Perfection in Automation
www.br-automation.com

APC910 / PPC900
Implementation Guide

Date: 17. January 2014

I Version information

Version	Date	Comment	Responsible
1.00	10-Sep-12	First edition	HOH
1.10	11-Jan-13	Additions regarding UPS IF option: • Differences with APC810 described (starting on page 9). • New: UPS Service command (starting on page 39). • New: UPS Service register (starting on page 46). • BrMtcx.h listing updated (starting on page 59). • New: UPS code examples (starting on page 151). New: Notice for applying Serial ID updated (page 19). New: Notes indicating supported modules for module-specific commands (starting on page 29). New: Module correction command (page 34). Module fan data status bits corrected (page 38). Foot notes for module numbers updated (page 15).	HOH
1.11	20-Mar-13	Reading the PC fan statistic values updated/corrected. UPS Service command corrected: Device number and address were in different order. UPS User Settings data and UPS Shutdown data corrected: the lower limit is 10 instead of 0 seconds. Typos corrected. Reading the UPS statistics ("On Battery" hours and cycles) updated: Command data, example.	HOH
2.00	09-Jan-14	„APC910 IF USV“ changed to „UPS IF Option“. MTCX Command Status register: • MTCX error code 86h added. • Mode R/W corrected to RO. MTCX Command Param register: Target device 0 and 2 removed and 8 (UPS) added. The target device in the command descriptions is now defined as number also. Flash / EEPROM Service command: Address range defined more exact (0 to 3FFFh). Panel commands: Device number range corrected (0 to 15 instead of 1 to 15). Several write and text errors corrected. References corrected and added. LED Service command: LEDs 64 to 127 added in the data. Display Service command: Command code corrected (48h instead of 30h). Watchdog Service command: Time unit ms added. Module Info command: Data size for material number part 3 corrected (8 instead of 16 bytes) and Module Device ID data corrected (hardware revision removed). Module Voltage command: Bit range corrected (0 to 15 instead of 0 to 14) and notes added for Vin (always 0) and Wcpu (power of system instead of CPU). UPS Status register: Description/footnotes of status flags updated and corrected (e.g. temperature alarm at -30 or +60 instead of -42 and +82 °C). UPS Service command: Description completed. UPS user settings: Description completed. Code sample for read/write UPS user settings: description completed. Code sample for UPS shutdown: description completed. GetFanStatistics code example: Parameter FanNumber added. PPC900 added: • New display unit (module 1) and bus fan kit (module 12).	HOH

Version	Date	Comment	Responsible
		• RAM temperatures can be read. • MTCX device ID 0Fh added. • Comments added for supported panel numbers. • Display unit added in descriptions. FPGA Header ID „P9S3“ for SDL3 Display Link added. Device IDs added for AP830, AP9x3 with SDL and AP9x3 with SDL3. Addition „resistive“ for touch screen in function descriptions. BIOS ID „S“ for chipset HM76 added. Module Statistics command: backlight statistics for display unit and fan number for fan statistics added. GetPanelType code example simplified. Code example added for reading operating hours and power-on cycles of the display unit.	

Table 1: Version information

II Organization of safety notices

Safety notices in this document are organized as follows:

Safety notice	Description
Danger!	Disregarding these safety guidelines and notices can be life-threatening.
Warning!	Disregarding these safety guidelines and notices can result in severe injury or substantial damage to equipment.
Caution!	Disregarding these safety guidelines and notices can result in injury or damage to equipment.
Information:	This information is important for preventing errors.

Table 2: Organization of safety notices

III Table of contents

1 Introduction

This document describes how the device-specific functions of the B&R Automation PC 910 (abbreviation: APC910) and B&R Panel PC 900 (abbreviation: PPC900) can be operated using a PC application.

The following functions are available on an APC910 and PPC900:

- Reading firmware versions
- Reading device information (serial number, model number, etc.)
- Reading statistical information (hours of operation, etc.)
- Reading temperatures
- Reading fan speed
- Reading device information (serial number, model number, etc.) from the display unit and connected Automation Panels
- Reading and setting the display brightness of the display unit of a PPC900 and connected Automation Panels
- Reading the key matrix of the display unit of a PPC900 and connected Automation Panels
- Reading the LEDs of the display unit of a PPC900 and connected Automation Panels
- Controlling the Run LED
- Operating the watchdog
- Reading and setting a user serial ID
- Reading status values from the UPS IF option
- Reading and writing user settings from the UPS IF option
- Shutting down the UPS IF option

The device-specific functions are primarily controlled using the built-in Maintenance Controller (MTCX) on the APC910 and PPC900 systems.

A 128-byte I/O area can be used to operate the MTCX as well as provide device-specific data. Commands are used for functions that are not time-critical. Data that must be queried often or accessed in short time intervals is provided by the MTCX in registers.

The BIOS version can be read from the BIOS ROM using the B&R vendor string.

Note: Device specific functions on an APC910 and PPC900 can be accessed using the B&R Automation Device Interface (ADI) in Windows. ADI is included as a driver in Windows images created by B&R (Windows Embedded Standard 2009, Windows Embedded Standard 7) and can be installed separately as well. Support for the ADI is provided by the ADI Development Kit and ADI .NET SDK. The ADI driver, ADI development kit and ADI .NET SDK can be downloaded from the B&R homepage www.br-automation.com .

2 Prerequisites and requirements

It is recommended that you install the latest BIOS and firmware versions on the PC. These can be downloaded from www.br-automation.com.

Information:

Some functions may require specific BIOS or firmware versions. The necessary versions are indicated in the respective register descriptions, command descriptions and code examples.

To communicate with the MTCX, you must be able to access I/O port addresses in the PC application. To read the BIOS ROM, you must be able to access memory addresses.

If this is not possible, I/O and memory may have to be accessed using a driver or another system component (depending on the operating system). Interrupt operation is not necessary.

Basic knowledge of C is required in order to understand the code examples in this document.

3 Overview

The section "Compatibility with past device families" (page 9) provides information about porting existing functions from other device families (e.g. APC810).

The "MTCX Interface" section (starting on page 11) includes information about the basics of device-specific functions.

It is possible to skip over these sections and go right to using the functions from the "Code examples" section (starting on page 58). However, please see the "Coding notes" section first (starting on page 55).

4 Compatibility with past device families

The APC910 and PPC900 are much more modular than the previous APC620, PPC700, APC810, PPC800, PP300/400 and PPC300 device families.
It is not only possible to access system unit functions and information or to be accessed by connected Automation Panels, but the following additional modules are also supported on the device:

- Display unit (PPC900 only)
- Bus unit (optional on PPC900)
- Display link (= Monitor / Panel option, optional, APC910 only)
- Memory module 1 (optional)
- Memory module 2 (optional)
- Fan kit (optional)
- IF option 1 (optional)
- IF option 2 (optional)
- Slide-in 1 (optional)
- Slide-in 2 (optional, APC910 only)
- CPU board
- Bus fan kit (optional, PPC900 only)

New module-specific MTCX commands have been introduced that make it possible to access these modules.

For this reason, the following points must be taken into account when planning to use functions implemented on another device family (e.g. APC810):

- The **MTCX_CMD_DEVICE_INFO** command is used to read the CPU board's device information (model number, serial number, etc.). To read device information from the system unit of other devices, e.g. the APC810, the new **MTCX_CMD_MODULE_INFO** command must be used (see page 29).
- Statistics (operating hours, etc.) can be read from individual modules using the new **MTCX_CMD_MODULE_STATISTICS** command (see page 32).
- It is no longer possible to read the temperature directly from MTCX registers; instead, the new **MTCX_CMD_MODULE_TEMPERATURE** command is used (see page 31).
- It is also no longer possible to read fan revolutions directly from MTCX registers; instead, the new **MTCX_CMD_MODULE_FAN** command is used (see page 38).
- The **UPS Service** command (see page 39) can only be used to shutdown the UPS – restarting is not possible. The delay time for shutting down the UPS has a valid value range of 10 to 1200 instead of 2 tp 65535 seconds.
- The UPS does not have its own firmware – the UPS function is performed by the MTCX - therefore there is no firmware version to read.
- Some of the UPS status flags in the **UPS Status** register (see page 47) are not relevant.
- The **UPS Values** register (see page 48) only contains battery voltage and current. The UPS IF option cannot read the battery capacity and service life.
- The **UPS Specials** register is not provided.

There are also a few characteristics specific to APC910 and PPC900 devices that must be kept in mind:

- Device-specific voltage values can be read using the new **MTCX_CMD_MODULE_VOLTAGE** command (see page 36).
- The Display Link FPGA version can be read with the **MTCX_CMD_MODULE_HEADER** command (see page 37).
- The MTCX firmware is no longer being divided up between the FPGA and PX32 components. For this reason, it is no longer possible to read the FPGA version. The "entire" MTCX version can be read like the PX32 version used to be read (see page 86).
- After writing a new user serial ID with the **Device Info** command (see page 19), it is not necessary to restart the device to effect the change in the MTCX so that it can be read back again.

Some of the other things that remain the same as before include all key and LED functions, all Automation Panel-specific functions and watchdog handling.

Keep in mind the following differences when using functions that were implemented for PP500, APC510 or APC511:

- The APC910 and PPC900 uses different module numbers and addresses Automation Panels differently: Panel number starting with 0 instead of 1.
- The APC910 does not have a built-in display.
- The display unit of a PPC900 is addressed with panel number 15 instead of 0 as on the PP500.
- Rotary encoders and node numbers are not supported on the APC910 and PPC900.
- The device may contain a Display Link FPGA instead of an I/O board FPGA.

