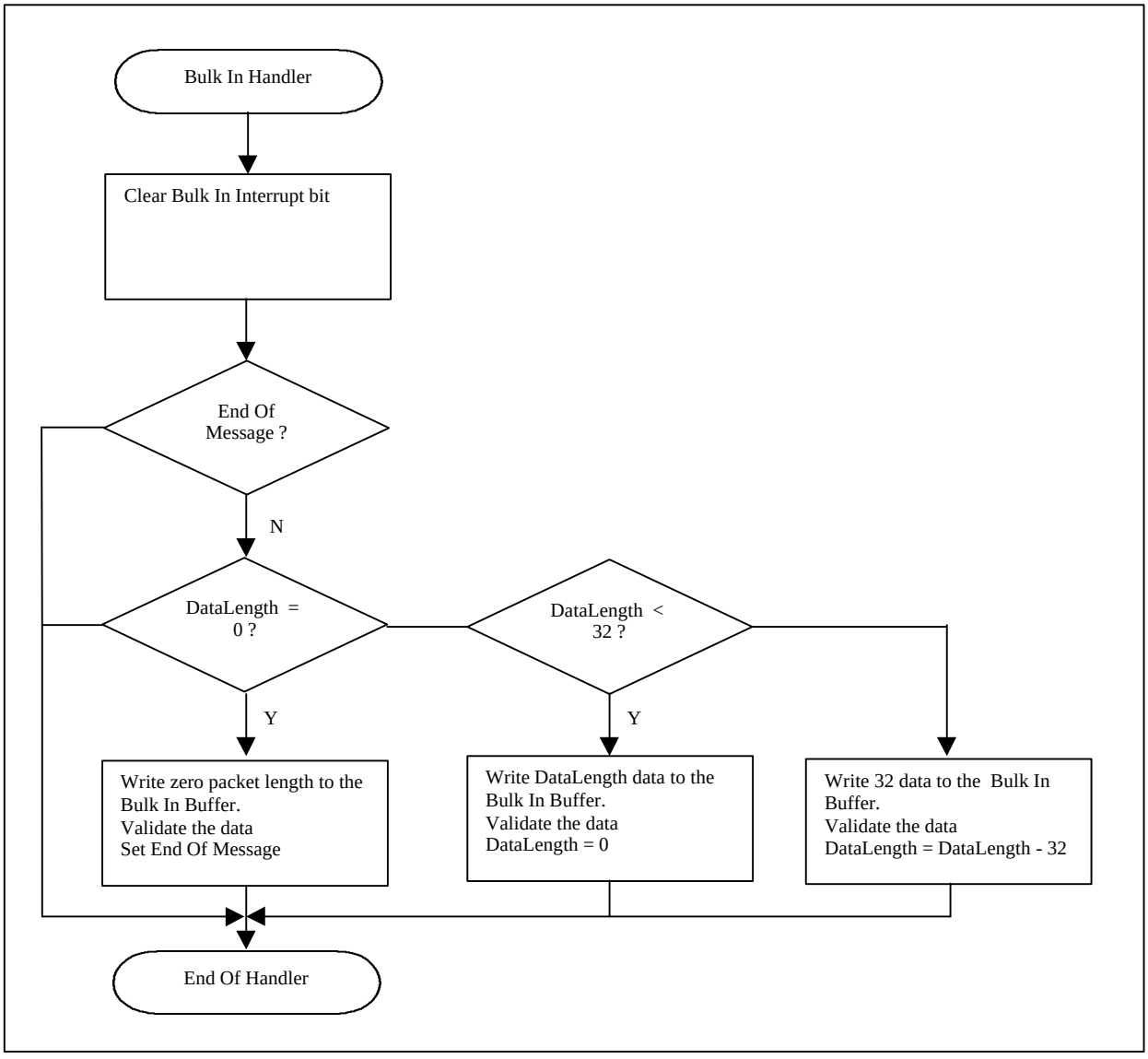
3.2.3 Bulk Out Handler

The bulk out endpoint is configured to receive data from the host. When the Usb interface receives OUT tokens from the host (identified through the Read Interrupt Register), the related interrupt bit has to be cleared. This flowchart below shows the processing of data incoming to this endpoint.



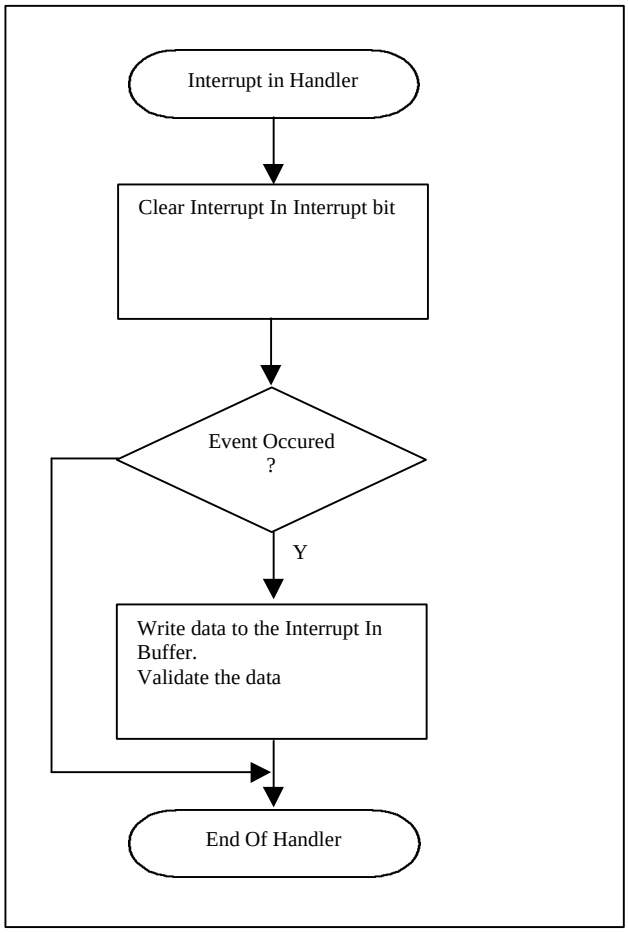
3.2.4 Bulk In Handler

The bulk in endpoint is configured to send data to the host. When the Usb interface receives IN tokens from the host (identified through the Read Interrupt Register), the related interrupt bit has to be cleared. This flowchart below shows the processing of data outcoming from this endpoint.



