

Programmers Guide

Universal Board Support Package
for Embedded PC-Solutions
MSC API for BIOS functions

UspEpc



MSC Vertriebs GmbH
Design Center Neufahrn
Zeppelinstraße 1a
D-85375 Neufahrn

Version V2.03
24/03/24/2010

Copyright Notice :

Copying of this document, and giving it to others and the use or communication of the contents thereof, are forbidden without express authority. Offenders are liable to the payment of damages. All rights are reserved in the event of the grant of a patent or the registration of a utility model or design.

Weitergabe sowie Vervielfältigung dieser Unterlage, Verwertung und Mitteilung ihres Inhalts nicht gestattet, soweit nicht ausdrücklich zugestanden. Zuwiderhandlungen verpflichten zu Schadenersatz. Alle Rechte für den Fall einer Patenterteilung oder Gebrauchsmuster-Eintragung vorbehalten.

1. Contents

1. CONTENTS	2
2. REVISION HISTORY	4
3. INTRODUCTION	5
4. DEFINITION OF THE INTERFACE.....	6
4.1. BOARD SUPPORT PACKAGE	6
4.1.1. <i>Handling Functions</i>	7
4.1.1.1. UspOpen.....	7
4.1.1.2. UspClose	7
4.1.1.3. UspGetDllVersion	7
4.1.1.4. UspGetVersion	7
4.1.1.5. UspGetLastError	8
4.1.2. <i>General Functions</i>	9
4.1.2.1. UspGetBoardInfo	9
4.1.2.2. UspGetCPUInfo	10
4.1.2.3. UspGetBoardCfgSize	11
4.1.2.4. UspGetBoardCfg	11
4.1.2.5. UspSetBoardCfg.....	11
4.1.3. <i>NVRAM Functions</i>	12
4.1.3.1. UspGetNVRamCount.....	12
4.1.3.2. UspGetNVRamInfo	12
4.1.3.3. UspGetNVDeviceInfo	13
4.1.3.4. UspQueryNVBlock	14
4.1.3.5. UspReadNVRam	15
4.1.3.6. UspWriteNVRam	15
4.1.3.7. UspNVDeviceRaw	16
4.1.4. <i>I2C Functions</i>	17
4.1.4.1. UspGetI2CBusCount.....	17
4.1.4.2. UspQueryI2CBus	18
4.1.4.3. UspI2CRead8	19
4.1.4.4. UspI2CWrite8	19
4.1.4.5. UspI2CRead16	20
4.1.4.6. UspI2CWrite16	20
4.1.4.7. UspI2CReadBlock.....	21
4.1.4.8. UspI2CWriteBlock.....	21
4.1.5. <i>Watchdog Functions</i>	22
4.1.5.1. UspWDConfig.....	22
4.1.5.2. UspWDGetStatus	22
4.1.5.3. UspWDTrigger.....	22
4.1.6. <i>VGA Functions</i>	23
4.1.6.1. UspBacklightCtrl.....	23
4.1.6.2. UspGetBacklightValue.....	23
4.1.6.3. UspSetBacklightValue	23
4.1.6.4. UspGetContrastValue.....	23
4.1.6.5. UspSetContrastValue	23
4.1.7. <i>Monitoring Functions</i>	24
4.1.7.1. UspGetThermSensorCount.....	24
4.1.7.2. UspQueryThermSensor	24
4.1.7.3. UspGetThermSensorValue.....	25
4.1.7.4. UspGetVoltSensorCount	26
4.1.7.5. UspQueryVoltSensor.....	26
4.1.7.6. UspGetVoltSensorValue	27
4.1.8. <i>Mobile Extensions</i>	28
4.1.8.1. UspGetSystemState	28
4.1.8.2. UspGetBatteryCount	29
4.1.8.3. UspGetBatteryInfo	29
4.1.8.4. UspGetBatteryState	30

4.1.9.	<i>OEM Functions</i>	31
4.1.9.1.	UspOEMRead	31
4.1.9.2.	UspOEMWrite	32
4.1.9.3.	UspOEMPassThru	32
5.	WIN32 DEFINITIONS	33
6.	WINDOWS CE	33
7.	LINUX	33
8.	ERROR CODES (USPGETLASTERROR)	34
9.	DATA TYPES	35
10.	EEPROM DATA ORGANISATION	35
11.	PRODUCT SUPPORT	36

2. Revision History

Version	Chapter	Modification	Originator
V2.00		initial release	HAH
X2.01a	4.1.7	corrected: DWORD TSensorID	HAH
		corrected: DWORD VSensorID	
	4.1.8	corrected: DWORD *pBatteryCount	
		corrected: DWORD BatID	
	4.1.4.2	corrected UspQueryI2Cbus input parameters and I2CBUSINFO structure definition	
	4.1.2.1	add BIOSTIME definition	
	4.1.9.3	modified UspOemPassThru arguments	
	4.1.2.2	modified size of CPUINFO.CPUString	
		corrected comment to CPUINFO. CPUSpeed	
	4.1.7.5	corrected VSENSORINFO structure description	
X2.01b	4.1.4.x	I2CbusFlags: add block transfer flags Protocol: add block transfer flags	HAH
X2.01c	4.1.2.x	Add board configuration functions UspGetBoardCfgSize, UspGetBoardCfg, UspSetBoardCfg	HAH
V2.01		Release	HAH
V2.02		Release to Website	THEA
V2.03		Added Section 10 – EEPROM org	THEA

3. Introduction

This document describes the MSC API (application program interface) for accessing low-level BIOS functions – called UspEpc.