5 MTCX Interface

5.1 Overview of registers

The MTCX interface is comprised of several registers with a starting I/O port address of 4100h and a size of 128 bytes. Each register is 32 bits wide. Register information is stored in INTEL format.

Registers are divided into the following main categories:

4100h+ Offset	Register range	Description
0000h-000Fh	MTCX configuration registers	Settings for the MTCX interface and the PC
0010h-0027h	MTCX Command Service registers	Interface for MTCX commands
0028h-002Fh	Hardware Info registers	Device-specific information
0030h-004Fh	Baseboard Service registers	Baseboard settings and information
0050h-0057h	UPS Service registers	UPS status values and operating data
005Ch-005Fh	Panel Switch registers	Selects the panel for the Panel Service range (60h-7Fh)
0060h-007Fh	Panel Service registers	Settings for panels 1-16

Table 3: Overview of MTCX registers

Individual registers will be explained in the following sections.

5.2 MTCX configuration registers

MTCX configuration registers contain the settings for the MTCX interface and the PC. They start at I/O port 4100h.

4100h+ Offset	Register	Description
0000	Version	MTCX firmware version
0004	Config COM	COM port configuration
0008	Config Specials	Special device-specific settings
000C	Config 2nd	Extended device-specific settings

Table 4: Overview of MTCX configuration registers

5.2.1 Version register

This register contains the version of the MTCX firmware and is located at I/O port 4100h.

Version register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	B				A							

ID	Name / Function	Default	Mode	Comment
A	Minor version		RO	Minor version number
B	Major version		RO	Major version number

Table 5: MTCX Version register

5.2.2 Config COM register

This register contains settings for device-specific interfaces and is located at I/O port 4104h.

Information:

This register is reserved by B&R.

Config COM register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	E				D				C				F				x	B			A							

ID	Name / Function	Default	Mode	Comment
A	I/O Addr		R/W	Defines the I/O address of the target device. If 03FFh is read, then the device being addressed does not exist.
B	Device activation		R/W	Switches the controller on/off: 0 = Turned on; 1 = Turned off
C	Device index		R/W	Device index of the module
D	Device Type		R/W	Device type: 0 = KBC 1 = COM 2 = LPC
E	Module index		R/W	Module index
F	IRQ number		R/W	Interrupt number

Table 6: MTCX Config COM register

5.2.3 Config Specials register

This register can be used to read special device-specific settings. It is located at I/O port 4108h.

Config Specials register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	B	x	x	x	x	A							

ID	Name / Function	Default	Mode	Comment
A	Reserved		RO	
B	Panel count	0	RO	Maximum number of supported panels[1]: 0 = 16 panels 1 = 1 panel

Table 7: MTCX Config Specials register

5.2.4 Config 2nd register

This register contains extended device-specific settings and is located at I/O port 410Ch.

Config 2nd register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	C								x	x	x	B	A			

ID	Name / Function	Default	Mode	Comment
A	Reserved	Fh	RO	
B	Display found	0	RO	"Display present" code: 0 = No display, 1 = Display found
C	User input	00h	R/W	The user can write directly to this byte: Bit 0: Run LED status: 0 = Off, 1 = On

Table 8: MTCX Config 2nd register

[1] On APC910 and PPC900 systems, the default setting is always 0 (16 panels).

5.3 MTCX command service registers (command interface)

MTCX command service registers represent the command interface of the MTCX. They are available for general, non time-critical commands and consist of several registers starting at I/O port 4110h.

4110h+ Offset	Register	Description
0000h	MTCX Command Status	Command statuses
0004h	MTCX Command Param	Command parameters
0008h	MTCX Command Data 0	Command data: Bytes 0 to 3
000Ch	MTCX Command Data 1	Command data: Bytes 4 to 7
0010h	MTCX Command Data 2	Command data: Bytes 8 to 11
0014h	MTCX Command Data 3	Command data: Bytes 12 to 15

Table 9: Overview of MTCX Command Service registers

Individual registers are explained in the following sections.

5.3.1 MTCX Command Status register

The MTCX uses this register to transmit status and error codes for a command being executed.
It is located at I/O port 4110h.

MTCX Command Status register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	C	B			A							

ID	Name / Function	Default	Mode	Comment
A	Error code		RO	Error code (only applies for status 011b): 00h = No error 80h = IRQ conflict 81h = Unknown command 82h = Timeout 83h = Incorrect parameter 84h = Device busy 85h = Device error (DVI data R/W) 86h = Data error
B	Status		RO	Status of command: 000b = Interface is available 001b = Command completed without error 010b = Reserved 011b = Command completed with error 100b to 111b = Reserved
C	Lock	0	R/W	Lock bit: This bit is not evaluated by the MTCX and can be used by the application to synchronize with other commands. 0 = Interface available, 1 = Locked

Table 10: MTCX Command Status register

5.3.2 MTCX Command Param register

The parameters for the command to be executed by the MTCX must be entered in this register.
It is located at I/O port 4114h.

MTCX Command Param register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
E								D				C				x	B	A													

ID	Name / Function	Default	Mode	Comment
A	Addr		WO	Destination address
B	Dir	0	WO	Command direction: 1 = Read data 0 = Write (or program) data
C	DevNum	0	WO	Device number With panel-specific commands, the device number is used as the panel number: 0 = 1. Device, 1 = 2. Device, ...15 = 16. Device With module-specific commands (command code 6xh), the device number is used as the module number: 0 = System unit 1 = Display unit (PPC900 only) 2 = Bus unit (optional on PPC900) 3 = Display Link (optional, APC910 only) 4 = IF Option 1 (optional) [2] [3] 5 = IF Option 2 (optional) 6 = Memory module 1 (optional) 7 = Memory module 2 (optional) 8 = Fan kit (optional) 9 = Slide-in 1 (optional) 10 = Slide-in 2 (optional, APC910 only) 11 = CPU board 12 = Bus fan kit (optional, PPC900 only)
D	Target	0	WO	Identifies the target device: 1 = Baseboard 3 = Panel 4 = Panel settings 8 = UPS and other values (with module specific commands)
E	Command	0	WO	Command code Writing to this byte triggers the interrupt for executing the command in the MTCX.

Table 11: MTCX Command Param register

Information:

If you do not have 32-bit access to I/O ports in your PC application, you cannot write to Byte 3 of the Command Param register until valid values have been written to Bytes 0–2 since this is what triggers command execution on the MTCX.

[2] An APC910 IF UPS is addressed as IF Option 1.
[3] MTCX versions prior to 0.07 use different module numbers for the IF options: 4 = IF Option 2 and 5 = IF Option 1.

5.3.3 MTCX Command Data [0..3] registers

MTCX Command Data registers 0–3 contain the data area for reading and writing data. They begin at I/O port 4118h. A maximum of 16 bytes can be transferred. All registers have the same structure.

MTCX Command Data [0..3] registers				
31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0				
A				
ID	Name / Function	Default	Mode	Comment
A	Data		R/W	Data depends on the command.

Table 12: MTCX Command Data [0..3] registers

5.3.4 Command execution

A command is executed by the MTCX command interface according to the following schema:

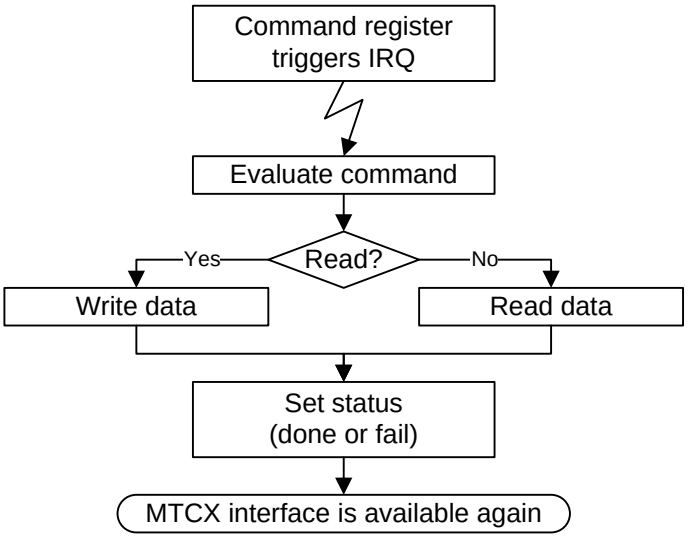

Figure 1: MTCX command execution

When a command is executed, the MTCX is always the slave and the PC is always the master. The commands are issued by the PC application. Each time the Command Param register is written to, an interrupt is triggered in the MTCX. The command is then evaluated in the MTCX's interrupt function. Depending on the command, the data is either accepted or returned. When the command is finished executing, its status is set in the command status register. The command interface is then available once again.

The lock bit in the Command Status register allows the PC application itself to manage whether the command interface is busy or available.

5.3.5 MTCX commands

The following commands are supported by the MTCX:

Command	Command code	Comment
Version Info	10h	Reads the firmware version
Device Info	12h	Reads device information
Key Layer	18h	Sets the key layer
Key Service	19h	Controls key processing
Flash / EEPROM Service	20h	Reads data from / writes data to flash or EEPROM memory
Statistics Info	30h	Reads statistics (Automation Panels only)
LED Service 0	40h	Switches LEDs on the first layer
LED Service 1	41h	Switches LEDs on the second layer
LED Service 2	42h	Switches LEDs on the third layer
LED Service 3	43h	Switches LEDs on the fourth layer
Display Service	48h	Configures the display
Watchdog Service	50h	Configures the watchdog, resets the software
Module Info	61h	Reads device information from a PC module
Module Tempera-ture	64h	Reads temperatures from a PC module
Module Statistics	65h	Reads statistics from a PC module
Module Correction	66h	Reads/writes a PC module's correction data
Module Voltage	68h	Reads a PC module's voltage values
Module Header	6Ah	Reads the firmware header from a PC module
Module Fan	6Bh	Reads fan information from a PC module
UPS Service	90h	UPS commands, shutdown

Table 13: MTCX commands

A detailed description of individual commands is provided in the following sections.