3.2.5 Interrupt In Handler

The interrupt in endpoint is configured to trigger card movement or emergency deactivation events to the host . When the Usb interface receives IN tokens from the host (identified through the Read Interrupt Register), the related interrupt bit has to be cleared. This flowchart below shows the processing of data outcoming from this endpoint.



3.3 DESCRIPTION OF THE USB LIBRARY FUNCTIONS

The library USB.LIB is made of three modules: Devhl.c, Chap9.c and Devci.c. The export functions of these modules are described below and can be found in the corresponding header modules (Devhl.h and Devci.h). The module Chap9.c does not export any functions.

The file Edphl.c is considered as not a part of the library. The body of the handlers EDP_BulkOut, EDP_BulkIn and EDP_InterruptIn can be recoded for every specific application. Nevertheless, their names can not be changed because they are referenced in the module Devhl.c.

All the USB library functions use bank 0 and should be called in the same bank.

MODULE DEVHL.C			
Function Name	**Syntax**	**Description**	**Stack Level**
InitialiseUsb	void InitialiseUsb(void)	This function initialises the USB interface. It should be called after every power-on -reset of the TDA8030.	3
UsbHandler	void UsbHandler(void)	This function handles all the USB events triggered by the USB interface of the TDA8030. It should be called in the main loop of the application.	6
MODULE EDPHL.C			
Function Name	**Syntax**	**Description**	**Stack Level**
EDP_BulkOut	void EDP_BulkOut(void)	This bulk out handler is application-dependent and so can be recoded for specific application.	3
EDP_BulkIn	void EDP_BulkIn(void)	This bulk in handler is application-dependent and so can be recoded for specific application.	3
EDP_InterruptIn	void EDP_InterruptIn(void)	This interrupt in handler is application-dependent and so can be recoded for specific application.	3
EDP_TxCardData	void EDP_TxCardData (unsigned int StartAddress, unsigned int CardDataLength)	This function is used to initiate a data transfer via the bulk in endpoint.	1
MODULE CHAP9.C			
Function Name	**Syntax**	**Description**	**Stack Level**
This module contains no export functions. They are related to the USB standard requests: - get status - clear feature - set feature - set address - get descriptor - get configuration - set configuration - get interface - set interface.			
MODULE DEVCI.C			
Function Name	**Syntax**	**Description**	**Stack Level**
WriteCommand	void WriteCommand (unsigned char bCommand)	This function sends a command to the USB interface	1
WriteOneData	void WriteOneData (unsigned char bCommand, (unsigned char bDataByte)	This function writes a databyte to the specified command.	1
ReadOneData	unsigned char ReadOneData (unsigned char bCommand)	This functions returns a databyte read from the specified command.	2
SetAddressEnable	void SetAddressEnable (unsigned char bAddress, bit fEnable)	This functions allows to assign a device address to the USB interface.	2
SetEndpointEnable	void SetEndpointEnable (bit fEnable)	This functions enables all endpoints of the USB interface	2
SetMode	void SetMode (unsigned char bMode)	This function writes a databyte to the command SetMode (F3h)	2

ReadInterruptRegister1	unsigned char ReadInterruptRegister1(void)	This functions returns a databyte read from the command ReadInterruptRegister (F4h).	2
GetDeviceStatus	unsigned char GetDeviceStatus(void)	This functions returns a databyte read from the command GetDeviceStatus (FEh).	2
SelectEndpoint	unsigned char SelectEndpoint (unsigned char bCommand)	This functions returns a databyte read from the command SelectEndpoint (40-46h).	2
ReadLastTransactionStatus	unsigned char ReadLastTransactionStatus (unsigned char bCommand)	This functions returns a databyte read from the command SelectEndpoint/ClearInterrupt (40-46h).	2
ReadEndpointStatus	unsigned char ReadEndpointStatus (unsigned char bCommand)	This functions returns a databyte read from the command SelectEndpoint (00-06h).	2
SetEndpointStatus	void SetEndpointStatus (unsigned char bCommand, unsigned char bEndpointStatus)	This function sends the command SetEndpointStatus (command 40-46h).	2
SendResume	void SendResume(void)	This function allows to generate a remote wakeup of the Usb interface in suspend.	2
ReadEndpoint	unsigned char ReadEndpoint (unsigned char bCommand, unsigned char idata *pbBuf, unsigned char bLen)	This function reads bLen databyte from the specified command bCommand.	3
WriteEndpoint	unsigned char WriteEndpoint (unsigned char bCommand, unsigned char idata *pbBuf, unsigned char bLen)	This function writes bLen databyte to the specified command.	3
AcknowledgeEndpoint	yoid AcknowledgeEndpoint (unsigned char bCommand)	This function sets up acknowledge to the specified control endpoint.	2
stall_ep0	void stall_ep0(void)	This functions stalls both the control in/out endpoints.	2
transmit_empty_ packet	void transmit_empty_packet (void)	This function load a zero packet in the Control In buffer.	3
single_transmit	void single_transmit (unsigned char idata *pbBuf, unsigned char bLen)	This function writes bLen databyte to the Control In buffer.	2

