Embedded Solutions

BIOS-API

Manual

rev. 1.6

ADL Embedded Solutions Inc. 4411 Morena Blvd., Suite 101 San Diego, CA 92117-4345
P. +1 858 490-0597 F. +1 858 490-0599
e-mail: sales@adl-usa.com; web: www.adl-usa.com

Contents

0 Document History

Version	Changes
1.0	first released version
1.1	added information on driver design
1.2	added error codes
1.3	added two comments in code example
1.4	added GPIO services
1.5	added 64bit support, added new functions for watchdog and user area, added another code sample in the annex
1.6	size of user data area corrected to 128 bytes; added chapter on driver installation

NOTE

1 Introduction

1.1 Important Notes

Please read this manual carefully before you begin installation of this hardware device. To avoid Electrostatic Discharge (ESD) or transient voltage damage to the board, adhere to the following rules at all times:

- You must discharge your body from electricity before touching this board.
- Tools you use must be discharged from electricity as well.
- Please ensure that neither the board you want to install, nor the unit on which you want to install this board, is energized before installation is completed.
- Please do not touch any devices or components on the board.

CAUTION

As soon as the board is connected to a working power supply, touching the board may result in electrical shock, even if the board has not been switched on yet. Please also note that the mounting holes for heat sinks are connected to ground, so when using an externally AC powered device, a substantial ground plane differential can occur if the external device's AC power supply or cable does not include an earth ground. This could also result in electrical shock when touching the device and the heat sink simultaneously.

1.2 Technical Support

Technical support for this product can be obtained in the following ways:

- By contacting our support staff at +1 858-490-0597 or +49 (0) 271 250 810 0
- By contacting our staff via e-mail at support@adl-usa.com or support@adl-europe.com
- Via our website at www.adl-usa.com/support or www.adl-europe.com/support

1.3 Warranty

This product is warranted to be free of defects in workmanship and material. ADL Embedded Solutions' sole obligation under this warranty is to provide replacement parts or repair services at no charge, except shipping cost. Such defects which appear within 12 months of original shipment of ADL Embedded Solutions will be covered, provided a written claim for service under warranty is received by ADL Embedded Solutions no less then 30 days prior to the end of the warranty period of within 30 days of discovery of the defect – whichever comes first. Warranty coverage is contingent upon proper handling and operation of the product. Improper use such as unauthorized modifications or repair, operation outside of specified ratings, or physical damage may void any service claims under warranty.

1.4 Return Authorization

All equipment returned to ADL Embedded Solutions for evaluation, repair, credit return, modification, or any other reason must be accompanied by a Return Material Authorization (RMA) number. ADL Embedded Solutions requires a completed RMA request form to be submitted in order to issue an RMA number. The form can be found under the Support section at our website: www.adl-usa.com or www.adl-europe.com. Submit the completed form to support@adl-usa.com or fax to +1 858-490-0599 for the USA office, or to rma@adl-europe.com or fax to +49 (0) 271 250 810 20 to request an RMA from the European office in Germany. Following a review of the information provided, ADL Embedded Solutions will issue an RMA number.

1.5 Description of Safety Symbols

The following safety symbols are used in this documentation. They are intended to alert the reader to the associated safety instructions.

ACUTE RISK OF INJURY!

If you do not adhere to the safety advise next to this symbol, there is immediate danger to life and health of individuals!

RISK OF INJURY!

If you do not adhere to the safety advise next to this symbol, there is danger to life and health of individuals!

HAZARD TO INDIVIDUALS, ENVIRONMENT, DEVICES, OR DATA!

If you do not adhere to the safety advise next to this symbol, there is obvious hazard to individuals, to environment, to materials, or to data.

NOTE OR POINTER

This symbol indicates information that contributes to better understanding.

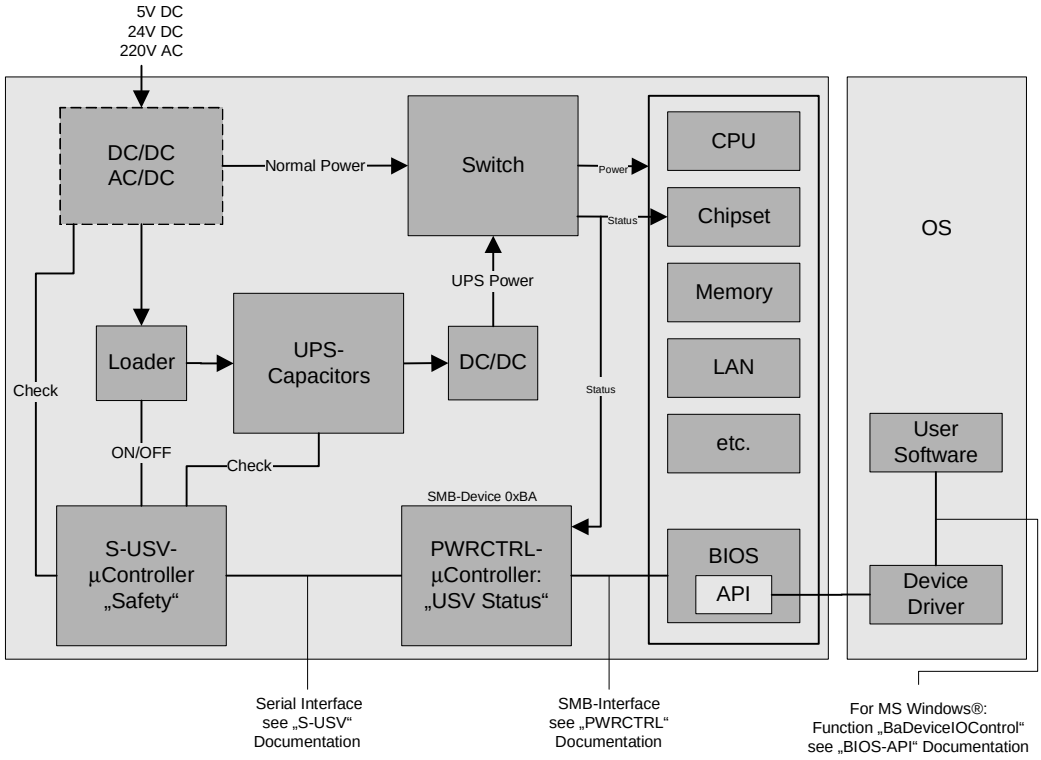
1.6 RoHS

The PCB and all components are RoHS compliant (RoHS = Restriction of Hazardous Substances Directive). The soldering process is lead free.

2 Overview

2.1 General

The “BIOS-API” is a piece of software which is part of the BIOS in our industrial motherboards. It offers a one-stop solution for communicating with several components on the board, such as temperature and voltage sensors, the S-USV microcontroller, the PWRCTRL microcontroller, the Watchdog and other components (if installed). It also offers access to a small memory area in the EEPROM reserved for user data. The following diagram, taken from the S-USV documentation, illustrates the general setup in the lower right corner:



As can be seen, the API is integrated into the BIOS. The OS which is running on the board needs to have a special Device Driver installed to access the API functions. It is through this driver that user software can take advantage of the API functionality. For Microsoft® Windows® operating systems, there is a driver available which is being shipped upon request (32bit and 64bit support available).