5.3.5.1 Version Info command

This command is used to read the version information from a device's flash memory.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
MTCX Version Info	10h	1 (Base-board)	1	1	-	4 bytes	Reads the MTCX firmware version (also possible via the **Version register**, see page 12)
Scan code version info	10h	1 (Base-board)	1	2	-	4 bytes	Reading the version of the scan code data format

Table 14: Version Info command

The data read by the Version Info command are arranged the same as the **Version** register of the MTCX (see page 12).

5.3.5.2 Device Info command

This command can be used to read and set the User Serial ID and to read the device information from the PC's factory settings.

Information:

This command will always access the system unit on an APC910 and the CPU board on a PPC900! Information from other PC modules can be read using the new Module Info MTCX command (see page 29).

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
ID Service	12h	1 (Base-board)	1/0	0	0	8 bytes	Reads/sets the user serial ID
Manufacturer Info	12h	1 (Base-board)	1	1	0	16 bytes	Reads the device ID, manufacturer ID, etc.
Serial number	12h	1 (Base-board)	1	2	0	16 bytes	Reads the serial number
Model Number 0	12h	1 (Base-board)	1	2	10h	16 bytes	Reads the first part of the model number
Model Number 1	12h	1 (Base-board)	1	2	20h	16 bytes	Reads the second part of the model number
Model Number 2	12h	1 (Base-board)	1	2	30h	9 bytes	Reads the third part of the model number
Parent info	12h	1 (Base-board)	1	2	40h	8 bytes	Reads the parent device ID and compatibility ID

Table 15: Device Info command

The following table explains the **ID Service** data.

Offset	Name / Function	Data length	Comment
00h	Reserved	4 bytes	
04h	User	4 bytes	Can be freely defined by the customer: 00000000h to FFFFFFFFh This information is saved in built-in EEPROM memory.

Table 16: ID Service data

Note: Changes to the User Serial ID are applied immediately by the MTCX – the device does not have to be restarted.

The following table explains the **Manufacturer Info** data:

Offset	Name / Function	Data length	Comment
00h	Device ID	4 bytes	Device ID (SAP model number): 00000000h to FFFFFFFFh
04h	Compatibility ID	2 bytes	Compatibility ID: 0000h to FFFFh
06h	Vendor ID	4 bytes	Vendor ID: 0 = B&R
0Ah	Hardware Revision	5 bytes	Hardware revision as an ASCII string (including null character)
0Fh	Reserved	1 byte	

Table 17: Manufacturer Info data

The following table explains the **Serial Number** data:

Offset	Name / Function	Data length	Comment
00h	Serial number	12 bytes	Serial number of the PC as an ASCII string (including null character)
0Ch	Reserved	4 bytes	

Table 18: Serial Number data

The following tables explain the **Model Number** data:

Offset	Name / Function	Data length	Comment
00h	Model Number Part 1	16 bytes	Part 1 (Bytes 0 to 15) of the model number as an ASCII string (including null character)

Table 19: Model Number 0 data

Offset	Name / Function	Data length	Comment
00h	Model Number Part 2	16 bytes	Part 2 (Bytes 16 to 31) of the model number as an ASCII string (including null character)

Table 20: Model Number 1 data

Offset	Name / Function	Data length	Comment
00h	Model Number Part 3	9 bytes	Part 3 (Bytes 32 to 41) of the model number as an ASCII string (including null character)

Table 21: Model Number 2 data

The following table explains the **Parent Info** data:

Offset	Name / Function	Data length	Comment
00h	Device ID	4 bytes	Device ID (SAP model number) of the parent device: 00000000h to FFFFFFFFh (= no parent)
04h	Compatibility ID	2 bytes	Compatibility ID (in binary) of the parent device: 0000h to FFFFh (= no parent)
06h	Reserved	2 bytes	

Table 22: Parent Info data

5.3.5.3 Key Layer command

This command sets the key layer.

Information:

The key layer is only set if a valid key configuration is present on the device. Nevertheless, keep in mind that this command does not return an error! The status of the key configuration can be determined by reading the MTCX Baseboard KeyState register (see page 44).

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Key Layer	18h	1 (Base-board)	0	0	-	4 bytes	Sets the key layer

Table 23: Key Layer command

The following table illustrates the **Key Layer** data.

Offset	Name / Function	Data length	Comment
00h	Key Layer	1 byte	Key layers: 0 to 3
01h	Reserved	3 bytes	

Table 24: Key Layer data

Note: The current key layer can be read from the MTCX's **Baseboard Key State** register (see page 44).

5.3.5.4 Key Service command

This command makes it possible to control how keys (scan codes, panel lock) on the display unit of a PPC900 and a connected Automation Panel are processed.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Scancode Lock	19h	1 (Base-board)	1/0	0	-	4 bytes	Locks a panel's scan codes
Scancode Restart	19h	1 (Base-board)	1	1	-	-	Rereads scan codes from flash memory (e.g. after an update)
Panel Lock Time data	19h	1 (Base-board)	1/0	3	-	4 bytes	Reading/setting the panel lock time
Panel Lock	19h	1 (Base-board)	1/0	4	-	4 bytes	Locks/unlocks panels

Table 25: Key Service command

Note: Scancode Lock and Scancode Restart are only relevant if a valid key configuration is present on the device.

The following table explains the **Scan Code Lock** data:

Offset	Name / Function	Data length	Comment
00h	Lock Bits	2 bytes	Lock bits: Bit x = 1: Scan codes from Panel x are locked Bit x = 0: Scan codes from Panel x are unlocked
02h	Reserved	2 bytes	

Table 26: Scan Code Lock data

Note: It is also possible to determine whether a panel's scan codes are locked by reading the MTCX's **Panel Flags** register (see page 53).

The following table explains the **Panel Lock Time** data:

Offset	Name / Function	Data length	Comment
00h	Lock time	2 bytes	Panel lock time in ms: 0 to 65535
02h	Reserved	2 bytes	

Table 27: Panel Lock Time data

The panel lock time specifies how long a panel is locked (via resistive touch screen or matrix keys) while data is being input on another panel.

When the PC is restarted or when the scan codes are refreshed (e.g. after downloading a new key configuration), the panel lock time is reset to the value listed in the key configuration.

The following table explains the **Panel Lock** data when reading (direction = 1):

Offset	Name / Function	Data length	Comment
00h	Lock Bits	2 bytes	Lock bits: Bit x = 1: Panel x is locked Bit x = 0: Panel x is unlocked
02h	Reserved	2 bytes	

Table 28: Panel Lock data (reading)

The following table explains the **Panel Lock** data when writing (direction = 0).

Offset	Name / Function	Data length	Comment
00h	Lock Bits	2 bytes	Lock bits: Bit x = 1: Panel x is locked Bit x = 0: Panel x is unlocked
02h	Mask Bits	2 bytes	Lock bit masking: Bit x = 1: Evaluate Lock Bit x, Bit x = 0: Ignore Lock Bit x

Table 29: Panel Lock data (writing)

Panels are normally locked automatically when data is being input on other panels. This command is used to selectively lock particular panels.

Note: It is also possible to determine whether a panel is locked by reading the MTCX's **Panel Flags** register (see page 53).

5.3.5.5 Flash / EEPROM Service command

This command is used to read and program data in flash and EEPROM memory.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Scan Codes	20h	1 (Base-board)	1/0	2	0 to 3FFFh	16 bytes	Reads/writes the key configuration Max. 256 kB (flash memory)
Factory settings	20h	1 (Base-board)	1/0	3	0 to 7Fh	16 bytes	Reads/writes the factory settings (of the system unit) 2048 bytes (EEPROM)
Panel FPGA	20h	3 (Panel)	1/0	0 to 14	0 to 3FFFh[4]	16 bytes	Reads/writes the panel's FPGA firmware Max. 256 Kb (flash memory) on AP900 and AP830 or max. 640 Kb on AP830 and AP9x3
Panel Factory Settings	20h	4 (Panel Settings)	1/0	0 to 14	0 to Fh or 7Fh	16 bytes	Reads/writes the panel's factory settings 256 bytes (EEPROM) on AP900 and AP800 or max. 2048 bytes on AP830 and AP9x3

Table 30: Flash / EEPROM Service command

Up to 16 bytes of data or firmware code is stored in the data registers per command execution.

Caution!

Make sure that no data is written to flash or EEPROM memory. Otherwise, data or firmware that is needed to operate the PC may get overwritten!

Information:

The "Panel FPGA" and "Panel Factory Settings" command function are not supported by the display unit of a PPC900 (panel 15) and AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51).

[4] On AP830 and AP9x3 special address values must be used for access to the panel firmware (for B&R internal use only).

5.3.5.6 Statistics Info command

This command is used to read statistics from connected Automation Panels.

Information:

This command is not supported by the display unit of a PPC900 (panel 15) and AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51).
Statistics from PC modules can be read using the new MTCX Module Statistics command (see page 32).

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Panel Statistics	30h	3 (Panel)	1	0 to 14	-	4 bytes	Reads a panel's operating hours and power-on cycles (of the back-light)

Table 31: Statistics Info command

The following table explains the **Panel Statistics** data:

Offset	Name / Function	Data length	Comment
00h	Power-on Cycles	2 bytes	Backlight power-on cycles: 0 to 65355
02h	Power-on Hours	2 bytes	Backlight operating hours: 0 to 65535

Table 32: Panel Statistics data

5.3.5.7 LED Service command

This command reads or sets the LEDs of the display unit of a PPC900 and a connected Automation Panel.