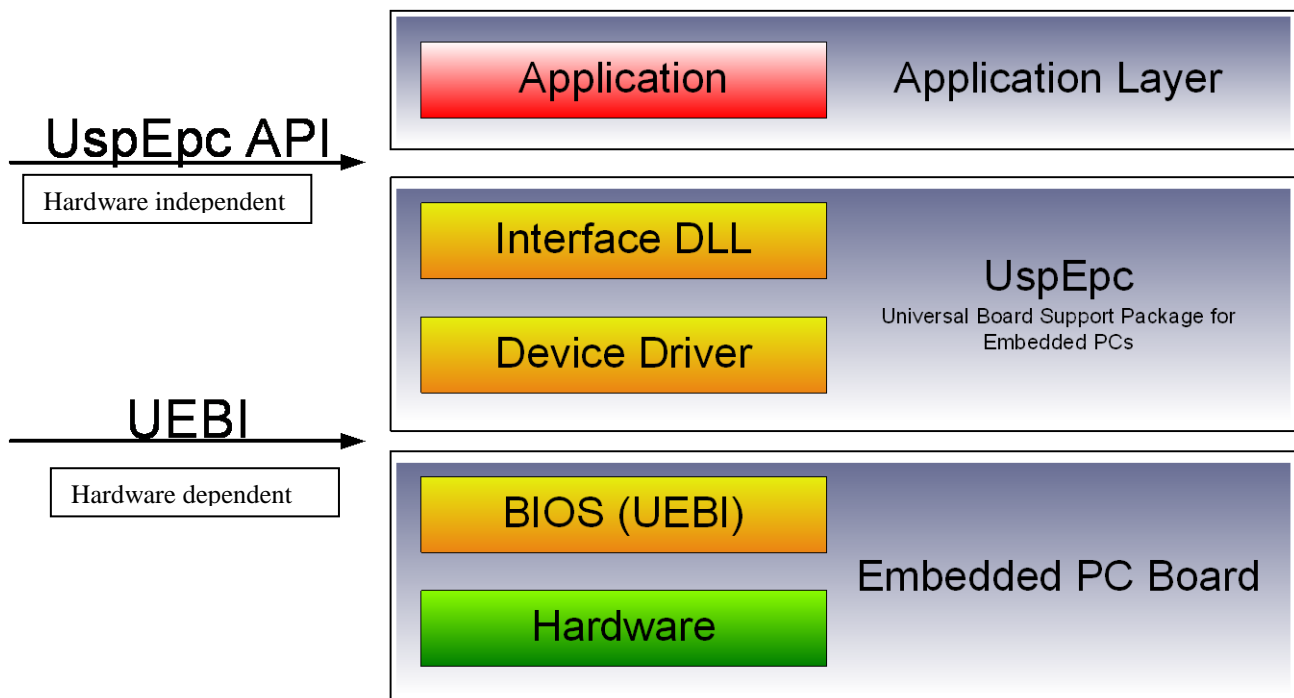
UspEpc defines a universal interface for embedded PC solutions, which standardizes the user access to typical OEM features. Thus customers can develop application programs which make use of low level BIOS features – such as temperature monitoring, fan speed control, setup configuration store/restore etc.

This API specifies the interface for operating systems using the WIN32-API. In addition many other operating systems available on the market can be supported using the same model underlying this interface (see software block diagram). Software packages are available for Windows XP, Windows 2000, Windows Vista, Windows XPe, Windows CE and Linux.

The various software libraries and DLL's are available on request to MSC customers. Please use the contact details included at the end of this document in the Product Support section for such requests.

The BIOS implementation of the low level routines implementing the hardware access is not dependent on a specific operating system. Because of this the operating system dependent software components are basically limited to the adaptation of the MSC UEBI (Universal Embedded BIOS Interface) interface to the particular driver of the corresponding operating system

see block diagram below for more details :



The common interface to the hardware “Universal Embedded Bios Interface“ (UEBI) is independent of the operating system and specified separately.

4. Definition of the Interface

Access to special board functions for embedded PC solutions is abstracted via interface libraries and device drivers.

4.1. *Board Support Package*

The Board Support Package consists of two components – the interface library and the drivers.

An application is assigned to a device driver via the interface library. Parameters are transferred according to standard “C” conventions.

The device driver contains all the functions necessary to access the OEM features of the different embedded hardware solutions via BIOS interface APIs.

Driver functions are never called directly by applications but always use the interface library.

4.1.1. Handling Functions

4.1.1.1. UspOpen

API

BOOL UNIVERSAL UspOpen(void);

Returns: TRUE for correct execution of the function, otherwise FALSE.

With this function the device driver and the DLL are initialized. It must be called once before any other functions are called.

4.1.1.2. UspClose

API

BOOL UNIVERSAL UspClose(void);

Returns: TRUE for correct execution of the function, otherwise FALSE.

This function has to be called before the Win32 application ends in order to clear all resources that might be in use or to terminate all BIOS activities that have been started by the device driver.

4.1.1.3. UspGetDllVersion

API

BOOL UNIVERSAL UspGetDllVersion (DWORD *pVersion);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pVersion: Pointer to the variable where the version number of the interface DLL is stored BCD-encoded.

4.1.1.4. UspGetVersion

API

BOOL UNIVERSAL UspGetVersion (WORD *pVersion);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pVersion: Pointer to a variable where the version number of the UspEpc release is stored hexadecimal encoded. High byte is the major release number, low-byte is the minor release number.
Bit 7 of the high byte, if set indicates a preliminary release.
Example: A return value of 0x0110 represents the UspEpc release V1.10, 0x8200 represents the preliminary release X2.00

4.1.1.5. UspGetLastError

API

BOOL UNIVERSAL UspGetLastError(DWORD *pLastErrorCode, char *lastErrorString, DWORD lastErrorLen);

Returns: always TRUE.

This function returns the error code or the error string of the last call to a UspEpc function, respectively (error code "SUCCESSFULL" or error string "BIOS: SUCCESS" for correct execution of the last call), and can be used to detect the error cause if the return value has been "FALSE".

For lastErrorString = NULL or lastErrorLen = 0 no ErrorString will be returned.

pLastErrorCode:	Pointer to a DWORD variable where the error code of the last UspEpc-function is stored
lastErrorString:	Pointer to a character array where the error string of the last UspEpc-function is stored (maximal lastErrorLen characters).
lastErrorLen:	maximal number of characters stored in lastErrorString. For lastErrorLen = 0 no ErrorString will be returned.

Error codes and corresponding error messages are listed in chapter 8.

4.1.2. General Functions

4.1.2.1. UspGetBoardInfo

API

BOOL UNIVERSAL UspGetBoardInfo(BOARDINFO *pBoardInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pBoardInfo: Pointer to a BOARDINFO structure where information of the CPU board is stored.