2.2 Features

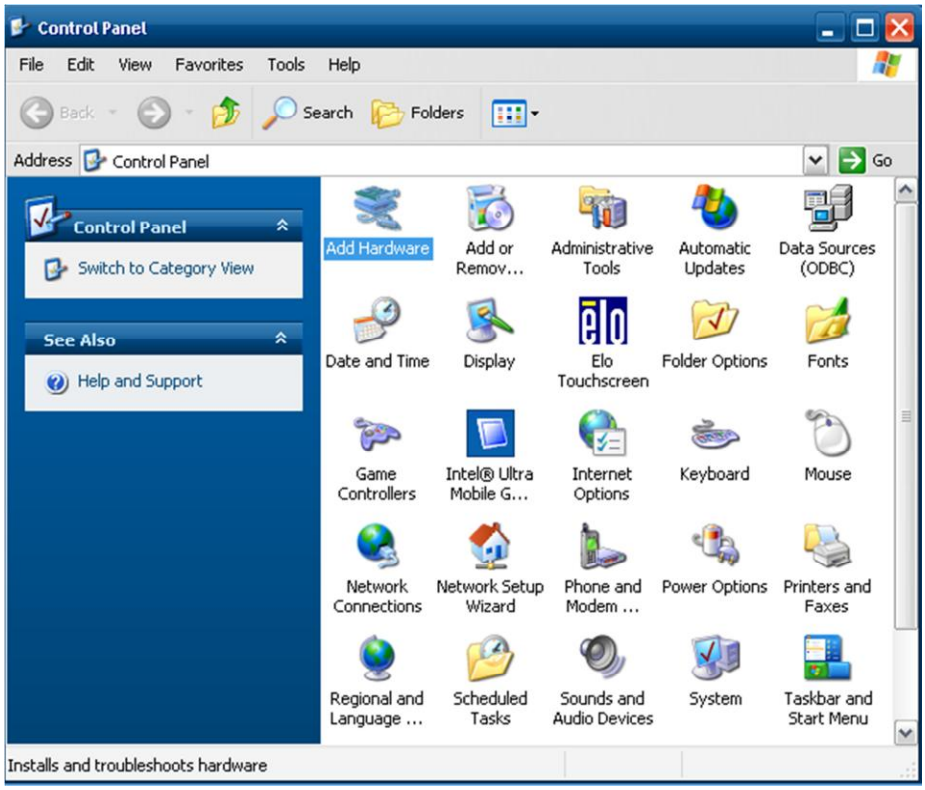
The API offers the following features which will be explained in more detail in later chapters of this document:

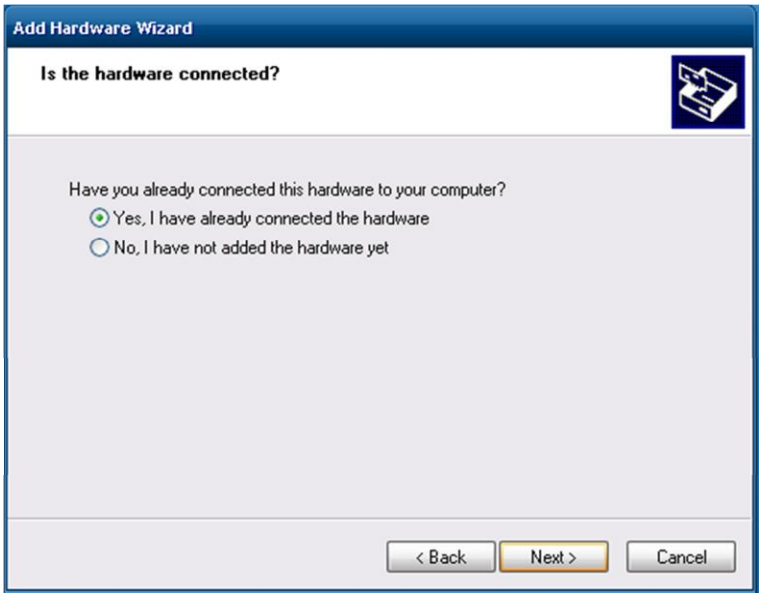
- Read out temperature sensors.
- Read out voltage sensors.
- Read access to PWRCTRL microcontroller.
- Read and Write access to S-USV microcontroller.
- Enabling, disabling, and configuring Watchdog.
- Control system LEDs.
- Read out software information (API version, BIOS version).
- Read out hardware information (Platform, Board name, Board revision).

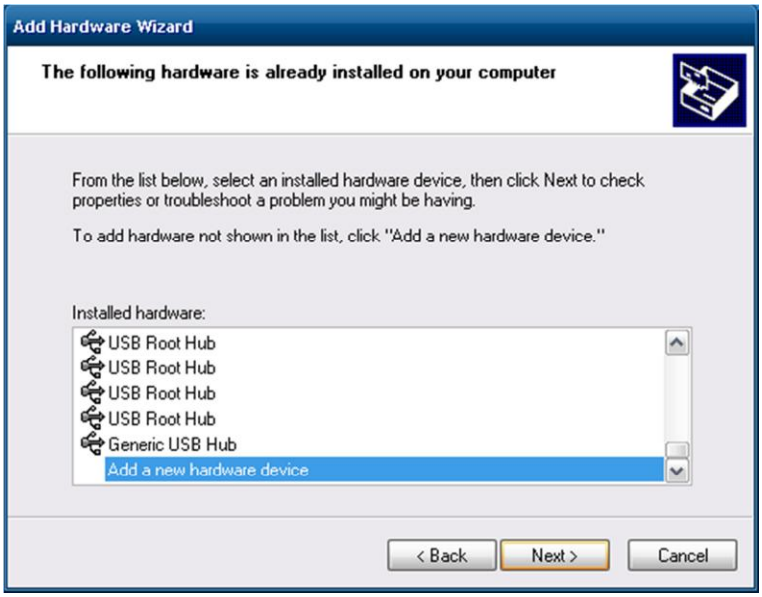
Note: For a particular feature to be available, the corresponding hardware (such as the S-USV microcontroller, for example) must be present on the motherboard.

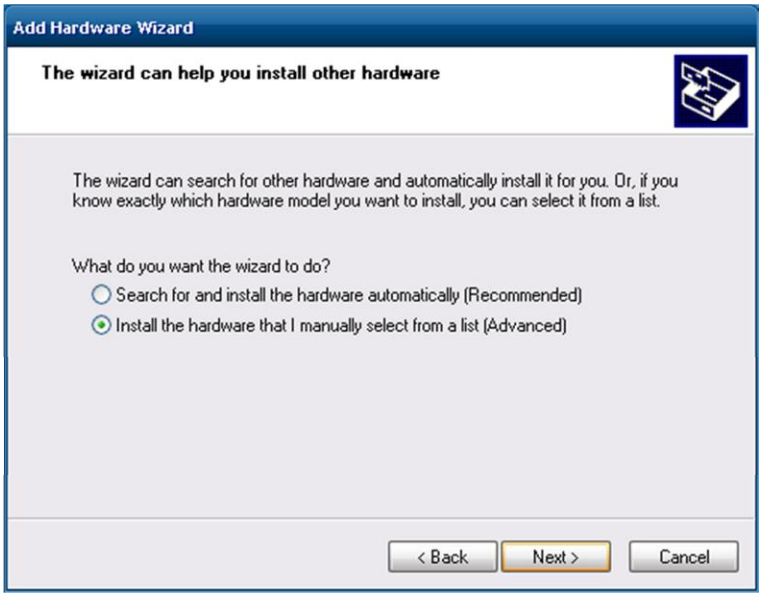
3 Windows® Driver Installation

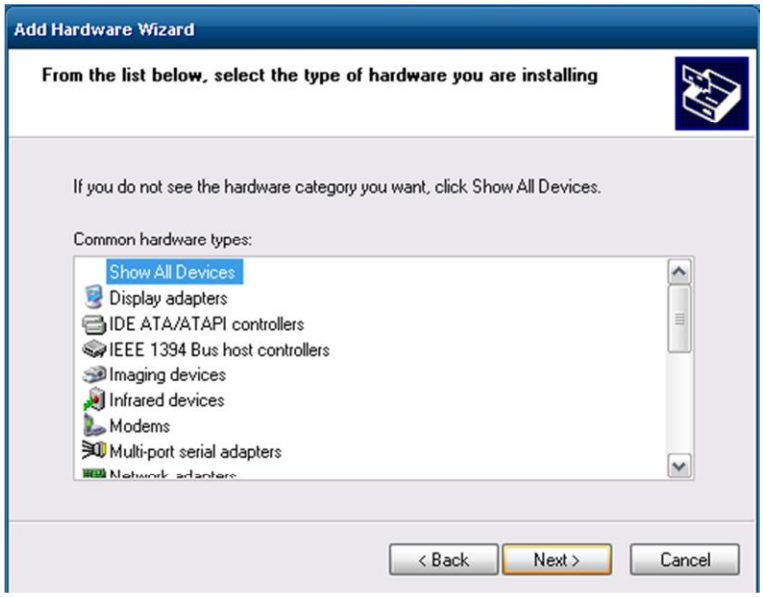
Before the BIOS-API can be used, the driver has to be added to the device manager. The necessary steps to achieve this are shown in the following screen shots.