Command function	Command code	Target device	Direction	Device number	Address	Data length	Comment
LED Service 0	40h	3 (Panel)	1/0	0 to 15	[5]	16 bytes	Reads/Sets the 64 LEDs on the first layer
LED Service 1	41h	3 (Panel)	1/0	0 to 15	[5]	16 bytes	Reads/Sets the 64 LEDs on the second layer
LED Service 2	42h	3 (Panel)	1/0	0 to 15	[5]	16 bytes	Reads/Sets the 64 LEDs on the third layer
LED Service 3	43h	3 (Panel)	1/0	0 to 15	[5]	16 bytes	Reads/Sets the 64 LEDs on the fourth layer

Table 33: LED Service command

Up to 64 LEDs can be read or set simultaneously with one command. Two commands are therefore necessary in order to read/set all of the LEDs. The number of LEDs is specified by the user. For example, it is possible to read or set only 3 LEDs in a layer starting with the 14th LED.

Information:

LED Service commands do not return an error if the device does not support LEDs at all. This situation must be handled in the application, e.g. by first reading the number of LEDs from the MTCX Panel Specials register (see page 52).

The following table explains the **LED Service** data:

Offset	Name / Function	Data length	Comment
00h	LED[0..15] or [64..79]	4 bytes	2 bits per LED status
04h	LED[16..31] or [80..95]	4 bytes	2 bits per LED status
08h	LED[32..47] or [96..111]	4 bytes	2 bits per LED status
0Ch	LED[48..63] or [112..127]	4 bytes	2 bits per LED status

Table 34: LED service data

Each LED status is defined using 2 bits. The following bit patterns are valid:

Bits	LED status
00b	Off
01b	Slow blinking
10b	Fast blinking
11b	On

Table 35: LED status

[5] Address: Bits 0–7 = LED offset, Bits 8–13 = Number of LEDs – 1

5.3.5.8 Display Service command

This command can be used to read and modify settings for the display unit of a PPC900 and a connected Automation Panel.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Display Settings	48h	3 (Panel)	1/0	0 to 15	0	4 bytes	Reads/writes display settings

Table 36: Display Service command

The following table explains the **Display Settings** data.

Offset	Name / Function	Data length	Comment
00h	Brightness	1 byte	Brightness of the backlight in percent: 0 to 100, 255 = not supported
01h	Reserved	3 bytes	

Table 37: Display Settings data

5.3.5.9 Watchdog Service command

This command configures the watchdog or performs a software reset of the PC.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Watchdog Config	50h	1 (Base-board)	1/0	0	-	4 bytes	Reads/sets the watchdog time window
Software reset	50h	1 (Base-board)	0	2	-	-	Resets the PC

Table 38: Watchdog Service command

The following table explains the **Watchdog Config** data:

Offset	Name / Function	Data length	Comment
00h	Minimum time	2 bytes	Watchdog minimum time: 0 to 65535 ms
02h	Maximum time	2 bytes	Watchdog maximum time: 0 to 65535 ms

Table 39: Watchdog Config data

Notes:

- If the minimum time is greater than the maximum time, the MTCX exchanges the times automatically: minimum time will be maximum time and vice versa.
- If minimum time and maximum time have the same value (greater 0), the MTCX returns an error.
- If minimum time and maximum time are both 0, the watchdog is disabled.
- After the time window has been set, the watchdog is enabled when the toggle bit in the Baseboard Support register (see page 43) is written to.

5.3.5.10 Module Info command

This command can be used to read device information from a PC module.

Command function	Command code	Target device	Direction	Device number	Address	Data length	Comment
Module Device ID	61h	0	1	0 to 12 (module number)	100h	4 bytes	Reads a module's device ID
Module Vendor & Compatibility ID	61h	0	1	0 to 12 (module number)	200h	6 bytes	Reads a module's manufacturer and compatibility ID
Module HW Revision	61h	0	1	0 to 12 (module number)	300h	5 bytes	Reads a module's hardware revision information
Module Serial Number	61h	0	1	0 to 12 (module number)	400h	12 bytes	Reads a module's serial number
Module Model Number 0	61h	0	1	0 to 12 (module number)	500h	16 bytes	Reads part 1 of a module's model number
Module Model Number 1	61h	0	1	0 to 12 (module number)	501h	16 bytes	Reads part 2 of a module's model number
Module Model Number 2	61h	0	1	0 to 12 (module number)	502h	9 bytes	Reads part 3 of a module's model number
Module Parent Info	61h	0	1	0 to 12 (module number)	700h	6 bytes	Reads a module's parent device and compatibility ID

Table 40: Module Info command

Each PC module can theoretically contain device information. This command will return an error for device information that does not exist.

Note:

- On APC910 module 11 (CPU board) provides limited device information (i.e. only Device ID, Hardware Revision and Model Number contain valid values and e.g. the Serial Number is "empty").
- On PPC900 module 0 (system unit) and module 1 (CPU board) provide the same device information.
- On PPC900 modules 6 and 7 (memory module 1 and 2) provide the serial number and model number from the SPD data of the DRAM and pseudo values for all other device information.

The following table explains the **Module Device ID** data:

Offset	Name / Function	Data length	Comment
00h	Device ID	4 bytes	Device ID (SAP model number): 00000000h to FFFFFFFFh

Table 41: Module Device ID data

The following table explains the **Module Vendor & Compatibility ID** data:

Offset	Name / Function	Data length	Comment
00h	Vendor ID	4 bytes	Vendor ID: 0 = B&R
04h	Compatibility ID	2 bytes	Compatibility ID: 0000h to FFFFh

Table 42: Module Vendor & Compatibility ID data

The following table explains the **Module HW Revision** data:

Offset	Name / Function	Data length	Comment
00h	Hardware Revision	5 bytes	Hardware revision as an ASCII string (including null character)

Table 43: Module HW Revision data

The following table explains the **Module Serial Number** data:

Offset	Name / Function	Data length	Comment
00h	Serial number	12 bytes	Serial number as an ASCII string (including null character)

Table 44: Module Serial Number data

The following tables explain the **Module Model Number** data:

Offset	Name / Function	Data length	Comment
00h	Model Number Part 1	16 bytes	Part 1 (Bytes 0 to 15) of the model number as an ASCII string (including null character)

Table 45: Module Model Number 0 data

Offset	Name / Function	Data length	Comment
00h	Model Number Part 2	16 bytes	Part 2 (Bytes 16 to 31) of the model number as an ASCII string (including null character)

Table 46: Module Model Number 1 data

Offset	Name / Function	Data length	Comment
00h	Model Number Part 3	9 bytes	Part 3 (Bytes 32 to 41) of the model number as an ASCII string (including null character)

Table 47: Module Model Number 2 data

The following table explains the **Module Parent Info** data:

Offset	Name / Function	Data length	Comment
00h	Device ID	4 bytes	Device ID (SAP model number) of the parent device: 00000000h to FFFFFFFFh (= no parent)
04h	Compatibility ID	2 bytes	Compatibility ID (in binary) of the parent device: 0000h to FFFFh (= no parent)

Table 48: Module Parent Info data

5.3.5.11 Module Temperature command

This command can be used to read temperature values from a PC module.

Command function	Command code	Target device	Direction	Device number	Address	Data length	Comment
Module Temperature	64h	0 to 3 (sensor number)	1	0, 1, 3, 6, 7, 9, 10 and 11 (module number)	0	4 bytes	Reads a module's temperature sensor value

Table 49: Module Temperature command

Each PC module can theoretically have up to four temperature sensors. This command will return an error for temperature sensors that do not exist.

Note: Currently, temperature values can only be read for module 0, 1, 3, 6, 7, 9, 10 and 11 (6 and 7 only on PPC900).

The following table explains the **Module Temperature** data:

Offset	Name / Function	Data length	Comment
00h	Status	1 byte	Internal at B&R
01h	Reserved	1 byte	
02h	Value	2 bytes	Bits 0–5: Reserved Bits 6–15: Temperature value in 0.25°C

Table 50: Module Temperature data

5.3.5.12 Module Statistics command

This command can be used to read statistics from a PC module.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Module Statistics	65h	0	1	0, 1, 2, 3, 4, 5 (module number)	100h	8 bytes	Reads the number of operating hours and power-on cycles from a module
Backlight Statistics	65h	1	1	1 (module number)	100h	8 bytes	Reads the number of operating hours and power-on cycles from the backlight of the display unit
Fan Statistics	65h	0 to 3 (fan number)	1	8, 12 (module number)	200h	8 bytes	Reads the number of operating hours and power-on cycles from a fan
UPS Statistics	65h	3	1	4 (module number)	100h	8 bytes	Reads "On Battery" hours and cycles from the UPS

Table 51: Module Statistics command

Each PC module can theoretically contain statistic values. This command will return an error for statistic values that do not exist.

Note: Currently, statistic values can only be read for module 0, 1, 2, 3, 4, 5, 8 and 12 (1 and 12 only on PPC900).