The structure BOARDINFO is defined as follows:

```
typedef struct
{
    CHAR HWPlattform [16];
    CHAR SysBIOSVersion [16];
    CHAR BBVersion [16];
    CHAR HWRevison [16];
    CHAR SerialNumber [16];
    DWORD BootCounter;
    DWORD OperatingTime;
    BIOSTIME ManufacturingDate;
    BIOSTIME MaintenanceTime;
} BOARDINFO;
```

HWPlattform	ASCII, ID string of CPU Board	0-terminated
SysBIOSVersion	ASCII, System BIOS Version	0-terminated
BBVersion	ASCII, Bootblock BIOS Version	0-terminated
HWRevison	ASCII, Hardware Board Revision	0-terminated
SerialNumber	ASCII, boardspecific serial number	0-terminated
BootCounter	Binary	
OperatingTime	Binary, time in hours since Power On	contains 0 if not supported
ManufacturingDate	BIOSTIME, date of production	contains 0 if not supported
MaintenanceTime	BIOSTIME, date of last maintenance	contains 0 if not supported

```
typedef struct
{
    BYTE Day;
    BYTE Month;
    WORD Year;
    BYTE Hour;
    BYTE Minute;
} BIOSTIME;
```

Day	BCD	Date / Day
Month	BCD	Date / Month
Year	BCD	Date / Year
Hour	BCD	Time of day / Hour
Minute	BCD	Time of day / Minute

4.1.2.2. UspGetCPUInfo

API

BOOL UNIVERSAL UspGetCPUInfo(CPUINFO *pCPUInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pCPUInfo: Pointer to a CPUINFO structure where information about the CPU is stored.

The structure CPUINFO is defined as follows:

```
typedef struct
{
    DWORD CPUInfoSize;
    BYTE CPUClass;
    BYTE CPUCount;
    WORD CPUSpeed;
    BYTE CPUModel;
    BYTE CPUStepping;
    CHAR CPUString [64];
} CPUINFO;
```

CPUInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
CPUClass	Binary	0x0F – x86 0x10 – ARM Types 0x20 – Hitachi SHx
CPUCount	Binary	number of installed physical CPUs or CPU cores
CPUSpeed	Binary	nominal (max) CPU core frequency in MHz
CPUModel	Binary	CPU family/model ID
CPUStepping	Binary	CPU stepping ID
CPUString	ASCII, CPU specific ID string	0-terminated

4.1.2.3. UspGetBoardCfgSize

API

BOOL UNIVERSAL UspBoardGetBoardCfgSize(DWORD *pCfgSize);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pCfgSize: Pointer to a **DWORD** variable **CfgSize** where the size of the board configuration is stored.

4.1.2.4. UspGetBoardCfg

This function is used to read the BIOS Setup configuration data stored in the backup EEPROM . (Note since the format of this data is BIOS dependant, it cannot be interpreted, only stored and written back to the EEPROM).

API

BOOL UNIVERSAL UspGetBoardCfg (BYTE *pData);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pData: Pointer to a buffer to store the board configuration data

<p>PLEASE NOTE: The calling application is responsible to allocate enough memory to store the configuration data with a size of CfgSize bytes.</p>
--

4.1.2.5. UspSetBoardCfg

API

BOOL UNIVERSAL UspSetBoardCfg (BYTE *pData);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pData: Pointer to a buffer containing the new configuration data to be stored on the board.

<p>PLEASE NOTE: After successful receipt of this command, BIOS sets an internal flag to save the transferred configuration and use it after the next reboot. Other internal configuration data dependent to the board configuration data (like ESCD data) is reset and initialized.</p>
--

4.1.3. NVRAM Functions

General configuration data and user specific system data are permanently stored on non-volatile memories implemented on the CPU board (normally EEPROMs or battery backed CMOS memories). The following functions can be used to access these memories from the application.

4.1.3.1. UspGetNVRamCount

API

BOOL UNIVERSAL UspGetNVRamCount(DWORD *pNVRamCount);

Returns: TRUE for correct execution of the function, otherwise FALSE.

This function returns the number of non-volatile RAMs on the special hardware. This information is necessary to determine how often the function UspGetNVRAMInfo has to be called.

NVRamCount: Pointer to a DWORD variable where the number of available NV RAMs is stored.

4.1.3.2. UspGetNVRamInfo

API

BOOL UNIVERSAL UspGetNVRamInfo(DWORD NVRamIndex, NVRAMINFO *pNVRamInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

NVRamIndex: Index of the non-volatile RAM (0 ... NVRamCount-1)

pNVRamInfo: Pointer to a NVRAMINFO structure where information about the NV RAM is stored.

This functions returns hardware information (type, length, read / write permission) of the non-volatile RAM with the index NVRamIndex. It must be called as many times as it has been determined with NVRAMCount (see above) to get information about all existing NV RAMs.

The structure NVRAMINFO is defined as follows:

```
typedef struct
{
    DWORD    NVRamType;
    DWORD    NVRamSize;
    DWORD    NVRamFlags;
} NVRAMINFO;
```

NVRamType	0x00 – NVRamUserCMOS 0x01 – NVRamUserEEPROM 0x02 – NVRamVGAParameter 0x03 - NVRAMFlash	for Type 0x03 use functions and UspGetNVDeviceInfo and UspQueryNVBlock to get detailed information about the device
NVRamSize	Binary	size in bytes
NVRamFlags	Bit 0 = 1: read access allowed Bit 1 = 1: write access allowed Bit 2-31: reserved	

4.1.3.3. UspGetNVDeviceInfo

API

BOOL UNIVERSAL UspGetNVDeviceInfo (DWORD NVRamIndex, NVDEVICEINFO *pNVDeviceInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

NVRamIndex: Index of the non-volatile RAM (0 ... NVRamCount-1)
pNVDeviceInfo: Pointer to a NVDEVICEINFO structure where information about the device is stored.