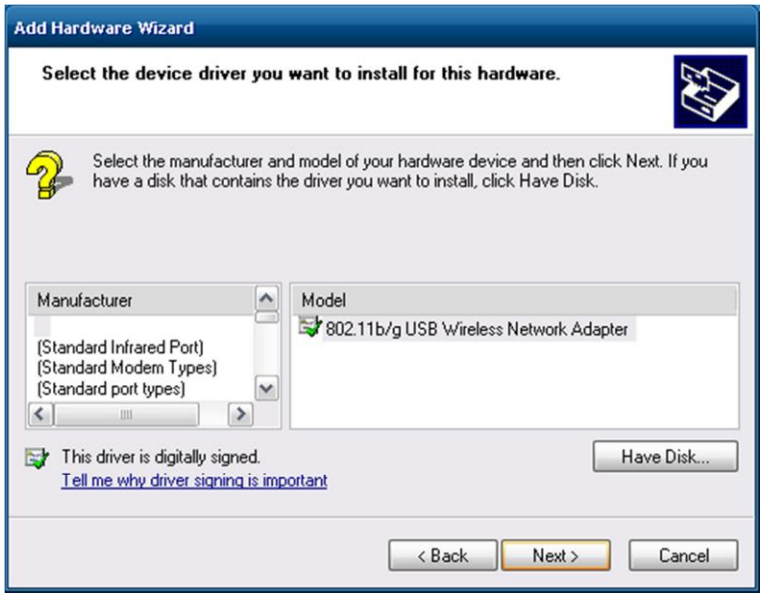

Add Hardware Wizard
Is the hardware connected?
Have you already connected this hardware to your computer?
Yes, I have already connected the hardware
No, I have not added the hardware yet
< Back
Next >
Cancel

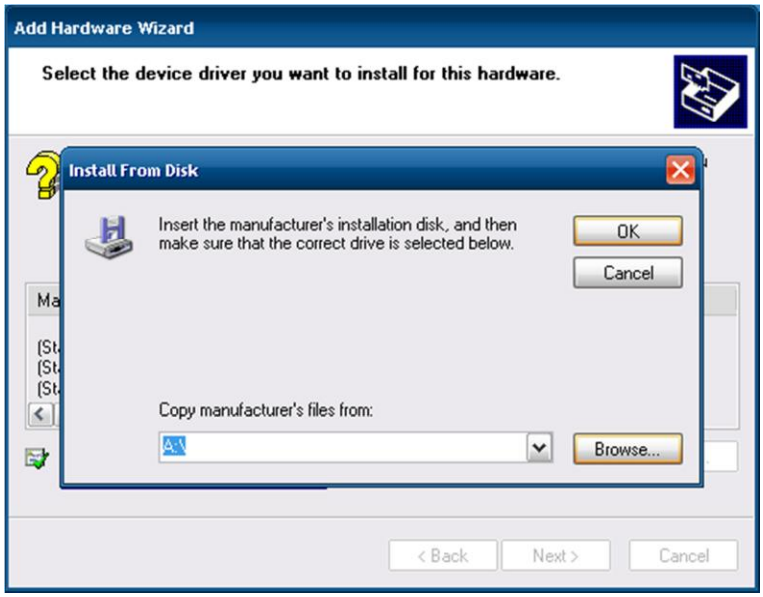

Add Hardware Wizard
The following hardware is already installed on your computer
From the list below, select an installed hardware device, then click Next to check properties or troubleshoot a problem you might be having.
To add hardware not shown in the list, click "Add a new hardware device."
Installed hardware:
USB Root Hub
USB Root Hub
USB Root Hub
USB Root Hub
Generic USB Hub
Add a new hardware device
< Back
Next >
Cancel

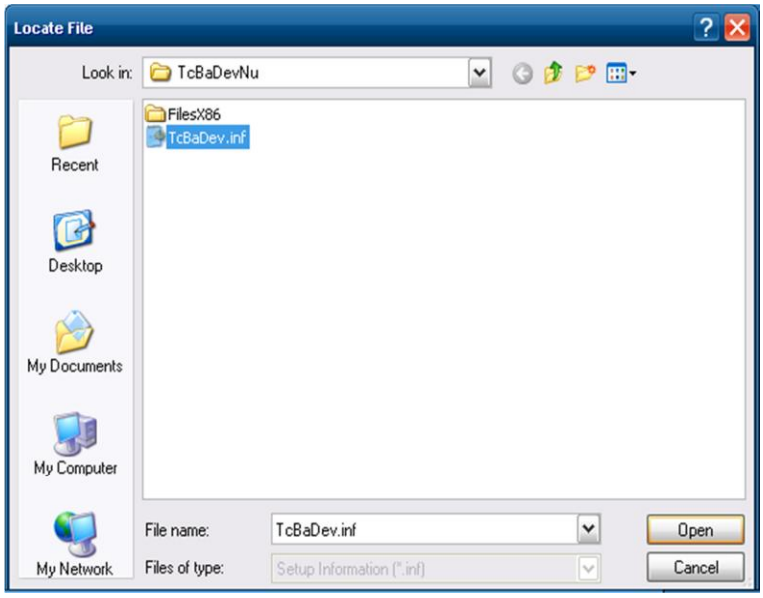

Add Hardware Wizard
The wizard can help you install other hardware
The wizard can search for other hardware and automatically install it for you. Or, if you know exactly which hardware model you want to install, you can select it from a list.
What do you want the wizard to do?
Search for and install the hardware automatically (Recommended)
Install the hardware that I manually select from a list (Advanced)
< Back
Next >
Cancel