3.4 CODING EXAMPLES

3.4.1 Example of main loop

```
/*===========================================================================*/
/*
;     SOURCE_FILE:      MAIN.C
;     APPLICATION:      TDA8030
;     VERSION:          1.0
;     DATE:             08/03/2001
;
;     (C) 2001:         PHILIPS COMPONENTS & SEMICONDUCTORS
;                       SYSTEM & APPLICATIONS -CAEN
;
;     All rights are reserved. Reproduction in whole or in part is
;     prohibited without the prior written consent of the copyright
;     owner. The information presented in this document does not
;     form part of any quotation or contract, is believed to be
;     accurate and reliable and may be changed without notice.
;     No liability will be accepted by the publisher for any
;     consequence of its use. Publication thereof does not convey
;     nor imply any license under patent- or other industrial or
;     intellectual property rights.
*/
/*===========================================================================*/

/*===========================================================================*/
/*     I N C L U D E S                                                       */
/*===========================================================================*/

#include "common.h"
#include "./aud26/edphl.h"
#include "./aud26/devci.h"
#include "./aud26/devhl.h"

/*===========================================================================*/
/*
    FUNCTION NAME:      main
    DESCRIPTION:        main loop of the application CAKE8030
*/
void main(void)
/*===========================================================================*/
{
   test_card_move = TRUE;                // trick !!! to update the card status after
   card_move_latched = TRUE;             // trick !!! power-on_reset

   init_system(512);
   DEC_initialise_usb_hardware();
   InitialiseUsb();                   // initialise usb device
   while (TRUE)
   {
      UsbHandler();

         if (EDP_OpcodeReceived)
      {  // if usb opcode received then process the opcode
         DEC_handle_rx_usb_opcode();
      }
      proceed_alarm();        // proceed a possible alarm
```

```
     if (test_card_move)
     {
        EX0            = 0;      // when debouncing timer reached then test presence bit
        test_card_move = 0;

        if (card_move_latched)
        {
            DEC_card1_move = 1;
            DEC_card1_presence = ((read_register(MSR) & 0x04) == 0x04);
            card_move_latched = 0;
        }
        EX0 = 1;
     }
   }
}

//=========================================================================//
static void proceed_alarm(void)
//=========================================================================//
{
   EX0 = 0;

   if ((read_alarm()) && (test_card_move == 0))
   {
      if (read_error())
      {
         // trigger alarm message to the host
         // to be implemented later
      }
      else if (card_move_latched |= read_alarm_card_moved())
      {
         // latched presence, then start debouncing timer0 (20ms @ 14.745MHz)
         TL0 = (uchar)(~(24575)+1);
         TH0 = (uchar)((~(24575)+1)>>8);
         ET0 = 1;
         TR0 = 1;
      }
   }
   EX0 = 1;
}

//=========================================================================//
int_timer0() interrupt 1 using 1 // Timer 0 is used to manage the card debouncing.
//=========================================================================//
{
   TR0            = 0;
   ET0            = 0;
   test_card_move = 1;
   alarm          = 1;
}

/*=========================================================================*/
```