The structure NVDEVICEINFO is defined as follows:

```
typedef struct
{
    DWORD    NVDeviceInfoSize;
    WORD     NVManufacturerID;
    WORD     NVDeviceID;
    DWORD    NVBlockCount;
} NVDEVICEINFO;
```

DeviceInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
NVManufacturerID	Binary	Intelligent Identifier (Mfr)
NVDeviceID	Binary	Intelligent Identifier (Device)
NVBlockCount	Binary	number of blocks available in the device

PLEASE NOTE: This function is only valid for block-oriented memories of type **NVRAMFlash**.

4.1.3.4. UspQueryNVBlock

API

BOOL UNIVERSAL UspQueryNVBlock (DWORD NVRamIndex, DWORD BlockNum, NVBLOCKINFO *pNVBlockInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

NVRamIndex: Index of the non-volatile RAM (0 ... NVRamCount-1)
BlockNum: Number of the block to get information (0 ... NVBlockCount-1).
pNVBlockInfo: Pointer to a NVBLOCKINFO structure where information about the selected block is stored.

The structure NVBLOCKINFO is defined as follows:

```
typedef struct
{
    DWORD    NVBlockInfoSize;
    DWORD    NVBlockSize;
    DWORD    NVBlockFlags;
} NVBLOCKINFO;
```

NVBlockInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
NVBlockSize	Binary	total size of the block in bytes
NVBlockFlags	Bit 0 = Lock capable Bit 1 = Locked Bit 2-15: reserved	Bit 0=1: block can be locked by software Bit 1=1: block is locked

PLEASE NOTE: This function is only valid for block-oriented memories of type **NVRAMFlash**.

4.1.3.5. UspReadNVRam

API

BOOL UNIVERSAL UspReadNVRam(DWORD NVRamIndex, DWORD Offset, DWORD Length, BYTE *pBytes, DWORD *pBytesRead);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Reads a number of bytes from the memory with the index NVRamIndex.

NVRamIndex:	Index of the non-volatile RAM
Offset:	Offset inside the memory
Length:	Number of bytes to be read
pBytes:	Buffer where the read bytes are stored
pBytesRead:	Pointer to a DWORD variable where the number of bytes effectively read is returned

4.1.3.6. UspWriteNVRam

API

BOOL UNIVERSAL UspWriteNVRam(DWORD NVRamIndex, DWORD Offset, DWORD Length, BYTE *pBytes, DWORD *pBytesWritten);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Writes a number of bytes to the memory with the index NVRamIndex.

NVRamIndex:	Index of the non-volatile RAMs
Offset:	Offset inside the memory
Length:	Number of bytes to be written
pBytes:	Buffer where the bytes to be written are stored
pBytesWritten:	Pointer to a DWORD variable where the number of bytes effectively written is returned

4.1.3.7. UspNVDeviceRaw

API

BOOL UNIVERSAL UspNVDeviceRaw(DWORD NVRamIndex, DWORD BlockNum, DWORD Cmd);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Raw access to a Block within the specified NVRAM.

NVRamIndex:	Index of the non-volatile RAMs
BlockNum:	0-based block number within the NVRAM
Cmd:	Command to be executed on selected block
	Cmd = 0x0000 0000: Erase block
	Cmd = 0x0000 0001: Unlock Block
	Cmd = 0x0000 0002: Lock Block

PLEASE NOTE:

This function is only valid for block-oriented memories of type **NVRAMFlash**. Typically a block in a memory of type **NVRAMFlash** must be erased before data can be written to a block. The application is responsible to match the requirements of the device.

4.1.4. I2C Functions

With these functions OEM-specific I2C modules can be accessed directly from the applications. An identification (I2CBusID) is used to select different busses.

A list of device addresses used on the CPU board can be found in the hardware manual.

PLEASE NOTE: I2C modules on the CPU board should only be accessed directly in exceptional cases, as this may lead to run-time conflicts with cyclic routines of the system BIOS. The result can be malfunction or a system crash.

4.1.4.1. UspGetI2CBusCount

API

BOOL UNIVERSAL UspGetI2CBusCount(DWORD *pI2CBusCount);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pI2CBusCount: Pointer to a variable, where the actual number of available I2C busses is stored

4.1.4.2. UspQueryI2CBus

API

BOOL UNIVERSAL UspQueryI2CBus(DWORD I2CBusIndex, I2CBUSINFO *pI2CBusInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

I2CBusID: 0-based index of the requested I2C bus. Valid range 0...(pI2CBusCount-1)
pI2CBusInfo: Pointer to a structure to store I2C bus information

The structure I2CBUSINFO is defined as follows:

```
typedef struct
{
    DWORD      I2CBusInfoSize;
    BYTE       I2CBusID;
    BYTE       reserved [3];
    DWORD      I2CBusFlags;
    DWORD      I2CMaxBlockSize;
    CHAR       I2CBusString [8];
} I2CBUSINFO;
```

I2CBusInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
I2CBusID	Type of I2C bus	0x00 SMBus 0x01 OEM I2C Bus 0x10 LFP EDID 0x11 VGA DDC 0x80 and above OEM defined (see hardware manual)
reserved		
I2CBusFlags	Bit 0 = standard speed supported Bit 1 = fast speed supported Bit 2 = reserved Bit 3 = transfer speed fixed by hardware Bit 4 = I2C type block transfer supported Bit 5 = SMBUS type block transfer supported 6..7 reserved Bit 8 = 8-bit offset supported Bit 9 = 16-bit offset supported	caller must request standard transfer speed
I2CMaxBlockSize	Binary	maximum number of bytes accepted for block transfer
I2CBusString	ASCII, 0-terminated	optional platforms not supplying this information just return the termination

4.1.4.3. UspI2CRead8

API

BOOL UNIVERSAL UspI2CRead8 (BYTE I2CBusID, BYTE DevAddress, WORD Offset, BYTE *pData, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Reads a BYTE from the selected I2C device on the I2C bus which is specified with BusID.