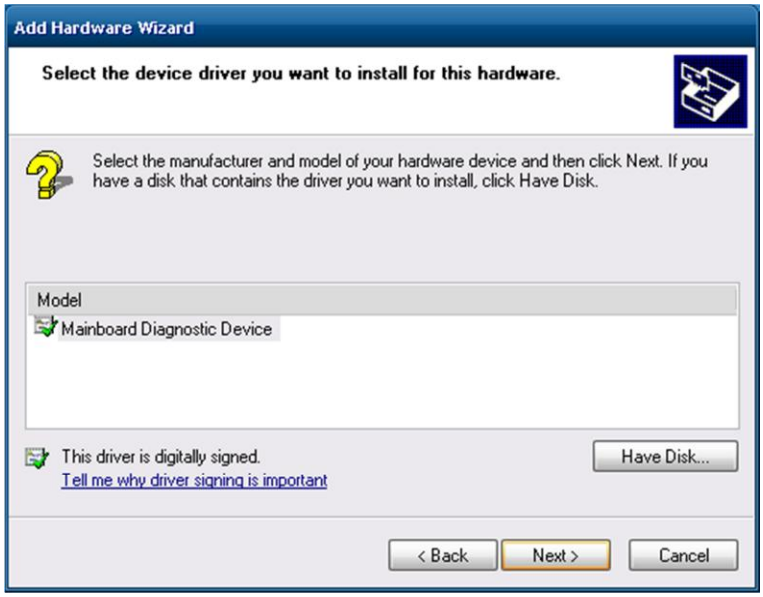


Click
„Have Disk…“

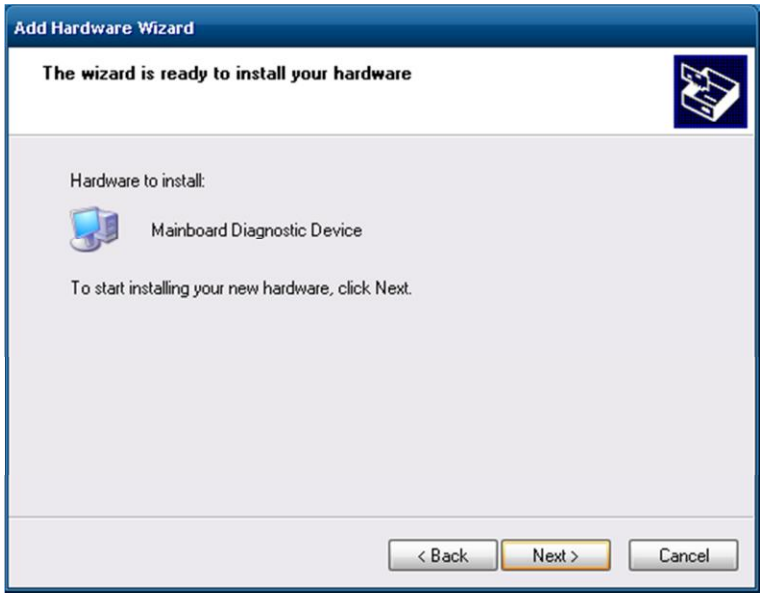


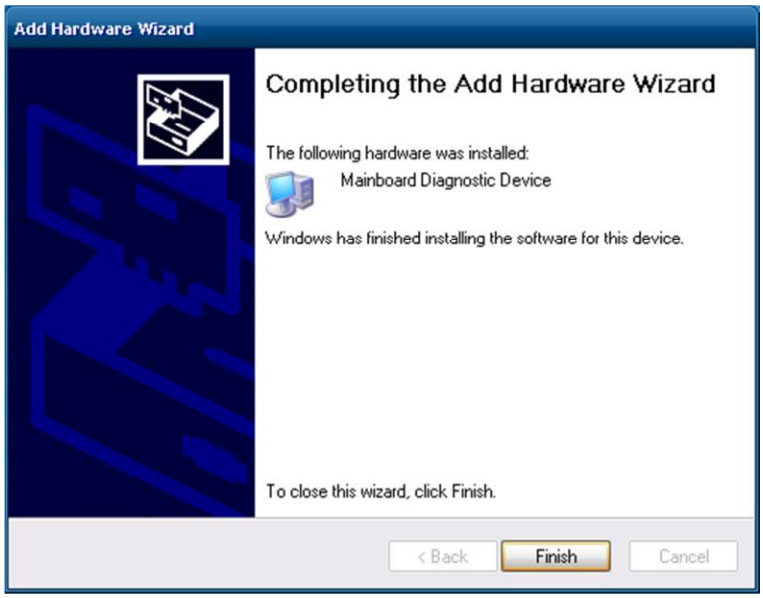


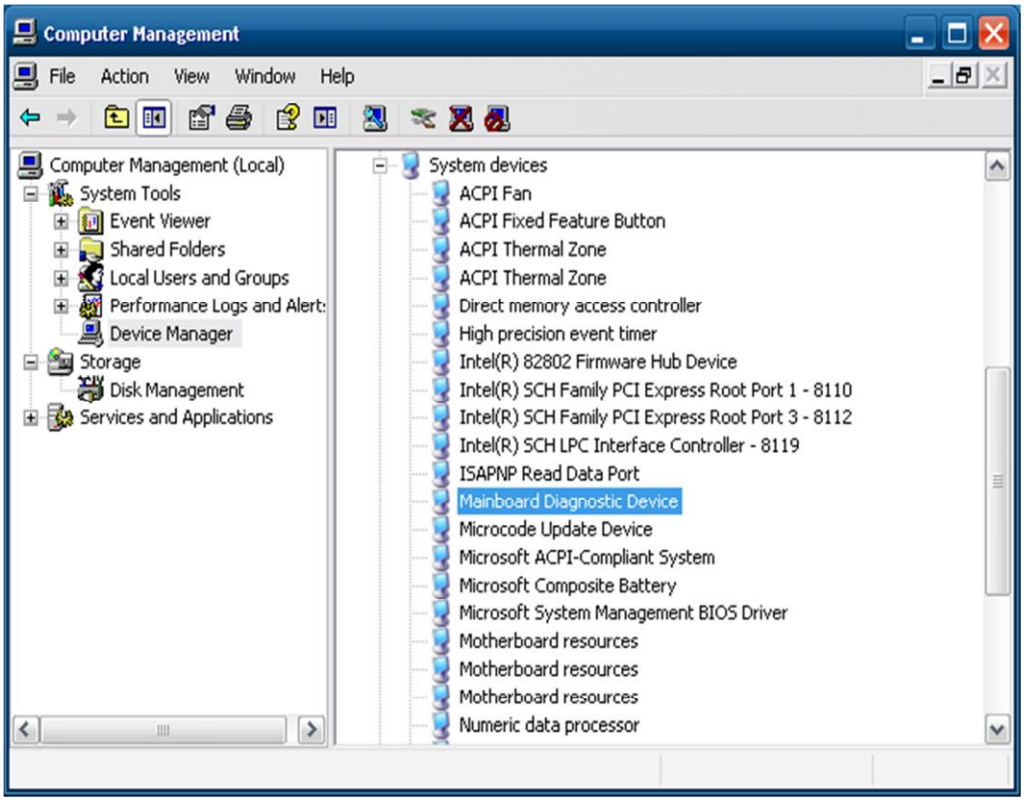

Locate File
Look in:
TcBaDevNu
FilesX86
TcBaDev.inf
Recent
Desktop
My Documents
My Computer
My Network
File name:
TcBaDev.inf
Files of type:
Setup Information (*.inf)
Open
Cancel


Add Hardware Wizard
Select the device driver you want to install for this hardware.
Select the manufacturer and model of your hardware device and then click Next. If you have a disk that contains the driver you want to install, click Have Disk.
Model
Mainboard Diagnostic Device
This driver is digitally signed.
Tell me why driver signing is important
Have Disk...
< Back
Next >
Cancel

Click „Next“


Add Hardware Wizard
The wizard is ready to install your hardware
Hardware to install:
Mainboard Diagnostic Device
To start installing your new hardware, click Next.
< Back
Next >
Cancel


Add Hardware Wizard
Completing the Add Hardware Wizard
The following hardware was installed:
Mainboard Diagnostic Device
Windows has finished installing the software for this device.
To close this wizard, click Finish.
< Back
Finish
Cancel


Computer Management
File
Action
View
Window
Help
Computer Management (Local)
System Tools
Event Viewer
Shared Folders
Local Users and Groups
Performance Logs and Alert
Device Manager
Storage
Disk Management
Services and Applications
System devices
ACPI Fan
ACPI Fixed Feature Button
ACPI Thermal Zone
ACPI Thermal Zone
Direct memory access controller
High precision event timer
Intel(R) 82802 Firmware Hub Device
Intel(R) SCH Family PCI Express Root Port 1 - 8110
Intel(R) SCH Family PCI Express Root Port 3 - 8112
Intel(R) SCH LPC Interface Controller - 8119
ISAPNP Read Data Port
Mainboard Diagnostic Device
Microcode Update Device
Microsoft ACPI-Compliant System
Microsoft Composite Battery
Microsoft System Management BIOS Driver
Motherboard resources
Motherboard resources
Motherboard resources
Numeric data processor

4 Function BaDeviceIoControl

Using the Windows® driver, all calls to the BIOS-API are accomplished by using the function “BaDeviceIoControl”, which has the following structure (for code examples, see annex):

```
unsigned long _stdcall BaDeviceIoControl(     unsigned long nIndexGroup,
                                               unsigned long nIndexOffset,
                                               void* pInBuffer,
                                               unsigned long nInBufferSize,
                                               void* pOutBuffer,
                                               unsigned long nOutBufferSize,
                                               unsigned long * pBytesReturned
                                          PBADEVICE_MODE pMode /* NULL => Optional
*/ );
```

nIndexGroup: Index-Group.

nIndexOffset: Index-Offset.

pInBuffer: Pointer to a buffer that contains the data required to perform the operation. This parameter can be NULL if the *nIndexGroup and nIndexOffset* parameter specifies an operation that does not require input data.

nInBufferSize: Size, in bytes, of the buffer pointed to by *pInBuffer*.

pOutBuffer: Pointer to a buffer that receives the operation’s output data. This parameter can be NULL if the *nIndexGroup and nIndexOffset* parameter specifies an operation that does not produce output data.

nOutBufferSize: Size, in bytes, of the buffer pointed to by *pOutBuffer*

pBytesReturned: Pointer to a variable that receives the size, in bytes, of the data stored into the buffer pointed to by *pOutBuffer*.

The *pBytesReturned* parameter cannot be NULL. Even when an operation produces no output data, and *pOutBuffer* can be NULL, the **BaDeviceIoControl** function makes use of the variable pointed to by *pBytesReturned*.