3.4.2 Example of opcode handler

```
/*=============================================================================*/
;     SOURCE_FILE:      DECODE.C
;     APPLICATION:      TDA8030
;     VERSION:
;     DATE:             02-02-2001
;
;     (C) 2001:         PHILIPS COMPONENTS & SEMICONDUCTORS
;                       APPLICATION LABORATORIES - CAEN
;
;     All rights are reserved. Reproduction in whole or in part is
;     prohibited without the prior written consent of the copyright
;     owner. The information presented in this document does not
;     form part of any quotation or contract, is believed to be
;     accurate and reliable and may be changed without notice.
;     No liability will be accepted by the publisher for any
;     consequence of its use. Publication thereof does not convey
;     nor imply any license under patent- or other industrial or
;     intellectual property rights.
*/
/*=============================================================================*/

/*=============================================================================*/
/*     I N C L U D E S                                                         */
/*=============================================================================*/

#include "common.h"
#include "decode.h"
#include ".\aud26\edphl.h"
#include ".\lib\lib.h"
#include ".\lib\tdalib.h"

/*=============================================================================*/
/*     G L O B A L   D E F I N I T I O N S                                     */
/*=============================================================================*/

/*=========================================================================*/
/*
    ITEM NAME:          DEC_card1_presence
    DESCRIPTION:        card1_presence is TRUE (FALSE) if a card (not)
                        inserted in the reader 1
/*=========================================================================*/

uchar xdata data_exch_buff[300];
bool DEC_card1_presence;
bool DEC_power_down_mode;
bool DEC_card1_move;

/*=============================================================================*/
/*     L O C A L   F U N C T I O N S   P R O T O T Y P E S                     */
/*=============================================================================*/

static void handle_mask_number(void);
static void handle_card_cmd(void);
static void handle_negociate(void);
static void handle_ifs_request(void);
static void handle_set_clock_card(void);
static void handle_set_card_baud_rate(void);
static void handle_power_down(void);
static void handle_power_up_5V(void);
static void handle_power_up_3V(void);
//static void handle_write_register(void);
//static void handle_read_register(void);

//static void handle_idle_clk_high(void);
//static void handle_idle_clk_low(void);
//static void handle_power_down_mode(void);
static void handle_check_card_presence(void);
```

```
static void handle_check_reader_status(void);

static void handle_test_mirror(void);
static void handle_bad_opcode(void);

static void LoadAnswerHeader(void);

/*=========================================================================*/
/*     L O C A L   V A R I A B L E S   D E F I N I T I O N S               */
/*=========================================================================*/
//static bit flag_power_up_S10;

/*=========================================================================*/
/*
    ITEM NAME:          COMMAND_FUNCTION_NEEDED_STRUCT
    DESCRIPTION:
*/
/*=========================================================================*/
typedef struct
{
   uchar                command;
   VOID_FUNCTION_PTR    function;
   uchar             CardNeeded;
}  COMMAND_FUNCTION_NEEDED_STRUCT;


/*=========================================================================*/
/*
    ITEM NAME:          DataLength
    DESCRIPTION:        DataLength contains the data number received from the card.

*/
/*=========================================================================*/

static word idata DataLength;
static uchar idata ErrorCode;

static uchar code num_mask[] = {"Cake8030-4-5"};
static uchar code date[] = __DATE__;
static uchar code time[] = __TIME__;

/*=========================================================================*/
/*
    ITEM NAME:          OPCODE_FUNCTION_TABLE
    PACKAGE:
    SCOPE:              COMPONENT
    DESCRIPTION:

    DEFINITION:
*/
/*=========================================================================*/

static COMMAND_FUNCTION_NEEDED_STRUCT code OPCODE_FUNCTION_TABLE[]=
{
   {  CHECK_CARD_PRESENCE,       handle_check_card_presence, 0},
   {  CARD_CMD,                  handle_card_cmd,        1},
   {  POWER_UP_5V,               handle_power_up_5V,        1},
   {  POWER_UP_3V,               handle_power_up_3V,        1},

   {  MASK_NUMBER,               handle_mask_number,        0},
   {  POWER_OFF,                 handle_power_down,         0},
   {  NEGOCIATE,                 handle_negociate,       1},
   {  POWER_DOWN_MODE,           handle_power_down_mode,     0},

   {  SET_CLOCK_CARD,            handle_set_clock_card,     1},
   {  IFS_REQUEST,               handle_ifs_request,        1},
   {  SET_CARD_BAUD_RATE,        handle_set_card_baud_rate, 1},
   {  CHECK_READER_STATUS,       handle_check_reader_status,   0},
   {  TEST_MIRROR,               handle_test_mirror,         0},
   {  BAD_OPCODE,                   handle_bad_opcode,         0}
};
```

```
/*=========================================================================*/
/*
    ITEM NAME:          PATTERN, MSB_LEN, LSB_LEN, OPCODE, ERROR_CODE
    DESCRIPTION:
*/
/*=========================================================================*/

#define  PATTERN        START_USB_BUFFER_HEADER
#define  MSB_LEN        PATTERN+1
#define  LSB_LEN        PATTERN+2
#define  OPCODE         PATTERN+3
#define  ERROR_CODE     PATTERN+4

/*=========================================================================*/
/*     E X P O R T E D    F U N C T I O N S                                */
/*=========================================================================*/

/*=========================================================================*/
/*
    FUNCTION NAME:      DEC_handle_rx_opcode
    DESCRIPTION:
*/
void DEC_handle_rx_usb_opcode(void)
/*=========================================================================*/
{
   uchar index = 0;
   DataLength = 0;              // reset this variable
   ErrorCode = NO_ERROR;        // reset error code

   if (EDP_MessageValid)
   {
         while ( (OPCODE_FUNCTION_TABLE[index].command != EDP_Opcode) &&
                 (OPCODE_FUNCTION_TABLE[index].command != BAD_OPCODE) )
         index ++;

         if (OPCODE_FUNCTION_TABLE[index].CardNeeded && !check_pres_card())
      {  // if the execution of the opcode needs card presence
         ErrorCode =   CARD_ABSENT;
         alarm     = 1;               // trigger alarm
      }
      else
      {  // if no error then execute the opcode
         OPCODE_FUNCTION_TABLE[index].function();
      }
   }
   else
   {  // if message not valid then set unknown command ...
      ErrorCode = UNKNOWN_COMMAND;
   }

   DEC_LoadMessage();
   EDP_OpcodeReceived = FALSE;                                  // the request is
processed, reset this flag
}

/*=========================================================================*/
/*
    FUNCTION NAME:      DEC_initialise_hardware
    DESCRIPTION:        This function is called after a power-on to initialise the hardware
                        of the tda8030.
*/
void DEC_initialise_usb_hardware(void)
/*=========================================================================*/
{
//    init_system(DATA_EXCH_BUFF_SIZE);   // initialise tda8007
         // for RB+, buffer is limited to  256 bytes
   DEC_power_down_mode = FALSE;