I2CBusID: Identification of the bus where the transfer shall be executed
DevAddress: I2C device address
Offset: If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
pData: Pointer to a variable where the read byte is stored
Protocol: Protocol to be used

Bit 0...2	addressing	0: none ("Offset" not valid) 1: 8-Bit addressing 2: 16-Bit addressing
Bit 3 ... 6	reserverd	
Bit 7	transfer speed	0: standard transfer speed 1: fast transfer speed

4.1.4.4. UspI2CWrite8

API

BOOL UNIVERSAL UspI2CWrite8 (BYTE I2CBusID, BYTE DevAddress, WORD Offset, BYTE Data, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Writes a BYTE to the selected I2C device on the I2C bus which is specified with BusID.

I2CBusID: Identification of the bus where the transfer shall be executed
DevAddress: I2C device address
Offset: If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
Data: The data to be written.
Protocol: Protocol to be used (definition see 4.1.4.3)

PLEASE NOTE: The transfer speed has a maximum of 100kHz ('standard') and 400kHz ('fast')
 The maximum transfer speed of hardware controlled busses (i.e. SMBus) is specified in the hardware manual.
 Dependent on the implementation real transfer frequencies may be below these limits.

DevAddress: Is transferred 'as is', bit 0 is set according to read (1) or write (0)
 Protocol 8-bit addressing: Only low byte of **Offset** is used and transferred without modification
 Protocol 16-bit addressing: **Offset** is transferred with MSByte first followed by LSByte.

4.1.4.5. UspI2CRead16

API

BOOL UNIVERSAL UspI2CRead16 (BYTE I2CBusID, BYTE DevAddress, WORD Offset, WORD *pData, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Reads a WORD from the selected I2C device on the I2C bus specified with BusID.

I2CBusID:	Identification of the bus where the transfer shall be executed
DevAddress:	I2C device address
Offset:	If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
pData:	Pointer to the variable where the read word is stored
Protocol:	Protocol to be used (definition see 4.1.4.3)

4.1.4.6. UspI2CWrite16

API

BOOL UNIVERSAL UspI2CWrite16 (BYTE I2CBusID, BYTE DevAddress, WORD Offset, WORD Data, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Writes a WORD to the selected I2C device on the I2C bus specified with BusID.

I2CBusID:	Identification of the bus where the transfer shall be executed
DevAddress:	I2C device address
Offset:	If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
Data:	The data to be written.
Protocol:	Protocol to be used (definition see 4.1.4.3)

PLEASE NOTE: The transfer speed has a maximum of 100kHz ('standard') and 400kHz ('fast')
The maximum transfer speed of hardware controlled busses (i.e. SMBus) is specified in the hardware manual.

Dependent on the implementation real transfer frequencies may be below these limits.

DevAddress: Is transferred 'as is', bit 0 is set according to read (1) or write (0)

Data: Data is transferred with LSByte first followed by MSByte.

Protocol 8-bit addressing: Only low byte of **Offset** is used and transferred without modification

Protocol 16-bit addressing: **Offset** is transferred with MSByte first followed by LSByte.

4.1.4.7. UspI2CReadBlock

API

BOOL UNIVERSAL UspI2CReadBlock (BYTE I2CBusID, BYTE DevAddress, WORD Offset, BYTE *pData, DWORD *pCount, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Reads pCount bytes from the selected I2C device on the I2C bus specified with BusID.

I2CBusID: Identification of the bus where the transfer shall be executed
DevAddress: I2C device address
Offset: If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
pData: Pointer to a buffer to store the read data
pCount: **Input:** pointer to a DWORD variable containing the number of bytes to be read
Output: number of bytes effectively read
Protocol: Protocol to be used

Bit 0...2	addressing	0: none ("Offset" not valid) 1: 8-Bit addressing 2: 16-Bit addressing
Bit 3	block transfer mode	0: I2C block transfer 1: SMBUS block transfer
Bit 4 ... 6	reserved	
Bit 7	transfer speed	0: standard transfer speed 1: fast transfer speed

4.1.4.8. UspI2CWriteBlock

API

BOOL UNIVERSAL UspI2CWriteBlock (BYTE I2CBusID, BYTE DevAddress, WORD Offset, BYTE *pData, DWORD *pCount, BYTE Protocol);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Writes pCount bytes to the selected I2C device on the I2C bus specified with BusID.

I2CBusID: Identification of the bus where the transfer shall be executed
DevAddress: I2C device address
Offset: If an addressing within the I2C device is necessary the address is transferred here. The value of "Protocol" specifies the length of the address.
pData: Pointer to a buffer containing the data to be written.
pCount: **Input:** Pointer to a DWORD variable containing the number of bytes to be written
Output: number of bytes totally written
Protocol: Protocol to be used (definition see 4.1.4.7).

PLEASE NOTE: The transfer speed has a maximum of 100kHz ('standard') and 400kHz ('fast')
 The maximum transfer speed of hardware controlled busses (i.e. SMBus) is specified in the hardware manual.
 Dependent on the implementation real transfer frequencies may be below these limits.

DevAddress: Is transferred 'as is', bit 0 is set according to read (1) or write (0)
 Protocol 8-bit addressing: Only low byte of **Offset** is used and transferred without modification
 Protocol 16-bit addressing: **Offset** is transferred with MSByte first followed by LSByte.

4.1.5. Watchdog Functions

Using these functions the watchdog, integrated on the CPU board, can be accessed. The return value of watchdog functions is FALSE if there is no watchdog available in the system.

4.1.5.1. UspWDConfig

API

BOOL UNIVERSAL UspWDConfig (WORD Delay, WORD Timeout, WORD Cmd);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Sets the actual configuration of the watchdog.

Delay:	Initial time until the WD goes active (0.2s steps)
Timeout:	Watchdog trigger interval (0.2s steps)
Cmd:	Command: 0 – Stop watchdog 1 – Start watchdog (after ' Delay ' a trigger has to be carried out within " Timeout "-intervals, the first trigger must come inside the time window of " Delay "+" Timeout ")

4.1.5.2. UspWDGetStatus

API

BOOL UNIVERSAL UspWDGetStatus (BYTE *pStatus);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Returns the status of the watchdog rest flag.

pStatus:	Pointer to a variable, where the actual watchdog status is stored 0 – Normal Power On Reset 1 – Watchdog Timeout Reset
-----------------	--

4.1.5.3. UspWDTrigger

API

BOOL UNIVERSAL UspWDTrigger (void);

Returns: TRUE for correct execution of the function, otherwise FALSE.