The following table explains the **Module Statistics** data:

Offset	Name / Function	Data length	Comment
00h	Power-on Hours	4 bytes	Operating hours (in ¼ hours): 0 to 16777212
04h	Power-on Cycles	4 bytes	Power-on cycles: 0 to 16777212

Table 52: Module Statistics data

The following table explains the **Backlight Statistics** data:

Offset	Name / Function	Data length	Comment
00h	Backlight-on Hours	4 bytes	Operating hours (in ¼ hours): 0 to 16777212
04h	Backlight-on Cycles	4 bytes	Power-on cycles: 0 to 16777212

Table 53: Backlight Statistics data

The following table explains the **Fan Statistics** data:

Offset	Name / Function	Data length	Comment
00h	Fan-on Hours	4 bytes	Operating hours (in ¼ hours): 0 to 16777212
04h	Fan-on Cycles	4 bytes	Power-on cycles: 0 to 16777212

Table 54: Fan Statistics data

The following table explains the **UPS Statistics** data:

Offset	Name / Function	Data length	Comment
00h	"On Battery" Hours	4 bytes	UPS "On Battery" hours (in ¼ hours): 0 to 16777212
04h	"On Battery" Cycles	4 bytes	UPS "On Battery" cycles: 0 to 16777212

Table 55: UPS Statistics data

5.3.5.13 Module Correction command

This command can be used to read correction data from a PC module.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
UPS User Settings	66h	0	1/0	4 (module number)	600h	8/4 bytes	Reads/writes a module's correction data

Table 56: Module Correction command

Each PC module can theoretically contain correction data. This command will return an error for correction data that does not exist.

Note: Only the correction data for the UPS IF option module is described in this document.

The **UPS User Settings** are in the UPS EEPROM and currently only contain the low battery shutdown time. This is the time used by the UPS to wait at low battery in battery mode or other errors (e.g. over temperature) before turning off the UPS unit's power outlets.

This avoids deep discharging the UPS battery when the UPS is not turned of by the operating system with the UPS Shutdown command (see page 39) or the operating system does not shutdown.

Caution!

A Low Battery Shutdown terminates any UPS Shutdown command (see page 39) already in progress, which can shorten the system's planned shutdown time!

The following table illustrates the **UPS User Settings** data during reading.

Offset	Name / Function	Data length	Comment
00h	OK	1 byte	"Data Ok" flag: 0 or 1
01h	Special	1 byte	Internal at B&R
02h	Counter	2 bytes	Number of writes of this data
04h	LowBatShutdownTime	2 bytes	Time in seconds before the UPS shuts down after a "Low Battery" alarm.
06h	Reserved	2 bytes	FFFFh

Table 57: UPS User Settings data (read)

Information:

If there are no UPS User Settings on the UPS, then the MTCX returns error 86h and the default value 180 seconds is used as LowBatShutDownTime.

The following table illustrates the **UPS User Settings** data during writing.

Offset	Name / Function	Data length	Comment
00h	LowBatShutdownTime	2 bytes	Time in seconds before the UPS shuts down after a "Low Battery" alarm: 10 to 1200 [6]
02h	Reserved	2 bytes	FFFFh

Table 58: UPS User Settings data (write)

Note: Changes to the **UPS User Settings** are applied immediately by the MTCX – the device does not have to be restarted.

[6] Values outside the valid range will be limited by the MTCX.

5.3.5.14 Module Voltage command

This command can be used to read voltage values from a PC module.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Voltage	68h	0	1	0 (module number)	0	8 bytes	Reads voltage values from the system unit

Table 59: Module Voltage command

Note: Voltage values can currently only be read for module 0.

The following table explains the **Voltage** data:

Offset	Name / Function	Data length	Comment
00h	Vbat	2 bytes	Battery voltage: Bits 0–14: Voltage in 10 mV Bit 15: Status 0 = Good, 1 = Bad
02h	Vin	2 bytes	Input voltage: Bits 0–14: Voltage in 100 mV **The value is always 0 on APC910 and PPC900!** [7] Bit 15: Status 0 = Good, 1 = Bad
04h	Wcpu	2 bytes	Power of the complete system in 10 mW [8] [9]
06h	Reserved	2 bytes	

Table 60: CPU Voltage data

[7] Requires MTCX version 1.01 or higher on APC910 and MTCX version 1.02 or higher on PPC900. Older MTCX versions return invalid values.
[8] On other platforms this value indicates the power of the CPU only.
[9] Requires MTCX version 1.01 or higher on APC910 and MTCX version 1.02 or higher on PPC900. Older MTCX versions return invalid values.

5.3.5.15 Module Header command

This command can be used to read the header of the current FPGA firmware from a PC module.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
FPGA header	6Ah	0	1	0, 3 (module number)	0	16 bytes	Reads the FPGA header

Table 61: Module Header command

Note: Currently, only the header for module 0 and 3 can be read.

The following table explains the **FPGA Header** data:

Offset	Name / Function	Data length	Comment
00h	ID	4 bytes	Header code: "APC9" on system unit (module 0) "LSDL" on SDL display link (module 3) “P9S3” on SDL3 display link (module 3)
04h	Version	4 bytes	FPGA firmware version as an ASCII string (without null character): "vvyy" vv is the major version, yy is the minor version.
08h	Counter	4 bytes	Update counter: 0 to 4294967295 Increased by 1 at every update
0Ch	Reserved	2 bytes	
0Eh	CRC	2 bytes	16-bit CRC for header

Table 62: FPGA Header data

5.3.5.16 Module Fan command

This command can be used to read the fan speed from a PC module.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
Module Fan	6Bh	0 to 3 (fan number)	1	8, 12 (module number)	0	4 bytes	Reads the fan speed from the fan kit module

Table 63: Module Fan command

Each PC module can theoretically have up to four fans. This command will return an error for fans that do not exist.

Note: Fan speeds can currently only be read for module 8 (fan kit) and module 12 (bus fan kit).

Information:

If a fan kit is not present, fan speeds can be read but they are always 0, and bit 1 is set in "Status". You can try to read the device ID with the Module Info command (see page 29) to detect if a fan kit is present.

The following table explains the **Module Fan** data:

Offset	Name / Function	Data length	Comment
00h	Status	1 byte	Fan status: Bit 0 = 1: Fan too slow, not reaching the minimum speed Bit 1 = 1: Stopped fan alarm
01h	Reserved	1 byte	
02h	Value	2 bytes	Fan rotations: 0 to 65535 RPM

Table 64: Module Fan data

5.3.5.17 UPS Service command

This command can be used to control special UPS functions.

Command function	Command code	Target device	Direc-tion	Device number	Ad-dress	Data length	Comment
UPS Shutdown	90h	8 (UPS)	0	0	1	4 bytes	Shutdown UPS

Table 65: UPS Service command

The following table illustrates the **UPS Shutdown** data.

Offset	Name / Function	Data length	Comment
00h	Delay	2 bytes	Delay time for shutting down the UPS in seconds: 10 to 1200 [10]
02h	Reserved	2 bytes	

Table 66: UPS Shutdown data

This command can be sent from the operating system to the UPS when the power supply fails before the operating system itself performs a shutdown.

Caution!

The specified delay time must be longer than the time required by the operating system to shut-down. Otherwise data may be lost. Make sure to also account for the UPS Low Battery Shutdown time (see page Seite 34)!

Information:

- **The UPS is always automatically turned off as soon as the system enters standby mode. Note: The UPS battery is charged in standby mode. [11]**
  The UPS is also automatically turned off in battery mode with a configurable amount of time (see page 34) if a low battery charge is detected.
  But by means of the shutdown command the UPS is turned off also when the system does *not* shutdown properly. This avoids further discharge of the battery.
- **The shutdown command is also necessary for automatic reboot of the PC, when the power returns already during shutdown of the system. [12]**
- **The UPS Shutdown command does not return an error if an UPS is not present. This situation must be handled in the application, e.g. by first reading the "linked" flag in the UPS Status register (see page 47).**

[10] Values outside the valid range will be limited by the MTCX.
[11] Charging in standby mode is supported only with MTCX version 1.00 or higher.
[12] On APC910 MTCX firmware 1.00 or higher and CPU board controller firmware 9.32 or higher is required.

5.4 Hardware Info registers

Hardware Info registers contain device-specific information and begin at I/O port 4128h.

4128h+ Offset	Register	Description
0000h	Hardware Info	Device-specific information
0004h	Hardware Info 2	Extended device-specific information

Table 67: Overview of MTCX Hardware Info registers

Individual registers are explained in the following sections.

5.4.1 Hardware Info registers

This register contains device-specific information and is located at I/O port 4128h.

Hardware Info registers

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	D	C	B	A							

ID	Name / Function	Default	Mode	Comment
A	Device Type		RO	Device type: 0Eh =APC910 0Fh = PPC900 00h and FFh indicate that the device type cannot be read and/or its detection is not supported. All other values are reserved.
B	MTCX Boot Area		RO	MTCX (on system unit) was booted from: 0 = Lower area 1 = Upper area
C	BIOS Boot Area		RO	BIOS was booted from: [13] 0 = Backup area 1 = Normal area
D	FPGA Boot Area		RO	FPGA (on Display Link) booted from: 0 = Lower area 1 = Upper area

Table 68: MTCX hardware info registers

[13] Not relevant on APC910.

5.4.2 Hardware Info 2 register

This register contains additional device-specific information and is located at I/O port 412Ch.

MTCX Hardware Info 2 register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																															

ID	Name / Function	Default	Mode	Comment
A	MTCX Counter		R/W	Update counter for the current MTCX image of the system unit. This value is in Intel format and is written to once during each boot procedure.

Table 69: MTCX Hardware Info 2 register

5.5 Baseboard Service registers

The Baseboard Service registers contain data and settings for the baseboard and begin at I/O port 4130h.

4130h+ Offset	Register	Description
0000h	Baseboard Support	Toggles NMI status and watchdog
0004h	Reserved	
0008h	Reserved	
000Ch	Reserved	
0010h	Reserved	
0014h	Reserved	
0018h	Baseboard KeyState	Key editing settings
001Ch	Baseboard Specials	Special device settings

Table 70: Overview of MTCX Baseboard Service registers

Individual registers are explained in the following sections.