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      DEC_LoadMessage
    DESCRIPTION:        this function loads 4 bytes for the answer string:
                           - if the transmission is successful
                             60h, msb of length, lsb of length, Opcode
                           - if there is an error
                             E0h, msb of length, lsb of length, Opcode, error number
                        and transmits the data
*/
void DEC_LoadMessage(void)
/*=========================================================================*/
{
   LoadAnswerHeader();
   EDP_TxCardData(START_USB_BUFFER_HEADER, (DataLength + OFFSET));
}

/*=========================================================================*/
/*     S T A T I C   F U N C T I O N S                                     */
/*=========================================================================*/

/*=========================================================================*/
/*
    FUNCTION NAME:      handle_check_card_presence
    DESCRIPTION:        This function returns the card presence in the reader.

*/
static void handle_check_card_presence(void)
/*=========================================================================*/
{
   data_exch_buff[OFFSET] = check_pres_card();
   DataLength = 1;
}

/*=========================================================================*/
/*
    FUNCTION NAME:      handle_command_card
    DESCRIPTION:        This function processes command card opcode
*/
static void handle_card_cmd(void)
/*=========================================================================*/
{
   DataLength = XmitAPDU(OFFSET, EDP_Length);
}

/*=========================================================================*/
/*
    FUNCTION NAME:      handle_mask_number
    DESCRIPTION:        This functions returns the mask number.
*/
static void handle_mask_number(void)
/*=========================================================================*/
{
   uchar index;
   for (index = 0; index < sizeof(num_mask) - 1; index++)
        data_exch_buff[OFFSET + index] = num_mask[index];
   // do not send the character EndOfString (0x00) of the string MASK_NR_STR[]
   DataLength = (sizeof(num_mask) - 1);
}

/*=========================================================================*/
/*
    FUNCTION NAME:  handle_power_down
    DESCRIPTION:  This function
*/
static void handle_power_down(void)
/*=========================================================================*/
{
   power_down();
}
```

```
/*===========================================================================*/
/*
    FUNCTION NAME:  handle_power_up_5V
    DESCRIPTION:     This function active a card with parameter given in
                     command.
*/
static void handle_power_up_5V(void)
/*===========================================================================*/
{
   DataLength = power_up(OFFSET, V5);
}

/*===========================================================================*/
/*
    FUNCTION NAME:      handle_power_down_mode
    DESCRIPTION:
*/
static void handle_power_down_mode(void)
/*===========================================================================*/
{
   DEC_power_down_mode = TRUE;
}

/*===========================================================================*/
/*
    FUNCTION NAME:  handle_power_up_3V
    DESCRIPTION:  This function performs a 3V activation of an async. card.
*/
static void handle_power_up_3V(void)
/*===========================================================================*/
{
   DataLength = power_up(OFFSET, V5);
}

/*===========================================================================*/
/*
    FUNCTION NAME:     handle_negociate
    DESCRIPTION:       This function negociates with card.
*/
static void handle_negociate(void)
/*===========================================================================*/
{
   DataLength = negotiate(OFFSET, data_exch_buff[OFFSET], data_exch_buff[OFFSET + 1]);
}

/*===========================================================================*/
/*
    FUNCTION NAME:     handle_set_clock_card
    DESCRIPTION:       This function sets a new clock card frequency.

*/
static void handle_set_clock_card(void)
/*===========================================================================*/
{
   set_clock_card(data_exch_buff[OFFSET]);
   DataLength = 1;
}

/*===========================================================================*/
/*
    FUNCTION NAME:  handle_set_card_baud_rate
    DESCRIPTION:    This function sets a new card baud rate. The baud
                    rate take place just after this command.
*/
static void handle_set_card_baud_rate(void)
/*===========================================================================*/
{
   if (set_baud_rate(data_exch_buff[OFFSET], data_exch_buff[OFFSET + 1]))
      DataLength = 2;
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:     handle_check_reader_status
    DESCRIPTION:       This functions returns to the host the reader status.
*/
static void handle_check_reader_status(void)
/*=========================================================================*/
{
   // store in data_exch_buff array
   data_exch_buff[OFFSET] = ((uchar)DEC_card1_presence << 1) + (uchar)DEC_card1_move;
   DataLength = 1;                 // 2 bytes to send
   DEC_card1_move = 0;                // reset card move informations
}

/*=========================================================================*/
/*
    FUNCTION NAME:     handle_ifs_request
    DESCRIPTION:       This function sends an ifs request to the card.
*/
static void handle_ifs_request(void)
/*=========================================================================*/
{
   send_Sblock_IFSD(data_exch_buff[OFFSET]);
}

/*=========================================================================*/
/*
    FUNCTION NAME:     handle_test_mirror
    DESCRIPTION:       This functions returns the host the same amout of
                       received data. It is used for test purposes.
*/
static void handle_test_mirror(void)
/*=========================================================================*/
{
   DataLength = EDP_Length;
}

/*=========================================================================*/
/*
    FUNCTION NAME:     handle_bad_opcode
    DESCRIPTION:
*/
static void handle_bad_opcode(void)
/*=========================================================================*/
{
   ErrorCode = UNKNOWN_COMMAND;
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      LoadAnswerHeader
    DESCRIPTION:        this function loads 4 bytes for the answer string:
                           - if the transmission is successful
                             60h, msb of length, lsb of length, Opcode
                           - if there is an error
                             E0h, msb of length, lsb of length, Opcode, error number
*/
static void LoadAnswerHeader(void)
/*=========================================================================*/
{
   if (ErrorCode == NO_ERROR)
   {  // get error code
     ErrorCode = read_error();
   }
   if ((DataLength == 0) && (ErrorCode != NO_ERROR))
   {
      data_exch_buff[PATTERN]    = ERROR_HEADER;
      data_exch_buff[MSB_LEN]    = 0;
      data_exch_buff[LSB_LEN]    = 1;
      data_exch_buff[OPCODE]   = EDP_Opcode;
      data_exch_buff[ERROR_CODE] = ErrorCode;     // error code to be transmitted
      DataLength = 1;                             // 1 databyte to be transmitted
   }
   else
   {
      data_exch_buff[PATTERN] = NORMAL_HEADER;
      data_exch_buff[MSB_LEN] = DataLength >> 8;  // msb of length
      data_exch_buff[LSB_LEN] = DataLength;       // lsb of length
      data_exch_buff[OPCODE]  = EDP_Opcode;       // opcode
   }
   ErrorCode = NO_ERROR;                      // reset error code afterwards
}
```