If the output buffer *pOutBuffer* is too small to return the data, then the call fails, **BaDeviceIoControl** returns the error code ERROR_INSUFFICIENT_BUFFER and the *pBytesReturned* variable returns the required *pOutBuffer* buffer size. The application should call **BaDeviceIoControl** again with the same operation, specifying a new pOutBuffer size.

If the requested *nIndexGroup*/*nIndexOffset* operaton is not supported, then the call fails and the **BaDeviceIoControl** returns the error code: ERROR_SERVICE_NOT_SUPPORTED

pMode: Optional parameter (if NULL => unused). This parameter can be used to change the Thread-Affinity (processor core) prior to the BIOS-API call. E.g. used to read the CPU-temperature from every CPU-Core.

4.1 Return value

Zero indicates success. Nonzero indicates failure.

4.2 Requirements

Header	TcBaDevApi.h (include TcBaDevDef.h)
Library	TcBaDevApi.lib
DLL	TcBaDevApi.dll

5 Sensor Communication

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00002000	0x00000000	NULL	0	PULONG	4	4	Get the max. number of available sensors (entries).
0x00002000	entry index (>= 1) 0x00000001..0x000000FF	NULL	0	Pointer to SENSORINFO structure variable (see below)	56	56	Get data of one sensor entry.

```
typedef struct {
        short           value; // -32767 .. +32767
        unsigned short  status; // status of value: 0x8000 == unused, 0x4000 ==
relative value (dig. temp.)
        unsigned long   rsv; // reserved
}INFOVALUE;
typedef struct {
        PROBETYPE       eType;          // sensor type
        LOCATIONTYPE    eLocation;      // sensor location
        INFOVALUE       readVal;                // current value
        INFOVALUE       nomVal;         // nominal value
        INFOVALUE       minVal;         // min. value
        INFOVALUE       maxVal;         // max. value
        DWORD           rsrv;
        char            desc[12];       // description of sensor as ASCII string
(includes null termination)
 SENSORINFO, *PSENSORINFO;
```

eType: Sensor type:

- 0 = Unknown;
- 1 = Temperature probe [°C];
- 2 = Voltage probe [0.01V];
- 3 = Fan tachometer input [RPM];
- 4 = Case intrusion input [0 = Closed, 1 = Open];
- 5 = Current probe [0.001A];

eLocation: Sensor location. Temperature/Voltage probe or fan locations:

- 0 = Unknown;
- 1 = Other;

- 2 = Processor;
- 3 = Disk;
- 4 = System Management Module;
- 5 = Motherboard;
- 6 = Memory Module;
- 7 = Power Supply;
- 8 = Add-In Card;
- 9 = Front Panel Board;
- 10 = Back Panel Board,
- 11 = Periphery;
- 12 = Chassis;
- 13 = Battery
- 14 = UPS
- 15 = Graphics Board;
- 16 = SuperIO
- 17 = Chipset

readVal: Read value (current temperature, fan speed, voltage...);

nomVal: Nominal value;

minVal: Minimal value;

maxVal: Maximal value;

rsrv: Reserved;

desc: Zero terminated string with additional short description, i.e.: "Vtt" or "Standby", "Vmem"

6 GPIO Services

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00003000	0x00000000	NULL	0	Byte	1	1	Read GPIO0-Input
0x00003000	0x00000001	NULL	0	Byte	1	1	Read GPIO1-Input
0x00003000	0x00000002	NULL	0	Byte	1	1	Read GPIO0-Output
0x00003000	0x00000003	NULL	0	Byte	1	1	Read GPIO1-Output
0x00003000	0x00000004	NULL	0	Byte	1	1	Read GPIO0-Mask
0x00003000	0x00000005	NULL	0	Byte	1	1	Read GPIO1-Mask
0x00003000	0x00000006	Byte	1	NULL	0	0	Write GPIO0-Output
0x00003000	0x00000007	Byte	1	NULL	0	0	Write GPIO1-Output
0x00003000	0x00000008	Byte	1	NULL	0	0	Write GPIO0-Mask
0x00003000	0x00000009	Byte	1	NULL	0	0	Write GPIO1-Mask

7 Micro Controller Communication

7.1 PWRCTRL services

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00004000	0x00000000	NULL	0	Byte[3]	3	3	Get the Bootloader Revision (xx.yy-zz)
0x00004000	0x00000001	NULL	0	Byte[3]	3	3	Get the Firmware Revision (xx.yy-zz)
0x00004000	0x00000002	NULL	0	Byte	1	1	Get Device ID returns the PIC-Type 2x000x_xxxx (PIC18F46K20), 2x100x_xxxx (PIC 18F44K20)
0x00004000	0x00000003	NULL	0	DWORD	4	4	Get Operation Time (minutes since production time)
0x00004000	0x00000004	NULL	0	Byte[2]	2	2	Get min/max Temp (-126..127[°C])
0x00004000	0x00000005	NULL	0	Byte[2]	2	2	Get min/max Voltage (0…255 [x 100mV])
0x00004000	0x00000006	NULL	0	Char[16]	17	17	Get serial number. Null terminated string of max. length <= 16 byte (inclusive null termination).
0x00004000	0x00000007	NULL	0	WORD	2	2	Get boot counter
0x00004000	0x00000008	NULL	0	Byte[2]	2	2	Get production date [WW].[JJ]
0x00004000	0x00000009	NULL	0	Byte	1	1	Get position of the PwrCtrl. Verifies if the PwCtrl is runing in Firmware or Bootloader <0x00> = Firmware <0xFF> = Bootloader
0x00004000	0x0000000A	NULL	0	Byte[3]	3	3	Get Last Shutdown Dump returns the reason of the Last Shutdown Only Supported for CBxx52 Mainboards
0x00004000	0x0000000B	NULL	0	Byte	1	1	Get test count. Number of tests since production.
0x00004000	0x0000000C	NULL	0	Char[6]	7	7	Get test number. Null terminated string of max. length <= 6 byte (inclusive null termiantion)

7.2 S-USV services

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00005000	0x00000000	Byte	1	NULL	0	0	Enable/disable S-UPS <0> = Disable <1> = Enable
0x00005000	0x00000001	NULL	0	Byte	1	1	Get S-UPS status 0x00 Dcin OK 0xAF S-UPS OFF 0xBB DCin Fail 0xEE Balance Error 0xFF µC Power Fail 0xCn Charging (n=(0..A)*10%; n>A=100%) Capacity 0xDn Discharging (n=(0..A)*10%; n>A=100%) Capacity remaining 0xF0 S-UPS not supported 0xF1 S-UPS no communication 0xF2 S-UPS NACK 0xF3 S-UPS Checksum Error 0xF4 S--UPS ID Error
0x00005000	0x00000002	NULL	0	Byte [2]	2	2	Get S-UPS revision [0].[1] == Version.Revision
0x00005000	0x00000003	NULL	0	WORD	2	2	Get S-UPS counter Returns the Number of PWR-Fails
0x00005000	0x00000004	NULL	0	DWORD[3]	12	12	Get S-UPS times Returns the Time from the last three PWR-Fails (operating time counter value, minutes)
0x00005000	0x00000005	Byte	1	NULL	0	0	Set Shutdown with Restart Writes the Behavior on Shutdown 0xFF = Shutdown WITHOUT Reboot 0x01 = Shutdown WITH Neustart Reboot ONCE 0xA1 = Shutdown WITH Reboot
0x00005000	0x00000006	NULL	0	Byte	1	1	Get Shutdown with Restart Get the Behaviorcode and resets the code to 0xFF 0xFF = Shutdown WITHOUT Reboot 0x01 = Shutdown WITH Neustart Reboot ONCE 0xA1 = Shutdown WITH Reboot
0x00005000	0x00000007	NULL	0	Byte	1	1	Get S-UPS Active Count Returns how ofter the System was running under S-UPS. Is resetted

							after reading.
0x00005000	0x00000008	NULL	0	Byte	1	1	Get PwrFail Status Returns the number of PwrFail in the S-UPS
0x00005000	0x00000009	NULL	0	NULL	0	0	Test S-UPS capacitor
0x00005000	0x0000000A	NULL	0	Byte	1	1	Get capacitor test result
0x00005000	0x0000000A0	NULL	0	SUPS_GPIO_INFO	4	4	Get S-UPS-GPIO Address returns the Address, witch is used to determine a PWR-Fail

```
typedef struct TSUps_GpioInfo
{
        unsigned short ioAddr;
        unsigned char Offset;
        unsigned char Params;
        unsigned long rsv;//reserved
}SUPS_GPIO_INFO, *PSUPS_GPIO_INFO;
```