5.5.1 Baseboard Support register

This register contains the watchdog toggle bit and the settings for SMC NMI processing. It is located on I/O port 4130h.

Baseboard Support register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	D	C	B	A						E							x	x	x	x	x	x	x	x	x

ID	Name / Function	Default	Mode	Comment
A	SMC Status	0	RO	Shows reason for reset: Bit 0 = 1: Watchdog Bit 1 = 1: Power fail Bit 2 = 1: Reset button Bit 3 = 1: Software reset Bit 4 = 1: Power button
B	SMC NMI IRQ		RO	Shows whether the SMC NMI IRQ is active (can be used in an NMI routine to inquire whether the NMI was triggered by the SMC): 1 = Active 0 = Inactive
C	SMC status invalid		RO	Indicates whether the SMC status is valid: 1 = No SMC status to read 0 = SMC status is valid The SMC status is already valid by the time the NMI is called and remains so after a warm restart.
D	Watchdog Toggle		R/W	Watchdog toggle bit: Must be 0 to confirm the watchdog
E	NMI Skip Reset	0	WO	NMI reset skip: After every NMI logic event, the following resets can be aborted within the NMI reset time (default = 10 ms). Bit 0 = 1: Watchdog Bit 2 = 1: Reset button Bit 3 = 1: Software reset

Table 71: MTCX Baseboard Support register

5.5.2 Baseboard KeyState register

This register contains special settings for handling matrix keys and key configurations. It is located at I/O port 4148h.

Baseboard KeyState register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	D	C								B				A			

ID	Name / Function	Default	Mode	Comment
A	Level	0	R/W	Currently set key layer: 0 to 3
B	Mode	0	RO	Currently set layer mode: 0 = Shift, 1 = Toggle, 2 = One-shot
C	Checksum	0	RO	Checksum of the key configuration in flash memory
D	Status	0	RO	Indicates whether the key configuration in flash memory is valid: 0 = Invalid 1 = Valid

Table 72: MTCX Baseboard KeyState register

Information:

A key level that has been changed with the key layer command (see page 21) will often be delayed briefly before being displayed in the Baseboard Key State register, due to the internal processes in the MTCX.

5.5.3 Baseboard Specials register

This register contains special PC values and is located at I/O port 414Ch.

Baseboard Specials register																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	X	x	x	x	x	x	x	x	x	x	x	x	x	x	x	C	B	A		x	x	x	x	x	x	x	x

ID	Name / Function	Default	Mode	Comment
A	Bat State		RO	CMOS battery status: [14] 0 = Battery OK 1 = Reserved 2 = Reserved 3 = Battery not OK
B	Power button		RO	Status of the power button: 0 = Pressed 1 = Not pressed
C	Reset button		RO	Status of the reset button: 0 = Pressed 1 = Not pressed

Table 73: MTCX Baseboard Specials register

[14] The battery is checked when the PC is started and every 24 hours thereafter. The battery is subjected to a brief load (1 second) during this measurement.

5.6 UPS Service registers

The UPS Service registers contain the status flags and UPS operating data and are located starting at I/O port 4150h.

4150h+ Offset	Register	Description
0000h	UPS status	UPS status flags and battery voltage
0004h	UPS Values	UPS operating data
0008h	UPS specials	Special UPS values (not used)

Table 74: MTCX UPS Service register overview

Individual registers are explained in the following sections.

5.6.1 UPS Status register

This register contains the status values of the UPS. It is located at I/O port 4150h.

UPS Status register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D								C																x	x	x	x	x	x	B	A

ID	Name / Function	Default	Mode	Comment
A	Detected	0	RO	Indicates whether the UPS has been detected: 1 = UPS detected 0 = UPS not detected
B	Linked	0	RO	Indicates whether the UPS is connected: [15] 1 = UPS connected 0 = UPS not connected
C	Flags	0	RO	UPS status flags: Bit 0 = 1: Power supply OK Bit 1 = 1: Battery settings are faulty or do not exist [16] Bit 2 = 1: Battery operation Bit 3 = 1: Battery low Bit 4 = 1: Battery reverse polarity Bit 5 = 1: Battery error (missing, wrong voltage or temperature etc.) [17] Bit 6 = 1: Battery lifespan has been reached [18] Bit 7 = 1: Shutdown in progress [19] Bit 8 = 1: Overcurrent alarm Bit 9 = 1: Factory settings are faulty or missing [20] Bit 10 = 1: User settings not supported [21] Bit 15 = 1: Battery temperature exceeded or not measurable [22]
D	Bat Temp	0	RO	Temperature of the UPS battery in °C: Measurement range is -42 to +82; -128 = Temperature cannot be read

Table 75: UPS Status register

[15] "Linked" indicates whether communication between MTCX and UPS is functioning. On the APC910 IF UPS, "Detected" and "Linked" are always both 0 or 1.
[16] Not supported from MTCX (always 0).
[17] Bit 5 can be set by alone or together with other bits (for example bit 4).
[18] Not supported from MTCX (always 0).
[19] Either automatic shutdown from the UPS itself or forced by UPS Service command (see page 38).
[20] Always 0. Note: UPS is not detected without valid factory settings ("Detected" and "Linked" = 0).
[21] APC910: User settings are supported only with MTCX version 0.08 or higher.
[22] Temperature alarm triggered at -30 and +60 °C.

5.6.2 UPS Values register

This register contains the operating data of the UPS. It is located at I/O port 4154h.

Information:

The register only contains relevant data if the UPS is connected (see UPS Status Register on page 47).

UPS Values register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
B																A															

ID	Name / Function	Default	Mode	Comment
A	Bat current	0	RO	Pos. values: Charging current of the UPS battery in mA. Neg. values: Discharge current of the UPS battery in mA.
B	Bat voltage	0	RO	Voltage of the UPS battery in mV.

Table 76: UPS Values register

5.6.3 UPS Specials register

This register contains special values for the UPS and is located on I/O port 4158h (not used by the UPS IF option).

UPS Specials register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

ID	Name / Function	Default	Mode	Comment

Table 77: UPS Specials register

5.7 Panel Switch register

This register is used to switch the MTCX **Panel Service** Register (see page 50). It is located at I/O port 415Ch.

The MTCX makes some panel data available via registers, e.g. temperature, key matrix, etc. All panels share a common register area. Using the **Pane Switch** register you can select the panel that will provide the data in this register area. This data is available immediately once the Panel Switch register has been written to.

Automation Panels connected to the Monitor / Panel port are addressed starting with panel number 0, while those connected to the optional Display Link are addressed starting with panel number 8. The display unit of a PPC900 is addressed with panel number 15.

Panel Switch registers																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	A				x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

ID	Name / Function	Default	Mode	Comment
A	Panel Number	0	R/W	Panel number: 0 = 1. Panel, 1 = 2. Panel etc.

Table 78: MTCX Panel Switch register

5.8 Panel Service registers

These registers contain the data and settings of the panel selected with the **Panel Switch** register (see page 49). They are located starting at I/O port 4160h.

Data for individual panels is queried cyclically by the MTCX and stored in the Panel Service registers. This allows the PC application immediate access to the last read data. The register data are valid as soon as a panel has been detected and remain even when it is no longer linked.

Information:

- **Before accessing this register, the desired panel must first be set using the Panel Switch register.**
- **In order to access panel-specific registers from multiple threads at the same time in the PC application, these access operations must be synchronized.**

4160h+ Offset	Register	Description
0000h	Panel Version	Version of the panel
0004h	Panel TempFan	Temperature and fan monitoring
0008h	Panel Specials	Number of LEDs/keys, key switches
000Ch	Panel Flags	Lock, link status, etc.
0010h	Panel Key Matrix 0	Key matrix: Bytes 0 to 3
0014h	Panel Key Matrix 1	Key matrix: Bytes 4 to 7
0018h	Panel Key Matrix 2	Key matrix: Bytes 8 to 11
001Ch	Panel Key Matrix 3	Key matrix: Bytes 12 to 15

Table 79: Overview of MTCX Panel Service registers

Individual registers are explained in the following sections.

5.8.1 Panel Version register

This register contains the version of the panel and is located at I/O port 4160h.

Information:

- **Before accessing this register, the desired panel must first be set using the Panel Switch register.**
- **No valid version is returned for the display unit of a PPC900 (Panel 15).**
- **The version of the base unit is returned for AP800 extension units.**

Panel Version register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
C																x	x	x	x	B				A							

ID	Name / Function	Default	Mode	Comment
A	Version Minor		RO	Major version number
B	Version Major		RO	Minor version number
C	Device ID		RO	Device detection – specifies the type of panel: 722Ch = AP900 SDL receiver/transceiver 2C6Bh = AP900 LDL receiver/transceiver (obsolete) E05Dh = AP800 SDL receiver (on base unit only) E0DDh = AP800 extension unit 0DE0h = AP830 SDL receiver E035h = AP9x3 SDL receiver 725Ch = AP9x3 SDL3 receiver 5886h = Built-in display of a PP300/400, PP500, PPC300, PPC700, PPC800 or PPC900 device

Table 80: MTCX Panel Version register

5.8.2 Panel TempFan register

This register contains the temperature (ambient temperature of display) and fan RPMs of the selected panel. It is located at I/O port 4164h.

Information:

- **Before accessing this register, the desired panel must first be set using the Panel Switch register.**
- **No valid temperature or fan speed is provided for the display unit of a PPC900 (panel 15) and AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).**

Panel TempFan register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	B												x	x	x	x	x	x	x	x	A							

ID	Name / Function	Default	Mode	Comment
A	Temp	0	RO	Temperature in °C: 0 to 127
B	Fan speed	0	RO	Fan rotations per minute / 4 (currently not supported)

Table 81: MTCX Panel TempFan register

5.8.3 Panel Specials register

This register contains special panel settings and information (backlight, key switch status, number of keys and LEDs). It is located at I/O port 4168h.