3.4.3 Example of endpoint handlers

```
/*===========================================================================*/
/*
;     SOURCE_FILE:      EDPHL.C
;     APPLICATION:      TDA8030
;     VERSION:
;     DATE:             08/03/2001
;
;     (C) 2001:         PHILIPS COMPONENTS & SEMICONDUCTORS
;                       APPLICATION LABORATORIES - CAEN
;
;     All rights are reserved. Reproduction in whole or in part is
;     prohibited without the prior written consent of the copyright
;     owner. The information presented in this document does not
;     form part of any quotation or contract, is believed to be
;     accurate and reliable and may be changed without notice.
;     No liability will be accepted by the publisher for any
;     consequence of its use. Publication thereof does not convey
;     nor imply any license under patent- or other industrial or
;     intellectual property rights.
*/
/*===========================================================================*/

/*****************************************************************************/
/*

*/
/*****************************************************************************/

/*=============================================================================================*/
/*                                                                                             */
/*                   STATUS OF ENDPOINTS USED IN THIS APPLICATION                              */
/*                                                                                             */
/*          CONTROL OUT, index 0: Control Out                                                  */
/*          CONTROL IN , index 1: Control In                                                   */
/*          EPT 1 OUT  , index 2: Bulk Out                                                     */
/*          EPT 1 IN   , index 3: Bulk In                                                      */
/*          EPT 2 IN   , index 4: Interrupt In                                                 */
/*          EPT 3 IN   , index 5: Not used                                                     */
/*                                                                  */
/*=============================================================================================*/

/*=============================================================================================*/
/*                                                                                             */
/*                   LAYOUT OF USB TRANSACTION USED IN THIS PROJECT                            */
/*                                                                                             */
/*    HOST                                                                  DEVICE             */
/*                            BULK OUT                                                         */
/* 1     ------------------------------------------------------------------------>            */
/*       60h, Msb BulkOutLength, Lsb BulkOutLength, Opcode, data[1],..,data[BulkOutLength]     */
/*                                                                                             */
/*                            BULK IN                                                          */
/* 2     <------------------------------------------------------------------------            */
/*       60h, Msb BulkInLength, Lsb BulkInLength, Opcode, data[1],..., data[BulkInLength]      */
/*                                                                                             */
/*                            INTERRUPT IN                                                     */
/* 3     <------------------------------------------------------------------------            */
/*       CARD1_INSERTION (or CARD1_EXTRACTION)                                                 */
/*                                                                                             */
/*=============================================================================================*/

/*===========================================================================*/
/*     I N C L U D E S                                                       */
/*===========================================================================*/

//include "reg51rd.h"
#include ".\src\common.h"
#include "devci.h"
#include "devhl.h"
```

```
#include "edphl.h"
#include ".\src\decode.h"
#include ".\src\lib\tdalib.h"

/*=========================================================================*/
/*     G L O B A L   D E F I N I T I O N S                                 */
/*=========================================================================*/

/*=========================================================================*/
/*
    ITEM NAME:          EDP_OpcodeReceived, EDP_Opcode, EDP_Length
    DESCRIPTION:        EDP_OpcodeReceived is TRUE (FALSE) if an opcode is
                        (not) received.
                        EDP_MessageValid is TRUE (FALSE) if the message just
                        received is (not) valid.
                        EDP_Opcode contents the opcode value just received.
                        EDP_Length contents the length of the incoming databytes
                        received at BULK OUT endpoint.
*/
/*=========================================================================*/
bool  EDP_OpcodeReceived;
bool  EDP_MessageValid;
uchar data EDP_Opcode;
word  data EDP_Length;

/*=========================================================================*/
/*
    ITEM NAME:          DEVICE_DATA_REG, DEVICE_COMMAND_REG
    DESCRIPTION:        These constants define the addresses of the data
                        and the command registers used in this application.
*/
/*=========================================================================*/
// P26 = Chip Select of TDA8030
// P25 = Usb Command/Data Selector
#define DEVICE_DATA_REG         0x4000
#define DEVICE_DATA_REG_BIS     0x4001
#define DEVICE_COMMAND_REG      0x6000
unsigned char xdata DEVICE_DATA_REG;
unsigned char xdata DEVICE_DATA_REG_BIS;
unsigned char xdata DEVICE_COMMAND_REG;

/*
uchar xdata data_exch_buff[300];

bool   DEC_power_down_mode;
bool   DEC_card1_presence;
bool   DEC_card2_presence;

#define NO_MOVE         0
#define CARD1_EXTRACTION   0
#define CARD2_EXTRACTION   0
#define CARD_INSERTION     0
*/

/*=========================================================================*/
/*     L O C A L   S Y M B O L   D E C L A R A T I O N S                   */
/*=========================================================================*/
sbit USB_MP_READY            = P1^2;
#define WAIT_DEVICE_READY    while (!USB_MP_READY)

/*=========================================================================*/
/*     L O C A L   F U N C T I O N S   P R O T O T Y P E S                 */
/*=========================================================================*/
static uchar ReadFromBulkBuffer(void);