Writes the trigger signal to the watchdog. With the watchdog activated this function has to be called at cyclic **Timeout** intervals as specified in 4.1.5.1.

4.1.6. VGA Functions

With these functions brightness and contrast of the connected flat screen can be adjusted.
If the technical hardware to adjust these parameters is not implemented on the CPU board, these functions return FALSE.

The VGA functions are only valid for control elements implemented on the corresponding CPU board. External implementations can not be accessed by these functions.

4.1.6.1. UspBacklightCtrl

API

BOOL UNIVERSAL UspBacklightCtrl(BOOL State)

Returns: TRUE for correct execution of the function, otherwise FALSE.

State: Turns the backlight on (**TRUE**) or off (**FALSE**)

4.1.6.2. UspGetBacklightValue

API

BOOL UNIVERSAL UspGetBacklightValue(BYTE* pBLValue)

Returns: TRUE for correct execution of the function, otherwise FALSE.

pBLValue: Pointer to a variable, where the actual backlight brightness is stored (range 0x00 ... 0xFF)

4.1.6.3. UspSetBacklightValue

API

BOOL UNIVERSAL UspSetBacklightValue(BYTE BLValue)

Returns: TRUE for correct execution of the function, otherwise FALSE.

BLValue: New value for backlight brightness (range 0x00 ... 0xFF).

4.1.6.4. UspGetContrastValue

API

BOOL UNIVERSAL UspGetContrastValue(BYTE* pCValue)

Returns: TRUE for correct execution of the function, otherwise FALSE

pCValue: Pointer to a variable, where the actual contrast value is stored (range 0x00 ... 0xFF)

4.1.6.5. UspSetContrastValue

API

BOOL UNIVERSAL UspSetContrastValue (BYTE CValue)

Returns: TRUE for correct execution of the function, otherwise FALSE

CValue: New value for contrast adjustment (range 0x00 ... 0xFF).

4.1.7. Monitoring Functions

With these functions the current system health state can be monitored. Depending on the hardware implementation of the sensors the values differ between board families.

For a detailed description about the sensor implementation refer to the board user manual.

4.1.7.1. UspGetThermSensorCount

API

BOOL UNIVERSAL UspGetThermSensorCount(DWORD *pTSensorCount);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pTSensorCount: Pointer to a variable, where the actual number of available thermal sensors is stored

4.1.7.2. UspQueryThermSensor

API

BOOL UNIVERSAL UspQueryThermSensor(DWORD TSensorID, TSENSORINFO *pTSensorInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

TSensorID: 0-based ID of the requested temperature sensor. Valid range 0...(pTSensorCount-1)

pTSensorInfo: Pointer to a structure to store sensor information

The structure TSENSORINFO is defined as follows:

```
typedef struct
{
    DWORD    TSensorInfoSize;
    DWORD    SensorType;
    CHAR     SensorString [8];
} TSENSORINFO;
```

TSensorInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
SensorType	Type of thermal sensor	0x00 CPU sensor 0x01 board sensor 0x02 memory sensor 0x03 system sensor 0x80 and above OEM Sensor (see hardware manual)
SensorString	ASCII, 0-terminated	optional platforms not supplying this information just return the termination

4.1.7.3. UspGetThermSensorValue

API

BOOL UNIVERSAL UspGetThermSensorValue(DWORD TSensorID, SDWORD *pTSensorValue);

Returns: TRUE for correct execution of the function, otherwise FALSE.

TSensorID: 0-based ID of the requested temperature sensor. Valid range (0...pTSensorCount-1)
pTSensorValue: Pointer to a variable to store sensor data. Value is in [°C] units. Negative values are reported in 2-complement.

4.1.7.4. UspGetVoltSensorCount

API

BOOL UNIVERSAL UspGetVoltSensorCount(DWORD *pVSensorCount);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pVSensorCount: Pointer to a variable, where the actual number of available voltage sensors is stored.

4.1.7.5. UspQueryVoltSensor

API

BOOL UNIVERSAL UspQueryVoltSensor(DWORD VSensorID, VSENSORINFO *pVSensorInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

VSensorID: 0-based ID of the requested voltage sensor. Valid range 0...(pVSensorCount-1)

pVSensorInfo: Pointer to a structure to store sensor information

The structure VSENSORINFO is defined as follows:

```
typedef struct
{
    DWORD      VSensorInfoSize;
    SDWORD     NominalValue;
    CHAR       SensorString [8];
} VSENSORINFO;
```

VSensorInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
NominalValue	Nominal value for this sensor	1mV units negative values are reported in 2-complement 0x8000 0000 represent 'unknown'
SensorString	ASCII, 0-terminated	optional platforms not supplying this information just return the termination

PLEASE NOTE: A **NominalValue** of 0x8000 0000 indicates that it can't be determined due to dynamic control by the hardware (i.e. dynamic voltage ID of CPU core voltage)

4.1.7.6. UspGetVoltSensorValue

API

BOOL UNIVERSAL UspGetVoltSensorValue(DWORD TSensorID, SDWORD *pVSensorValue);

Returns: TRUE for correct execution of the function, otherwise FALSE.

VSensorID: 0-based ID of the requested voltage sensor. Valid range 0...(pVSensorCount-1)

pVSensorValue: Pointer to a variable to store sensor data. Value is in [mV] units. Negative values are reported in 2-complement.

4.1.8. Mobile Extensions

The mobile extensions take care of requirements for mobile systems. The functions are mandatory for mobile battery powered systems. Non-mobile AC-powered boards and systems return error code `FUNCTION_NOT_SUPPORTED`.

4.1.8.1. UspGetSystemState

This function returns information about current docking and power state.

API

`BOOL UNIVERSAL UspGetSystemState(SYSTEMSTATE *pSystemState);`

Returns: TRUE for correct execution of the function, otherwise FALSE.

pSystemState: Pointer to a structure to store system state information.