Information:

Before accessing this register, the desired panel must first be set using the Panel Switch register.

Panel Specials register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
D								C							B									A							

ID	Name / Function	Default	Mode	Comment
A	Brightness [23]		RO	Brightness of the backlight: 0 to 100%
B	Key Switches	FFh	RO	Key switch status: 00h to FFh Key switch offsets must have been configured in the key configuration using B&R Key Editor Version 2.50 or higher.
C	Key Count		RO	Maximum number of supported matrix keys: 0 or 128 0 means that no keys are supported. Any value other than 0 means that up to x keys are supported. It does not, however, mean that the device actually has that many keys.
D	LED Count		RO	Maximum number of supported matrix LEDs: 0 or 128 0 means that no LEDs are supported. Any value other than 0 means that up to x LEDs are supported. It does not, however, mean that the device actually has that many LEDs.

Table 82: MTCX Panel Specials register

[23] Obsolete: The MTCX's Display Service command must be used to read the brightness value (see page 28).

5.8.4 Panel Flags register

This register contains codes and special settings for the selected panel. It is located at I/O port 416Ch.

Information:

Before accessing this register, the desired panel must first be set using the Panel Switch register.

Panel Flags register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	I	H				x	x	x	G	F				x	x	x	x	x	x	x	x	x	x	x	E	D	C	B	A

ID	Name / Function	Default	Mode	Comment
A	Detected	0	RO	Indicates whether the panel has been detected: 0 = Not detected, 1 = Detected
B	Linked	0	RO	Indicates whether the panel is connected: 0 = Not linked, 1 = Linked
C	Locked	0	RO	Indicates whether the panel (key matrix, resistive touch screen) is locked: 0 = Not locked, 1 = Locked
D	Scancode Lock	0	RO	Indicates whether the scan codes are locked: 0 = Not locked, 1 = Locked
E	Software lock	0	RO	Indicates whether the panel has been locked by a MTCX command: 0 = Not locked, 1 = Locked
F	Equalizer auto	0	RO	Automatic equalizer value (calculated by the system based on cable length): 0 (strong) to 15 (weak)
G	Equalizer Support [24]	0	RO	Indicates whether *Equalizer* is valid (i.e. supported): 0 = Invalid, 1 = Valid
H	Equalizer user	0	R/W	User-defined equalizer value: 0 (strong) to 15 (weak) Use low values (strong equalizer setting) for long cables.
I	Equalizer mode	0	R/W	Indicates whether *EqualizerUser* is being used: 0 = No (Automatic Mode), 1 = Yes (User Mode)

Table 83: MTCX Panel Flags register

Information:

Valid equalizer settings cannot be read or set for the display unit of a PPC900 (pane 15) and AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51).

[24] The equalizer is integrated into Automation Panel devices and adapts the DVI signal to various cable lengths. The equalizer value is automatically calculated based on the cable length. It is possible to set a different equalizer value in order to obtain the best possible display quality (e.g. in case of low-quality cables or poor DVI signal quality).

Information:

- **Due to how the MTCX works internally, a scan code lock modified with the Key Service command (see page 22) may sometimes be delayed briefly before being displayed in "Scan Code Lock" in the Panel Flags register.**
- **Due to how the MTCX works internally, a panel lock that has been modified with the Key Service command (see page 22) may sometimes be delayed briefly before being displayed in "Locked" and "Software Lock" in the Panel Flags register.**

5.8.5 Panel Key Matrix [0..3] registers

These registers contain the status of the matrix keys for the selected panel. They are located from I/O port 4170h to 417Fh.

There are 4 registers for the entire matrix. The key matrix is already debounced.

Information:

Before accessing this register, the desired panel must first be set using the Panel Switch register. Registers with the key status information can also be read when the device does not support any keys at all. This must be avoided in practice, for example, by first reading the number of keys from the MTCX Panel Specials Register (see page 52).

Panel Key Matrix [0..3] registers

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A																															

ID	Name / Function	Default	Mode	Comment
A	Matrix		RO	Up to 32 key states are indicated per register in the key matrix. A pressed key is indicated with 0.

Table 84: MTCX Panel Key Matrix [0..3] registers

6 Coding notes

The following sections will provide some notes about using the code examples found in this document.

6.1 Type punning problem

At some places in the following code examples, a special type of cast statement is used when reading MTCX registers, for example:

```
MTCX_BASEBOARD_TEMP_2ND_REG reg;

*(unsigned long*)&reg = MTCX_READ_PORT_ULONG(MTCX_BASEBOARD_TEMP_2ND_ADDR);
*Temp = reg.Temp1;
```

Listing 1: Reading an MTCX register with type punning

These types of cast statements carry out what is known as "type punning" (see http://en.wikipedia.org/wiki/Type_punning).

Information:

Depending on the compiler and its settings (level of optimization), type punning may result in warnings or errors during compilation. Because the code has been "de-optimized", it is even possible that the program will not be able to run.

Note: This problem does not occur in debug builds or with Microsoft compilers.

You can get around this problem by either reducing the level of optimization (e.g. with -fno-strict-aliasing for the GCC compiler, see also http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Optimize-Options.html#index-fstrict_002daliasing-542) or reworking the code:

```
union t_u
{
    unsigned long l;
    MTCX_BASEBOARD_TEMP_2ND_REG reg;
} u;

u.l = MTCX_READ_PORT_ULONG(MTCX_BASEBOARD_TEMP_2ND_ADDR);
*Temp = u.reg.Temp1;
```

Listing 2: Reading an MTCX register without type punning

Casting will still continue to be used in the code examples to ensure compatibility with older implementation guides as well as applications that have already been ported.

6.2 TODO: Statements

Code examples may include TODO: statements. They should be replaced with the appropriate code.

6.3 Data Formats

The standard C data types used in the code examples must have the following sizes:

Data format	Size (bytes)
char	1
short	2
long	4
int	>= 2

Table 85: Data formats used

6.4 READ_PORT_x, WRITE_PORT_x

You must replace the READ_PORT_x and WRITE_PORT_x functions with the respective I/O port access functions relevant to your development environment.

Example for implementing the READ_PORT_x and WRITE_PORT_x functions with **inp** and **outp** (e.g. for MS-DOS):

```
//
// I/O port access functions.
//

#define READ_PORT_UCHAR(port) inp(port)
#define WRITE_PORT_UCHAR(port, value) outp(port, value)
#define READ_PORT_USHORT(port) inpw(port)
#define WRITE_PORT_USHORT(port, value) outpw(port, value)

typedef union
{
    unsigned long ldata;
    unsigned char bdata[4];
} BYTES_AND_LONG;

// Read unsigned long value from I/O port.
unsigned long READ_PORT_ULONG(unsigned short port)
{
    BYTES_AND_LONG data;

    data.bdata[0] = inp(port++);
    data.bdata[1] = inp(port++);
    data.bdata[2] = inp(port++);
    data.bdata[3] = inp(port++);
    return data.ldata;
}

// Write unsigned long value to I/O port.
void WRITE_PORT_ULONG(unsigned short port, unsigned long value)
{
    BYTES_AND_LONG data;

    data.ldata = value;
    outp(port++, (unsigned)data.bdata[0]);
    outp(port++, (unsigned)data.bdata[1]);
    outp(port++, (unsigned)data.bdata[2]);
    outp(port++, (unsigned)data.bdata[3]);
}
```

Listing 3: Example for implementing the READ_PORT_x and WRITE_PORT_x functions

6.5 READ_REGISTER_x

You have to replace the READ_REGISTER_x function found in the code examples with the memory access functions relevant to your development environment.

Example for implementing the READ_REGISTER_x functions for MS-DOS:

```
// Convert linear address into segment offset address
// for example 000F0000h -> F000:0000
unsigned char READ_REGISTER_UCHAR(unsigned long Register)
{
    return *(char far*)(((Register & 0xFFFF0000) << 12) + (Register & 0x0000FFFF));
}
```

Listing 4: Example for implementing the READ_REGISTER_UCHAR function

7 Code examples

The following sections describe code examples for device-specific functions such as reading temperature values or device information.

7.1 Auxiliary functions

The following code examples show auxiliary functions that are used in the other code examples.

7.1.1 Swapping a 2-byte value

The following C code example shows how the byte order of a 2-byte value can be changed.

This function is needed, for example, when reading the device information of an Automation Panel since some of the data is stored in Big Endian format.

```
// Swap 2 byte value.
unsigned short swapw(unsigned short w)
{
    return (w >> 8) + (w << 8);
}
```

Listing 5: swapw – Swapping a 2-byte value

7.1.2 Swapping a 4-byte value

The following C code example shows how the byte order of a 4-byte value can be changed.

This function is needed, for example, when reading the device information of an Automation Panel since some of the data is stored in Big Endian format.

```
// Swap 4 byte value.
unsigned long swapl(unsigned long dw)
{
    return swapw((unsigned short)(dw >> 16)) + ((unsigned long)swapw((unsigned short)dw) << 16);
}
```

Listing 6: swapl – Swapping a 4-byte value

7.2 MTCX interface functions

The following code examples show how the MTCX interface can be accessed in C.

7.2.1 Definitions for the MTCX interface

This listing contains general C definitions for the MTCX interface: I/O port addresses, structures of MTCX registers, etc.