8 Watchdog Communication

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00006000	0x00000000	Byte	1	NULL	0	0	Enables/Triggers the Watchdog <0> = Disables WD <1...255> = Enables WD and sets the WD-Interval (Timbase: Seconds or Minutes, see next command below)
0x00006000	0x00000001	Byte	1	NULL	0	0	Watchdog Config <0> = Timebase: Second <1> = Timebase: Minute
0x00006000	0x00000002	NULL	0	Byte	1	1	Get WD Config
0x00006000	0x00000003	Byte	1	NULL	0	0	Set WD Config
0x00006000	0x00000004	Byte	1	NULL	0	0	activate PwrCtrl IO Watchdog <0> = Compatibility Mode <1> = PwrCtrl IO Watchdog
0x00006000	0x00000005	Byte[2]	2	NULL	0	0	Enables/Triggers the Watchdog with TimeSpan Byte[0] = MaxTime Byte[1] = MinTime
0x00006000	0x00000006	NULL	0	NULL	0	0	IO Retrigger
0x00006000	0x00000007	NULL	0	SUPS_GPIO_INFO	4	4	Get IO address for direct retriggers

9 Other Communication

General BIOS functions

nIndexGroup	nIndexoffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00000000	0x00000000	Null	0	Pointer to BADEVICE_VERSION structure variable	4	4	Returns internal BIOS API version information. To read the version information of the DLL use the BaGetDllVersion function.
0x00000000	0x00000001	Null	0	Byte[16]	16	16	Returns the mainboard platform as null terminated string. i.e. "CBxx52"
0x0000000	0x00000002	Null	0	Pointer to BADEVICE_MBINFO structure variable	12	12	Returns the mainboard information as BADEVICE_MBINFO Structure. Containing Boardname, Board-Revision and Bios Version.
0x0000000	0x00000003	Null	0	Byte	1	1	Returns whether it's a 32Bit platform or a 64Bit platform <0x00> = 32Bit API <0x01> = 64Bit API

```
typedef struct  TBaDevice_MBInfo
{
        char                    MBName[8];
        unsigned char           MBRevision;
        unsigned char           biosMajVersion; // Binary Coded Decimal
        unsigned char           biosMinVersion; // Binary Coded Decimal
        unsigned char           reserved;
}BADEVICE_MBINFO, *PBADEVICE_MBINFO;;
```

User Data Area

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00007000	0x00000000	NULL	0	Byte[256]	256	256	Read UserData
0x00007000	0x00000001	Byte[128]	128	NULL	0	0	Write UserData
0x00007000	0x00000002 (available in API rev. 2.0 and newer)	Byte	1	Byte	1	1	Read Byte Inbuffer=Byte offset
0x00007000	0x00000003 (available in API rev. 2.0 and newer)	Byte[2]	2	NULL	0	0	Write Byte Byte[0] = Byte offset Byte[1] = Value

LED Services

nIndexGroup	nIndexOffset	pInBuffer	nInBufferSize	pOutBuffer	nOutBufferSize	pBytesReturned	Description
0x00008000	0x00000000	BYTE	1	0	0	0	Set TC-LED 0 = OFF 1 = RED 2 = BLUE 3 = GREEN
0x00008000	0x00000001	BYTE	1	0	0	0	Set USER-LED 0 = OFF 1 = RED 2 = BLUE 3 = GREEN

10 Driver Design

Users who want to create their own driver, might find the following information useful.

10.1 BIOS API entry point

In the upper 4 GB of physical memory, at a 0x10 offset, the string „BBIOSAPI" (on 32bit systems) or the string "BBAPIX64" (on 64bit systems) is stored. Immediately following this 8-character string, the entry point of the BIOS API function is stored as a 32-Bit-Offset value.

```
////////////////////////////////////////////////////////////
// BIOS API call function prototype
typedef ULONG (_stdcall *PFN_BBIOSAPI_CALL)( ULONG nIndexGroup,
                                     ULONG nIndexOffset,
                                     PVOID pInBuffer,
                                     ULONG nInBufferSize,
                                     PVOID pOutBuffer,
                                     ULONG nOutBufferSize,
                                     PULONG pBytesReturned );
```

Offset	Name	Length	Description
0x00000000	Anchor String	8 BYTE	"BBIOSAPI" or "BBAPIX64" specified as 8 ASCII characters
0x00000008	API function entry point	DWORD	The 32-bit offset of the BIOS API function entry point.

10.1.1 Notes

The __**stdcall** calling convention is used to call Win32 API functions. **The callee cleans the stack.** The result is stored in EAX (the destination) (DWORD).
Functions that use this calling convention require a function prototype. The following list shows the implementation of this calling convention.

Element	Implementation
Argument-passing order	Right to left.
Argument-passing convention	By value, unless a pointer or reference type is passed.
Stack-maintenance responsibility	Called function pops its own arguments from the stack.
Name-decoration convention	An underscore (_) is prefixed to the name. The name is followed by the at sign (@) followed by the number of bytes (in decimal) in the argument list. Therefore, the function declared as `int func( int a, double b )` is decorated as follows: `_func@12`

Case-translation convention	None

Instruction RET
Purpose Return from a procedure call
Examples

```
RET
RET  8
```

Description The RET instruction returns from a procedure call. It simply pops whatever value is currently at [ESP] into the EIP (instruction pointer) register. The "RET XX" form does the same thing, and then adds XX to the ESP value. This is how __stdcall procedures clear parameters off the stack before returning to their caller. (Most Win32® APIs are __stdcall based.) By dividing the number of cleared bytes by four (the size of a DWORD), you can usually figure out how many parameters a procedure takes. For instance, a procedure that returns with a "RET 8" instruction takes two parameters.

Functions that return an integer or pointer value usually return the value in the EAX register. By examining what's in EAX before executing the RET instruction, you can see the function's return value.

NOTE

Microsoft® Windows® CE only supports the _cdecl calling convention. Use assembler language to implement the _stdcall function call.

10.1.2 Notes for 64bit Systems:

On 64bit Systems the Microsoft x64 calling convention is used to call the x64 BApi function. **The caller cleans the stack.** The result is stored in EAX (the destination) (DWORD).

The Anchor String for the x64 BApi is "BBAPIX64".

Element	**Implementation**
Arguments in Register	RCX,RDX,R8,R9
Argument-passing Stack order	Right to left.
Stack-maintenance responsibility	The Caller cleans the stack.