The structure `SYSTEMSTATE` is defined as follows:

```
typedef struct
{
    DWORD    SystemStateSize;
    DWORD    MiscState;
    DWORD    PowerState;
} SYSTEMSTATE;
```

SystemStateSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
MiscState	Bit 0 = 1: system docked Bit 1 = 1: lid open Bit 2-31: reserved	miscellaneous system information
PowerState	Bit 0 = 1: AC power connected Bit 1-31: reserved	miscellaneous power information

4.1.8.2. UspGetBatteryCount

This function returns information about supported batteries.

API

BOOL UNIVERSAL UspGetBatteryCount(DWORD *pBatteryCount);

Returns: TRUE for correct execution of the function, otherwise FALSE.

pBatteryCount: Pointer to a variable to store the number of supported batteries.

4.1.8.3. UspGetBatteryInfo

This function returns information about a specific battery.

API

BOOL UNIVERSAL UspGetBatteryInfo(DWORD BatID, BATTERYINFO *pBatteryInfo);

Returns: TRUE for correct execution of the function, otherwise FALSE.

BatID: 0-based index of a specific battery. Valid range is 0... (pBatteryCount - 1)

pBatteryInfo: Pointer to store information about a specific battery.

The structure BATTERYINFO is defined as follows:

```
typedef struct
{
    DWORD    BatteryInfoSize;
    DWORD    PowerUnit;
    DWORD    DesignCapacity;
    DWORD    FullChargeCapacity;
    DWORD    DesignVoltage;
    CHAR     DeviceName[16];
    CHAR     SerialNumber[16];
    CHAR     BatteryType[16];
    CHAR     Manufacturer[16];
}BATTERYINFO;
```

BatteryInfoSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
PowerUnit	Bit 0: 0: capacity/rate in [mWh/mW] 1: capacity/rate in [mAh/mA]	indicates the units used by the battery to report its capacity and charge/discharge rate information.
DesignCapacity	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	battery's design capacity Unit depends on PowerUnit.0
FullChargeCapacity	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	predicted battery capacity when fully charged. Unit depends on PowerUnit.0
DesignVoltage	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	nominal voltage of a new battery Unit is in [mV]
DeviceName	ASCII, 0-terminated	
SerialNumber	ASCII, 0-terminated	
BatteryType	ASCII, 0-terminated	
Manufacturer	ASCII, 0-terminated	

4.1.8.4. UspGetBatteryState

This function returns information about a specific battery.

API

BOOL UNIVERSAL UspGetBatteryState(DWORD BatID, BATTERYSTATE *pBatteryState);

Returns: TRUE for correct execution of the function, otherwise FALSE.

BatID: 0-based index of a specific battery. Valid range is 0... (pBatteryCount -1)
pBatteryState: Pointer to store information about a specific battery.

The structure BATTERYSTATE is defined as follows:

```
typedef struct
{
    DWORD    BatteryStateSize;
    DWORD    Status;
    DWORD    CurrentRate;
    DWORD    RemainingCapacity;
    DWORD    Voltage
} BATTERYSTATE;
```

BatteryStateSize	Binary	size of the structure in bytes, must be set by calling application as input parameter
Status	Bit 0 = 1: battery present Bit 1 = 1: charging Bit 2 = 1: battery critical energy state Bit 3 = battery requests conditioning cycle Bit 4-31: reserved	remaining information in the BATINFO structure is only valid if Status.0 = 1
CurrentRate	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	current that is supplied through the batteries terminals (direction depends on Status.1). Unit depends on PowerUnit.0
RemainingCapacity	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	estimated remaining battery capacity. Unit depends on PowerUnit.0
Voltage	Binary 0x00000000 – 0x7FFFFFFF 0xFFFFFFFF: unknown	voltage across the battery's terminals in [mV]

4.1.9. OEM Functions

Functions described in this chapter can be used to access the OEM hardware directly.

PLEASE NOTE: Improper use of these functions may lead to a system crash.

4.1.9.1. UspOEMRead

API

BOOL UNIVERSAL UspOEMRead(OEMACCESS *pAccessMode)

Returns: TRUE for correct execution of the function, otherwise FALSE

pAccessMode: Pointer to a OEMACCESS structure, where the access mode to the OEM hardware is described.

The structure is defined as follows:

```
typedef struct
{
    DWORD      AccessType;
    DWORD      Address;
    DWORD      reserved;
    WORD       AdressFlags;
    WORD       DataFlags;
    DWORD      DataValue;
} OEMACCESS;
```

AccessType	Access type to the OEM hardware 0: reserved 1: reserved 2: Direct IO	
Address	Address to be accessed	
Reserved		
AdressFlags	Qualification of the address Bit 0 = 1: BYTE address Bit 1 = 1: WORD address Bit 2 = 2: DWORD address Bit 3 – 15: reserved	
DataFlags	Qualification of the data Bit 0 = 1: BYTE Bit 1 = 1: WORD Bit 2 = 1: DWORD	
DataValue	Contains the read data	

4.1.9.2. UspOEMWrite

API

BOOL UNIVERSAL UspOemWrite(OEMACCESS *pAccessMode)

Returns: TRUE for correct execution of the function, otherwise FALSE

pAccessMode: Pointer to a OEMACCESS structure, where the access mode to the OEM hardware is described.

The structure OEMACCESS is described under UspOEMRead(). DataValue contains the data to be written.

4.1.9.3. UspOEMPassThru

BIOS function interface for OEM specific BIOS extension.

API

BOOL UNIVERSAL UspOemPassThru(far void *pPassThruArgs)

Returns: TRUE for correct execution of the function, otherwise FALSE

OemID: ID for a specific OEM function set

pPassThruArgs: Pointer to a structure containing arguments for the specific OEM function

PLEASE NOTE: This function can be used to execute an OEM function set. The arguments are passed through directly to the BIOS without modification.

Application must check before using this API whether it is running on a valid platform supporting the requested function set specified by **OemID**. Otherwise **UspOEMPassThru** will return **FALSE**, **UspGetLastError** reports FUNCTION_NOT_SUPPORTED.