static void WriteToBulkBuffer(uchar DevByteCnt);
```

```
/*===========================================================================*/
/*     L O C A L   D A T A   D E F I N I T I O N S                           */
/*===========================================================================*/

/*===========================================================================*/
/*
    ITEM NAME:          TxData, wTxLength
    DESCRIPTION:        TxData is TRUE (FALSE) if sending of data to the host
                        is enabled (disabled) at BULK IN endpoint.
                        wTxLength contents the data number to be transmitted
                        via BULK IN endpoint.
*/
/*===========================================================================*/
static bool TxData;
static word idata wTxLength;

/*===========================================================================*/
/*
    ITEM NAME:          EndOfMessage
    DESCRIPTION:        EndOfMessage is TRUE when a message is over.
*/
/*===========================================================================*/
static bool EndOfMessage;

/*===========================================================================*/
/*
    ITEM NAME:          card1_presence
    DESCRIPTION:        card1_presence is TRUE (FALSE) if a card is (not) inserted
                        in the reader
*/
/*===========================================================================*/
static bool card1_presence;
static bool card2_presence;

/*===========================================================================*/
/*
    ITEM NAME:          wDataAddress
    DESCRIPTION:
*/
/*===========================================================================*/
static word data wDataAddress;

/*===========================================================================*/
/*
    ITEM NAME:          PROTOCOL_HEADER_LENGTH
    DESCRIPTION:
*/
/*===========================================================================*/
#define  PROTOCOL_HEADER_LENGTH     4
#define BULK_LENGTH                32

/*===========================================================================*/
/*
    ITEM NAME:          CARD_MOVE_STR,
    DESCRIPTION:        Fixed string of ascii characters
*/
/*===========================================================================*/
//static uchar code CARD_MOVE_STR[] = {NORMAL_HEADER, 0x00, 0x00, CARD_INSERTION};
```

```
/*=========================================================================*/
/*     E X P O R T E D    F U N C T I O N S                                 */
/*=========================================================================*/

/*=========================================================================*/
/*
    FUNCTION NAME:      EDP_GetRestOfMessage
    DESCRIPTION:        This function is called to process the rest of the message > 64 bytes
                         incoming to BULK OUT endpoint.
*/
void EDP_GetRestOfMessage(void)
/*=========================================================================*/
{
   if (!EndOfMessage)           // if not end of message
      EDP_BulkOut();                  // then get rest of message
}

/*=========================================================================*/
/*
    FUNCTION NAME:      EDP_IntEp3In
    DESCRIPTION:        This function is used when the endpoint 3 INTERRUPT IN is
                        addressed.
*/
void EDP_InterruptIn(void)
/*=========================================================================*/
{
   ReadLastTransactionStatus(READ_LTRX_EPT2_IN);     /* Clear interrupt flag */

   if (!(ReadOneData(SEL_EPT_2_IN) & SUCCESSFULLY))  // check if buffer empty
   {
      if (card1_presence != DEC_card1_presence)
      {
         card1_presence = DEC_card1_presence;        // update card presence
         bDataBuffer[0] = 0x00;
         bDataBuffer[1] = 0x00;
         bDataBuffer[2] = 0x00;
         bDataBuffer[3] = CARD1_EXTRACTION + (uchar)card1_presence;
         WriteRamToDevice(bDataBuffer, 4);              // 4 bytes to write to buffer
      }
   }
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      EDP_BulkOut
    DESCRIPTION:        This function is called to handle the sending of data to the
                        endpoint BULK OUT.
                       The data are received by packet of 32 bytes. The processing
                        goes on until the reception of a packet of which the length
                        is less than 32 bytes. After that, the request (opcode + data)
                        is considered as complete.
*/
void EDP_BulkOut(void)
/*=========================================================================*/
{
   ReadLastTransactionStatus(READ_LTRX_EPT1_OUT);     /* Clear interrupt flag */

   if (ReadOneData(SEL_EPT_1_OUT) & SUCCESSFULLY)     // check buffer full (CRC16 and EOP
                                                      // correctly received)
   {
      if (!EDP_OpcodeReceived)
      {   // if the previous message is processed
            if (EndOfMessage)
            {  // if the previous message is over then new message can be received
            wDataAddress = START_USB_BUFFER_HEADER;  // set the pointer of ram to receive
                                                      // data
            EndOfMessage = FALSE;                     // indicate not end of message
         }

         if (ReadFromBulkBuffer() < BULK_LENGTH)
         {  // if end of message then
            EDP_OpcodeReceived = TRUE;                     // indicate a message received
            EDP_MessageValid   = TRUE;                     // and assumed to be valid
            EndOfMessage       = TRUE;                     // indicate end of message
            // get total length i.e. protocol header + useful data
            wTxLength = wDataAddress - START_USB_BUFFER_HEADER;
            // reset the length of useful data
            EDP_Length = 0;

            if ((data_exch_buff[START_USB_BUFFER_HEADER] != NORMAL_HEADER) || (wTxLength <
            PROTOCOL_HEADER_LENGTH) )
            {  // if wrong header or length too short
               EDP_MessageValid = FALSE;          // indicate a invalid message received
            }
            else
            {
               EDP_Opcode = data_exch_buff[START_USB_BUFFER_HEADER+3];    // get EDP_Opcode
               EDP_Length = wTxLength - PROTOCOL_HEADER_LENGTH;           // get length
            }
         }
      }
   }
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      EDP_BulkIn
    DESCRIPTION:        This function is called to handle the sending of data
                        from the endpoint BULK IN.
                       The data are loaded at the current IN token and transmitted
                        by packet of max. 32 bytes at the next IN token.

                        The frames of the data are as follows:
                            1st frame: max 32 bytes
                                       Header 1
                                      Header 2