Example:

```
//////////////////////////////////////////////////////////////
const ULONG BBIOSAPI_SIGNATURE_SEARCH_AREA = 0x1FFFFF;
const PHYSICAL_ADDRESS BBIOSAPI_SIGNATURE_PHYS_START_ADDR =
{0xFFE00000,0x00000000};
#ifdef _WIN64
	const BYTE BBIOSAPI_SIGNATURE[8] = {'B','B','A','P','I','X','6','4'};
	const DWORD BBIOSAPI_SEARCHBSTR = 0x50414242; // first four byte of the
API-String "BBAPIX64"
#else
	const BYTE BBIOSAPI_SIGNATURE[8] = {'B','B','I','O','S','A','P','I'};
	const DWORD BBIOSAPI_SEARCHBSTR = 0x4F494242; // first four byte of the
API-String "BBIOSAPI"
#endif

//////////////////////////////////////////////////////////////
typedef struct TBBIOSAPI_SEARCH_AREA
{
	BYTE Data[BBIOSAPI_SIGNATURE_SEARCH_AREA];
}BBIOSAPI_SEARCH_AREA, *PBBIOSAPI_SEARCH_AREA;

//////////////////////////////////////////////////////////////
//
typedef struct TBBIOSAPI_ENTRY
{
	BYTE anchStr[8]; // Anchor string "BBIOSAPI"
	ULONG entryPtr; // API function entry point
}BBIOSAPI_ENTRY, *PBBIOSAPI_ENTRY;

//////////////////////////////////////////////////////////////
// BIOS API call function prototype
typedef ULONG (_stdcall *PFN_BBIOSAPI_CALL)( ULONG nIndexGroup,
                                     ULONG nIndexOffset,
                                     PVOID pInBuffer,
                                     ULONG nInBufferSize,
                                     PVOID pOutBuffer,
                                     ULONG nOutBufferSize,
                                     PULONG pBytesReturned );
```

```
PBBIOSAPI_SEARCH_AREA m_pBios = NULL;
PFN_BBIOSAPI_CALL m_pProcAddr = NULL;
m_pBios =
(PBBIOSAPI_SEARCH_AREA)OsMapPhysAddr((PVOID)BBIOSAPI_SIGNATURE_PHYS_START_ADD
R,

BBIOSAPI_SIGNATURE_SEARCH_AREA );
if (m_pBios==NULL)
    return FALSE;

// search the anchor string
BOOL bFound = FALSE;
for (int i = 0; i< BBIOSAPI_SIGNATURE_SEARCH_AREA; i+=0x10) //aligned search
{
    if ( memcmp( &m_pBios->Data[i], BBIOSAPI_SIGNATURE,
sizeof(BBIOSAPI_SIGNATURE)) == 0)
    {
        BBIOSAPI_ENTRY entry;
        memset(&entry, 0, sizeof(entry));
        memcpy( &entry, &m_pBios->Data[i], sizeof(entry));
        if (entry.entryPtr != NULL)
        {
            DWORD physOffset = (i + entry.entryPtr);
            DWORD virtOffset = ((DWORD)m_pBios) + physOffset;

            m_pProcAddr = (PFN_BBIOSAPI_CALL)virtOffset;

            TRACE(("BBIOSAPI signature found at offset: 0x%X\n, entry
point<->signature offset:0x%.8X, \nphys/virt offset:0x%X, virtual API call
address:0x%X\n",
            BBIOSAPI_SIGNATURE_PHYS_START_ADDR + i,
            entry.entryPtr,
            physOffset,
            virtOffset ) );

            bFound = TRUE;
            break;
        }
    }
}

if (!bFound)
{
    OsUnMapPhysAddr( m_pBios, BBIOSAPI_SIGNATURE_SEARCH_AREA );
    m_pBios = NULL;
}
```

10.2 BIOS API Error Codes

Error code	Name	Description
0x0000	BIOSAPIERR_NOERR	No error
0x701	BIOSAPI_SRVNOTSUPP	Service/function not supported
0x702	BIOSAPI_INVALIDGRP	Invalid index group
0x703	BIOSAPI_INVALIDOFFSET	Invalid index offset
0x705	BIOSAPI_INVALIDSIZE	Invalid data size parameter
0x706	BIOSAPI_INVALIDDATA	Invalid data value
0x707	BIOSAPI_NOTREADY	Function/service not ready
0x70B	BIOSAPI_INVALIDPARM	Invalid parameter

I Annex: Code Example 1

The following code is an example of how to access the user space in EEPROM.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;

namespace Read_UserData
{
     class Program
     {
          //Declare BiosAPI MainFunction
          [DllImport("TcBaDevApi.dll", EntryPoint = "BaDeviceIoControl", CallingConvention =
CallingConvention.StdCall)]
          private static extern UInt32 BaDeviceIoControl(UInt32 nIndexGroup,
                                             UInt32 nIndexOffset,
                                             [MarshalAs(UnmanagedType.LPArray)] byte[] pInBuffer,
                                             UInt32 nInBufferSize,
                                             [MarshalAs(UnmanagedType.LPArray)] byte[] pOutBuffer,
                                             UInt32 nOutBufferSize,
                                             ref UInt32 pBytesReturned,
                                             UInt32 nModifier);

          const UInt32 cbUserData = 128;
          const UInt32 IGrpUserData = 0x7000;
          const UInt32 IOffReadUserData = 0x0;
          const UInt32 IOffWriteUserData = 0x1;

          static int Main(string[] args)
          {
               if (args.Length>=1 && args[0] == "/w")
                    return WriteRandomData();
               else
                    return ReadUserData();
          }
          static int ReadUserData()
          {
               UInt32 iGrp = IGrpUserData;
               UInt32 iOff = IOffReadUserData;
               Byte[] bIn = null;
               Byte[] bOut = new Byte[cbUserData];
               UInt32 inBufferSize = 0;
               UInt32 outBufferSize = cbUserData;
               UInt32 bCount = 0;

               UInt32 retVal = BaDeviceIoControl(iGrp, iOff, bIn, inBufferSize, bOut, outBufferSize,
ref bCount, 0);
               if (retVal != 0)
               {
                    System.Console.Write("Reading UserData failed! Error=" + retVal.ToString("X4"));
                    return (int)retVal;
               }

               System.Console.Write(bOut[0].ToString("X2") + " ");
               for (int i = 1; i < bOut.Length; i++)
               {
                    if (i % 16 != 0)
                    {
                         System.Console.Write(bOut[i].ToString("X2") + " ");
                    }
                    else
                    {
                         System.Console.Write("\r\n" + bOut[i].ToString("X2") + " ");
                    }
               }
               return (int)retVal;
          }


          static int WriteRandomData()
```

```
        {
            UInt32 iGrp = IGrpUserData;
            UInt32 iOff = IOffWriteUserData;
            Byte[] bIn = new Byte[cbUserData];
            Byte[] bOut = null;
            UInt32 inBufferSize = cbUserData;
            UInt32 outBufferSize = 0;
            UInt32 bCount = 0;

            Random rnd1 = new Random();

            rnd1.NextBytes(bIn);

            UInt32 retVal = (int)BaDeviceIoControl(iGrp, iOff, bIn, inBufferSize, bOut,
outBufferSize, ref bCount, 0);
            if (retVal != 0)
                System.Console.Write("Reading UserData failed! Error=" + retVal.ToString("X4"));
            else
                System.Console.Write("Writing UserData: succsess!");

            return (int)retVal;
        }
    }
}
```

II Annex: Code Example 2

The following C# code is an example of how to use the BIOS API to control GPIO signals.

```
using System;
using System.Windows.Forms;
using System.Runtime.InteropServices;


namespace BapiGpioSimple
{
  public partial class frmMain : Form
  {
    // load external DLL
    [DllImport("TcBaDevApi.dll", EntryPoint = "BaDeviceIoControl", CallingConvention =
CallingConvention.StdCall)]
         private static extern uint BaDeviceIoControl(uint nIndexGroup,
                                                      UInt32 nIndexOffset,
                                                      [MarshalAs(UnmanagedType.LPArray)]
byte[] pInBuffer,
                                                      UInt32 nInBufferSize,
                                                      [MarshalAs(UnmanagedType.LPArray)]
byte[] pOutBuffer,
                                                      UInt32 nOutBufferSize,
                                                      ref UInt32 pBytesReturned,
                                                      UInt32 nModifier);

         const UInt32 IGRP_GPIO              = 0x00003000;
         const UInt32 IOFF_GPIO_READ_INPUT   = 0x00000000;
         const UInt32 IOFF_GPIO_READ_OUTPUT  = 0x00000002;
         const UInt32 IOFF_GPIO_READ_MASK    = 0x00000004;
         const UInt32 IOFF_GPIO_WRITE_OUTPUT = 0x00000006;
         const UInt32 IOFF_GPIO_WRITE_MASK   = 0x00000008;


    Byte[] m_gpioMask = new Byte[2];
         Byte[] m_gpioInput = new Byte[2];
         Byte[] m_gpioOutput = new Byte[2];

    CheckBox[][] chkInputArr = new CheckBox[2][];
    CheckBox[][] chkValArr = new CheckBox[2][];