Installation check for a specific OEM function set is done by calling this function with a valid **OemID** and placing a DWORD value with the **inverted OemID** into the argument buffer. If the OEM function set is supported, **UspOEMPassThru** returns **TRUE**, **UspGetLastError** reports SUCCESSFUL.

5. Win32 Definitions

Under operating systems offering the full range of the Win32 API – Windows XP, Windows XPe, Windows 2000 and Windows Vista - the communication interface is implemented in the interface DLL **UspEpc.DLL**.

The device driver **UspEpc.SYS** communicates with the system BIOS.

A test program (UspEpcTest.exe) is included in the release package, this can be used to exercise some of the basic functions.

6. Windows CE

The functions are equivalent to those of the Win32 definitions.

7. Linux

The UspEpc driver for Linux is implemented as a library which can be linked statically or dynamically into the application that needs to access the UspEpc interface.

The functions are equivalent to those of the Win32 definitions.

The Linux package includes a sample test program, which exercises the function calls defined above. Also included are source files, header files and makefiles.

For a detailed description how to use the UspEpc package with Linux, see the release documentation included with the package.

8. Error Codes (UspGetLastError)

Following error codes are defined for the function **UspGetLastError** (Chapter 4.1.1.5):

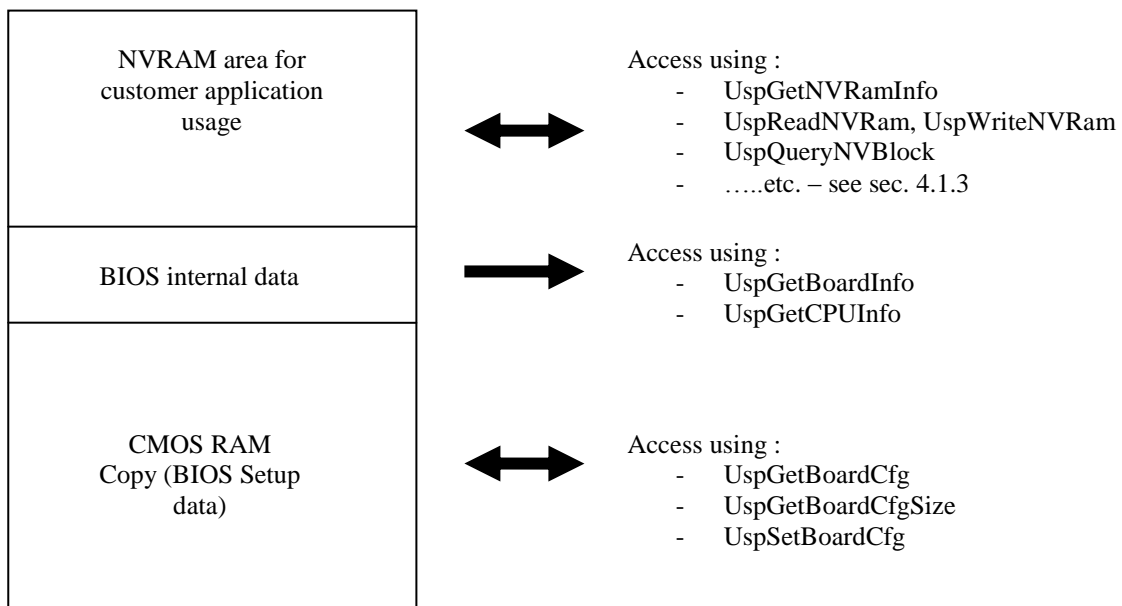
Error Code		Error String
0x0000	SUCCESSFULL	BIOS: SUCCESS
0x8000	FUNCTION_NOT_SUPPORTED	BIOS: function not supported
0x8011	INVALID_DEVICE	BIOS: invalid device
0x8012	DATA_ACCESS_ERROR	BIOS: data access error
0x8021	INVALID_BUS_INDEX	BIOS: I2C invalid bus index
0x8022	NOT_ACKNOWLEDGED	BIOS: I2C no acknowledge
0x8023	PROTOCOL_NOT_SUPPORTED	BIOS: I2C protocol not supported
0x8031	NVRAM_NOT_READABLE	BIOS: NVRam not readable
0x8032	NVRAM_NOT_WRITEABLE	BIOS: NVRam not writeable
0x8033	NVRAM_INDEX_OUT_OF_RANGE	BIOS: NVRam index out of range
0x8041	HARDWARE_NOT_PRESENT	BIOS: hardware not present
0x8051	PARAMETER_OUT_OF_RANGE	BIOS: parameter out of range
0x8100	REGISTER_DEVICE_FAILED	DLL: starting Win32 driver failed (register device)
0x8101	CREATE_FILE_FAILED	DLL: starting Win32 driver failed (create file)
0x8102	CREATE_MUTEX_FAILED	DLL: starting Win32 driver failed (create mutex)
0x8103	INVALID_PARAMETER_NULL_POINTER	DLL: invalid parameter (NULL pointer)
0x8104	INVALID_DLL_VERSION	DLL: invalid DLL version
0x8105	WD_INVALID_CMD	DLL: watchdog: invalid command
0x8106	INVALID_ACCESS_TYPE	DLL: OEMRead/Write, invalid AddressType
0x8107	INVALID_ADDRESS_FLAG	DLL: OEMRead/Write, invalid AddressFlags
0x8108	INVALID_DATA_FLAGS	DLL: OEMRead/Write, invalid DataFlags

9. Data Types

BYTE	unsigned 8-bit value	
WORD	unsigned 16-bit value	
DWORD	unsigned 32-bit value	
SBYTE	signed 8-bit value	2-complement
SWORD	signed 16-bit value	2-complement
SDWORD	signed 32-bit value	2-complement

10. EEPROM Data organsiation

Shown below is a diagram how the 512Byte EEPROM data is partitioned.



11. Product Support

MSC engineers and technicians are committed to provide support to our customers whenever needed.

Before contacting Technical Support of MSC Vertriebs GmbH, please consult the respective pages on our web site at www.msc-ge.com/support-boards for the latest documentation, drivers and software downloads. If the information provided there does not solve your problem, please contact our Technical Support as follows:

Email: support.boards@msc-ge.com

Phone: +49 (0)8165 906-200