                                      Header 3
                                       Header 4
                                       Data[1]
                                       ...
                                       Data[32 - 4]
                            2nd frame: max 32 bytes
                                      Data[32 - 3]
                                       Data[32 - 2]
                                                    ....
                      If the length of the data packet is modulo 32, an
                      empty packet should be sent to end the transmission.
*/
void EDP_BulkIn(void)
/*=========================================================================*/
{
   uchar DevByteCnt;

   ReadLastTransactionStatus(READ_LTRX_EPT1_IN);     /* Clear interrupt flag */

   if (!(ReadOneData(SEL_EPT_1_IN) & SUCCESSFULLY))  // check if buffer empty
   {                                                 // then
      if (TxData)                                    // if data left
      {                                              // then load
         if (wTxLength >= BULK_LENGTH)
         {
            DevByteCnt = BULK_LENGTH;
            wTxLength  -= BULK_LENGTH;
         }
         else
         {
            DevByteCnt = wTxLength;
            wTxLength = 0;
            //EndOfMessage = TRUE;                // then end of message
            TxData = FALSE;                       // then end
            //EDP_OpcodeReceived = FALSE;         // clear this flag to receive a new message
         }
         WriteToBulkBuffer(DevByteCnt);           // load to bulk in buffer
      }
   }
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      EDP_TxCardData
    DESCRIPTION:        This function is called to initiate the data transfer to
                        the endpoint BULK IN.
                        The transmission of the remainder of data is performed
                        afterwards by the function EDP_BulkIn.
*/
void EDP_TxCardData(word StartAddress, word CardDataLength)
/*=========================================================================*/
{
   // get length of data to be sent to the host via BULK IN endpoint
   wTxLength = CardDataLength;
   // initialise the address of the data buffer
   wDataAddress = StartAddress;
   // enable the transmission of data to the host via BULK IN endpoint
   TxData = TRUE;
}

/*=========================================================================*/
/*     S T A T I C   F U N C T I O N S                                     */
/*=========================================================================*/

/*=========================================================================*/
/*
    FUNCTION NAME:      ReadFromBulkBuffer
    DESCRIPTION:        This function performs a direct reading from the buffer
                        of the device to the data array pointed by the pointer
                        wDataAddress.
                         Before calling this function, the pointer wDataAddress
                        should be initialised and the endpoint should be
                        selected.
                       The data read are stored in the following way:

                        data_exch_buffer[START_USB_BUFFER_HEADER + 0]   = pattern
                        data_exch_buffer[START_USB_BUFFER_HEADER + 1]   = Msb Length
                        data_exch_buffer[START_USB_BUFFER_HEADER + 2]   = Lsb Length
                        data_exch_buffer[START_USB_BUFFER_HEADER + 3]   = Opcode
                        data_exch_buffer[START_USB_BUFFER_HEADER + 4]   = data 1
                        ...
                        data_exch_buffer[START_USB_BUFFER_HEADER + 4 + n] = data n
*/
static uchar ReadFromBulkBuffer(void)
/*=========================================================================*/
{
   uchar DummyByte, DevByteCnt, DevDataVal, SaveDevByteCnt;

   WriteCommand(READ_BUFFER);               // select read buffer
   //SELECT_DATA;
   DummyByte  = DEVICE_DATA_REG;            // 1st databyte = dummy byte
   WAIT_DEVICE_READY;                       // wait device ready
   DevByteCnt = DEVICE_DATA_REG_BIS;        // 2nd databyte = number of available databytes
   WAIT_DEVICE_READY;
   SaveDevByteCnt = DevByteCnt;             // save the number of available databytes

   for ( ; DevByteCnt > 0; DevByteCnt --)
   {
      //SELECT_EXTRAM;                      // select external ram
      DevDataVal = DEVICE_DATA_REG;         // get databyte from D12 device
      WAIT_DEVICE_READY;                    // wait device ready
      //DESELECT_EXTRAM;                    // deselect external ram
         data_exch_buff[wDataAddress] = DevDataVal;  // store the databyte
      wDataAddress ++;                      // increment the datapointer
   }
   WriteCommand(CLEAR_BUF);                 // clear buffer after reading

   return(SaveDevByteCnt);                  // return the number of available databytes
}
```

```
/*=========================================================================*/
/*
    FUNCTION NAME:      WriteToBulkBuffer
    DESCRIPTION:        This function writes directly data to the buffer of the
                        device. The data is get from a data array pointed by the
                        pointer wDataAddress.
                         Before calling this function, the pointer wDataAddress
                         should be initialised.
                         The data to be written are stored in the array data_exch_buff
                        in the following order:

                        data_exch_buffer[0]   = header
                        data_exch_buffer[1]   = Msb Length
                        data_exch_buffer[2]   = Lsb Length
                        data_exch_buffer[3]   = Opcode
                        data_exch_buffer[4]   = data 1
                        ...
                        data_exch_buffer[4+n] = data n
*/
static void WriteToBulkBuffer(uchar DevByteCnt)
/*=========================================================================*/
{
   uchar DevDataVal;

   WriteCommand(WRITE_BUFFER);                 // select write buffer
   //SELECT_DATA;                          // select data
   DEVICE_DATA_REG = 0;                     // always 0
   WAIT_DEVICE_READY;                       // wait device ready
   DEVICE_DATA_REG_BIS = DevByteCnt;           // number of bytes
   WAIT_DEVICE_READY;                       // wait device ready

   for ( ; DevByteCnt > 0; DevByteCnt --)
   {
       //DESELECT_EXTRAM;                          // deselect external ram
      DevDataVal = data_exch_buff[wDataAddress];  // get the byte
      wDataAddress ++;                    // increment datapointer
      //SELECT_EXTRAM;                    // select external ram
      DEVICE_DATA_REG = DevDataVal;               // write databyte to device
      WAIT_DEVICE_READY;                       // wait device ready
   }
   WriteCommand(VALIDATE_BUF);                 // validate the buffer after writing
}
```