    Timer timer;



    public frmMain()
    {
      InitializeComponent();

      chkInputArr[0] = new CheckBox[8];
              chkInputArr[1] = new CheckBox[8];
              chkValArr[0] = new CheckBox[8];
              chkValArr[1] = new CheckBox[8];

              chkInputArr[0][0] = chkInput00;
              chkInputArr[0][1] = chkInput01;
              chkInputArr[0][2] = chkInput02;
              chkInputArr[0][3] = chkInput03;
              chkInputArr[0][4] = chkInput04;
              chkInputArr[0][5] = chkInput05;
              chkInputArr[0][6] = chkInput06;
              chkInputArr[0][7] = chkInput07;

              chkValArr[0][0] = chkVal00;
              chkValArr[0][1] = chkVal01;
              chkValArr[0][2] = chkVal02;
              chkValArr[0][3] = chkVal03;
              chkValArr[0][4] = chkVal04;
              chkValArr[0][5] = chkVal05;
              chkValArr[0][6] = chkVal06;
              chkValArr[0][7] = chkVal07;

              chkInputArr[1][0] = chkInput10;
```

```
            chkInputArr[1][1] = chkInput11;
            chkInputArr[1][2] = chkInput12;
            chkInputArr[1][3] = chkInput13;
            chkInputArr[1][4] = chkInput14;
            chkInputArr[1][5] = chkInput15;
            chkInputArr[1][6] = chkInput16;
            chkInputArr[1][7] = chkInput17;

            chkValArr[1][0] = chkVal10;
            chkValArr[1][1] = chkVal11;
            chkValArr[1][2] = chkVal12;
            chkValArr[1][3] = chkVal13;
            chkValArr[1][4] = chkVal14;
            chkValArr[1][5] = chkVal15;
            chkValArr[1][6] = chkVal16;
            chkValArr[1][7] = chkVal17;

    }

    private void frmMain_Load(object sender, EventArgs e)
    {
      try
            {
        // read out ports 0 and 1
        refreshGpio(0);
                refreshGpio(1);

        // refresh GUI
        updateChkBoxes(0);
        updateChkBoxes(1);

        // set interval for probing of port states
        timer = new Timer();
        timer.Interval = 500;
        timer.Tick += new EventHandler(timer_Tick);
        timer.Enabled = true;
            }
            catch (BiosAPIExeption err)
            {
                if (err.ErrorID == (int)TCBADEV_Error.TCBADEV_ERROR_SRVNOTSUPP)
                {
                    MessageBox.Show(this, "GPIO-Service not supported", this.Text + " Error");
                }
                else
                {
                    MessageBox.Show(this, "An Error occured: 0x" + err.ErrorID.ToString("X8"),
this.Text + " Error");
                }
                this.Close();
            }
    }

    // periodical probing of port states
    // Note! GPIO interrupts are not forwarded automatically.
    //       The ports need to be actively probed.
    void timer_Tick(object sender, EventArgs e)
    {
      try
      {
        updateChkBoxes(0);
        updateChkBoxes(1);
      }
      catch(Exception)
      {

      }
    }

    // refresh GUI
    private void updateChkBoxes(UInt32 ByteNo)
         {
            refreshGpio(ByteNo);
            for (UInt32 BitNo = 0; BitNo < 8; BitNo++)
            {
                chkValArr[ByteNo][BitNo].Enabled = !(chkInputArr[ByteNo][BitNo].Checked =
isGpioInput(ByteNo, BitNo));
                chkValArr[ByteNo][BitNo].Checked = isGpioEnabled(ByteNo, BitNo);
```

```
            }
        }

    // User changed port config (Input / Output)
    private void chkInputMaskChanged(object sender, EventArgs e)
        {
            UInt32 ChkTagNo = UInt32.Parse(((CheckBox)sender).Tag.ToString());
            UInt32 ByteNo = ChkTagNo/8;
            UInt32 BitNo = ChkTagNo%8 ;
            inputMaskChanged(ByteNo, BitNo);
            updateChkBoxes(ByteNo);
        }

    // User switched output on or off
    private void chkOutputValChanged(object sender, EventArgs e)
        {
            UInt32 ChkTagNo = UInt32.Parse(((CheckBox)sender).Tag.ToString());
            UInt32 ByteNo = ChkTagNo / 8;
            UInt32 BitNo = ChkTagNo % 8;
            outputValChanged(ByteNo, BitNo);
            updateChkBoxes(ByteNo);
        }

    // Event! User changed port config
    public void inputMaskChanged(UInt32 ByteNo, UInt32 BitNo)
        {
            Byte Mask = 1;
            Mask = (Byte)((int)Mask << (int)BitNo);
            m_gpioMask[ByteNo] ^= Mask;

            setGpioParam(IOFF_GPIO_WRITE_MASK+ ByteNo, m_gpioMask[ByteNo]);
        }

    // Event! User switched output on or off
        public void outputValChanged(UInt32 ByteNo, UInt32 BitNo)
        {
            Byte Mask = 1;
            Mask = (Byte)((int)Mask << (int)BitNo);
            m_gpioOutput[ByteNo] ^= Mask;

            setGpioParam(IOFF_GPIO_WRITE_OUTPUT + ByteNo, m_gpioOutput[ByteNo]);
        }

    // Read current configuration and state, then store that information in registers
        public void refreshGpio(UInt32 i)
        {
            m_gpioMask[i] = getGpioParam(IOFF_GPIO_READ_MASK + i);
            m_gpioInput[i] = getGpioParam(IOFF_GPIO_READ_INPUT + i);
            m_gpioOutput[i] = getGpioParam(IOFF_GPIO_READ_OUTPUT + i);
        }

    // Is port enabled or not?
        public bool isGpioEnabled(UInt32 i, UInt32 j)
        {
            Byte o = (Byte)(m_gpioInput[i] | m_gpioOutput[i]);

            UInt32 Mask = 1;
            Mask = (UInt32)((int)Mask << (int)j);
            return (o & Mask) != 0;
        }

    // Is port configured for input or output?
        public bool isGpioInput(UInt32 i, UInt32 j)
        {
            Byte o = m_gpioMask[i];

            UInt32 Mask = 1;
            Mask = (UInt32)((int)Mask << (int)j);
            return (o & Mask) != 0;
        }

    // Driver interface for read access to the port
        private Byte getGpioParam(UInt32 param)
        {
            Byte[] bIn = null;
            Byte[] bOut = new Byte[1];
            UInt32 bCount = 0;
```

```
            Int32 err = 0;

            err = (Int32)BaDeviceIoControl(IGRP_GPIO, param, bIn, 0, bOut, 1, ref bCount, 0);
            if (err != (int)TCBADEV_Error.TCBADEV_ERROR_NONE)
            {
                throw new BiosAPIExeption(err);
            }
            return bOut[0];
        }

    // Driver interface for write access to the port
        private void setGpioParam(UInt32 param, Byte val)
        {
            Byte[] bIn = new Byte[1];
            Byte[] bOut = null;
            UInt32 bCount = 0;
            Int32 err = 0;

            bIn[0] = val;

            err = (Int32)BaDeviceIoControl(IGRP_GPIO, param, bIn, 1, bOut, 0, ref bCount, 0);
            if (err != (int)TCBADEV_Error.TCBADEV_ERROR_NONE)
            {
                throw new BiosAPIExeption(err);
            }
        }

  }


  public enum TCBADEV_Error
    {
        TCBADEV_ERROR_OFFSET = 0x20000000,                                          //BIOS api
error offset
        TCBADEV_ERROR_NONE = 0x00000000,                                          //No Error
        TCBADEV_ERROR_SRVNOTSUPP = (TCBADEV_ERROR_OFFSET + 0x701),    //Service/Function not
supported
        TCBADEV_ERROR_INVALIDGRP = (TCBADEV_ERROR_OFFSET + 0x702),    //Invalid Index Group
        TCBADEV_ERROR_INVALIDOFFSET = (TCBADEV_ERROR_OFFSET + 0x703), //Invalid Index Offset
        TCBADEV_ERROR_INVALIDSIZE = (TCBADEV_ERROR_OFFSET + 0x705),   //Invalid data size parameter
        TCBADEV_ERROR_INVALIDDATA = (TCBADEV_ERROR_OFFSET + 0x706),   //Invalid data value
        TCBADEV_ERROR_BUSY = (TCBADEV_ERROR_OFFSET + 0x708),          //Function/Service not ready
        TCBADEV_ERROR_INVALIDPARM = (TCBADEV_ERROR_OFFSET + 0x70B)    //Invalid Parameter
    }

    public class BiosAPIExeption : Exception
    {
        private Int32 m_errID;
        public BiosAPIExeption(Int32 errId)   :
base(((TCBADEV_Error)Enum.ToObject(typeof(TCBADEV_Error), errId)).ToString())
        {
            m_errID = errId;
        }

        public Int32 ErrorID
        {
            get
            {
                return m_errID;
            }
        }
    }

}
```