These definitions are used in the code examples in this document. They can be saved as BrMtcx.h and included in the C code.

```
// BrMtcx.h : Definitions for MTCX interface on APC910 and PPC900.
//
// Copyright (c) Bernecker + Rainer

#ifndef _BRMTCX_H_
#define _BRMTCX_H_

//
// MTCX I/O register addresses.
//

#define MTCX_BASE_ADDR                  0x4100

#define MTCX_VERSION_ADDR               (MTCX_BASE_ADDR + 0x00)
#define MTCX_CONFIG_COM_ADDR            (MTCX_BASE_ADDR + 0x04)
#define MTCX_CONFIG_SPECIALS_ADDR       (MTCX_BASE_ADDR + 0x08)
#define MTCX_CONFIG_2ND_ADDR            (MTCX_BASE_ADDR + 0x0C)

#define MTCX_CMD_STATUS_ADDR            (MTCX_BASE_ADDR + 0x10)
#define MTCX_CMD_PARAM_ADDR             (MTCX_BASE_ADDR + 0x14)
#define MTCX_CMD_DATA0_ADDR             (MTCX_BASE_ADDR + 0x18)
#define MTCX_CMD_DATA1_ADDR             (MTCX_BASE_ADDR + 0x1C)
#define MTCX_CMD_DATA2_ADDR             (MTCX_BASE_ADDR + 0x20)
#define MTCX_CMD_DATA3_ADDR             (MTCX_BASE_ADDR + 0x24)

#define MTCX_HW_INFO_ADDR               (MTCX_BASE_ADDR + 0x28)
#define MTCX_HW_INFO2_ADDR              (MTCX_BASE_ADDR + 0x2C)

#define MTCX_BASEBOARD_SUPPORT_ADDR     (MTCX_BASE_ADDR + 0x30)
#define MTCX_BASEBOARD_KEYSTATE_ADDR    (MTCX_BASE_ADDR + 0x48)
#define MTCX_BASEBOARD_SPECIALS_ADDR    (MTCX_BASE_ADDR + 0x4C)

#define MTCX_UPS_STATUS_ADDR            (MTCX_BASE_ADDR + 0x50)
#define MTCX_UPS_VALUES_ADDR            (MTCX_BASE_ADDR + 0x54)
#define MTCX_UPS_SPECIALS_ADDR          (MTCX_BASE_ADDR + 0x58)

#define MTCX_PANEL_SWITCH_ADDR          (MTCX_BASE_ADDR + 0x5C)
#define MTCX_PANEL_VERSION_ADDR         (MTCX_BASE_ADDR + 0x60)
#define MTCX_PANEL_TEMPFAN_ADDR         (MTCX_BASE_ADDR + 0x64)
#define MTCX_PANEL_SPECIALS_ADDR        (MTCX_BASE_ADDR + 0x68)
#define MTCX_PANEL_FLAGS_ADDR           (MTCX_BASE_ADDR + 0x6C)
#define MTCX_PANEL_KEY_MATRIX0_ADDR     (MTCX_BASE_ADDR + 0x70)
#define MTCX_PANEL_KEY_MATRIX1_ADDR     (MTCX_BASE_ADDR + 0x74)
#define MTCX_PANEL_KEY_MATRIX2_ADDR     (MTCX_BASE_ADDR + 0x78)
#define MTCX_PANEL_KEY_MATRIX3_ADDR     (MTCX_BASE_ADDR + 0x7C)

//
// MTCX command codes.
//
#define MTCX_CMD_VERSION_INFO       0x10
#define MTCX_CMD_DEVICE_INFO        0x12
#define MTCX_CMD_KEY_LAYER          0x18
#define MTCX_CMD_KEY_SERVICE        0x19
#define MTCX_CMD_FLASH_SERVICE      0x20
#define MTCX_CMD_STATISTICS_INFO    0x30
#define MTCX_CMD_LED_SERVICE0       0x40
```

```
#define MTCX_CMD_LED_SERVICE1       0x41
#define MTCX_CMD_LED_SERVICE2       0x42
#define MTCX_CMD_LED_SERVICE3       0x43
#define MTCX_CMD_DISPLAY_SERVICE    0x48
#define MTCX_CMD_WDOG_SERVICE       0x50
#define MTCX_CMD_MODULE_INFO        0x61
#define MTCX_CMD_MODULE_TEMPERATURE 0x64
#define MTCX_CMD_MODULE_STATISTICS  0x65
#define MTCX_CMD_MODULE_CORR        0x66
#define MTCX_CMD_MODULE_VOLTAGE     0x68
#define MTCX_CMD_MODULE_HEADER      0x6A
#define MTCX_CMD_MODULE_FAN         0x6B
#define MTCX_CMD_UPS_SERVICE        0x90

//
// MTCX command (target) device types.
//
#define MTCX_DEV_COMMON     0x00 // obsolet
#define MTCX_DEV_BASEBOARD  0x01
#define MTCX_DEV_SCANCODE   0x02 // obsolet
#define MTCX_DEV_PANEL      0x03
#define MTCX_DEV_PANELSET   0x04
#define MTCX_DEV_UPS        0x08

//
// MTCX command (baseboard) device numbers.
//
#define MTCX_DEVNUM_SCAN     0x02   // Scan codes (key configuration)
#define MTCX_DEVNUM_SET      0x03   // Settings

//
// MTCX command (module) device numbers
// for r/w of temperatures, fans, factory info, statistics
//
#define MTCX_DEVNUM_APC910_PPC900_SYS     0 // system unit
#define MTCX_DEVNUM_PPC900_DISPLAY        1 // display unit
#define MTCX_DEVNUM_APC910_PPC900_BUS     2 // bus unit
#define MTCX_DEVNUM_APC910_LINK           3 // Display Link
#define MTCX_DEVNUM_APC910_PPC900_IF1     4 // IF option 1
#define MTCX_DEVNUM_APC910_PPC900_IF2     5 // IF option 2
#define MTCX_DEVNUM_APC910_PPC900_MEM1    6 // DRAM module 1
#define MTCX_DEVNUM_APC910_PPC900_MEM2    7 // DRAM module 2
#define MTCX_DEVNUM_APC910_PPC900_FAN     8 // fan kit
#define MTCX_DEVNUM_APC910_PPC900_DRIVE1  9 // slide-in slot 1
#define MTCX_DEVNUM_APC910_DRIVE2        10 // slide-in slot 2
#define MTCX_DEVNUM_APC910_PPC900_CPU    11 // CPU board
#define MTCX_DEVNUM_PPC900_FAN2          12 // fan kit 2 (bus fans)
// next defines are for compatibility:
#define MTCX_DEVNUM_APC910_SYS     0 // system unit
#define MTCX_DEVNUM_APC910_BUS     2 // bus unit
#define MTCX_DEVNUM_APC910_LINK    3 // Display Link
#define MTCX_DEVNUM_APC910_IF1     4 // IF option 1
#define MTCX_DEVNUM_APC910_IF2     5 // IF option 2
#define MTCX_DEVNUM_APC910_MEM1    6 // DRAM module 1
#define MTCX_DEVNUM_APC910_MEM2    7 // DRAM module 2
#define MTCX_DEVNUM_APC910_FAN     8 // fan kit
#define MTCX_DEVNUM_APC910_DRIVE1  9 // slide-in slot 1
#define MTCX_DEVNUM_APC910_DRIVE2 10 // slide-in slot 2
#define MTCX_DEVNUM_APC910_CPU    11 // CPU board

//
// MTCX command directions.
//
#define MTCX_DIR_READ    1   // data read command
#define MTCX_DIR_WRITE   0   // data write command

//
// MTCX command error codes (returned at MTCX_FAIL_COMMAND).
//
#define MTCX_ERR_NONE               0x00
#define MTCX_ERR_IRQ_COLLISION      0x80
```

```
#define MTCX_ERR_UNKNOWN_COMMAND    0x81
#define MTCX_ERR_TIMEOUT            0x82
#define MTCX_ERR_BAD_PARAMETER      0x83
#define MTCX_ERR_DEVICE_BUSY        0x84
#define MTCX_ERR_DEVICE_FAILURE     0x85
#define MTCX_ERR_DATA_FAILURE       0x86

//
// MTCX version IDs.
//
#define MTXC_VER_ID_SDLR 0x722c      // AP900 SDL Receiver/Transceiver
#define MTCX_VER_ID_LDLR 0x2c6b      // AP900 LDL Receiver/Transceiver (obsolete)
#define MTCX_VER_ID_AP8H 0xe05d      // AP800 SDL Receiver
#define MTCX_VER_ID_AP8X 0xe0dd      // AP800 Extension Unit
#define MTCX_VER_ID_DISP 0x5886      // built-in display
#define MTCX_VER_ID_AP83 0x0de0      // AP830 SDL Receiver
#define MTCX_VER_ID_AP9X 0xe035      // AP9X3 SDL Receiver
#define MTCX_VER_ID_AP9X_SDL3 0x725c // AP9X3 SDL3 Receiver

//
// MTCX UPS status flags.
//
#define MTCX_UPS_STATUS_POWER_OK      0x0001
#define MTCX_UPS_STATUS_BAT_SETTINGS  0x0002
#define MTCX_UPS_STATUS_ON_BAT        0x0004
#define MTCX_UPS_STATUS_LOW_BAT       0x0008
#define MTCX_UPS_STATUS_BAT_POLARITY  0x0010
#define MTCX_UPS_STATUS_BAT_FAILURE   0x0020
#define MTCX_UPS_STATUS_BAT_LIFETIME  0x0040
#define MTCX_UPS_STATUS_SHUTDOWN      0x0080
#define MTCX_UPS_STATUS_OVER_CURRENT  0x0100
#define MTCX_UPS_STATUS_FACT_SETTINGS 0x0200
#define MTCX_UPS_STATUS_USER_SETTINGS 0x0400
#define MTCX_UPS_STATUS_BAT_TEMP      0x8000

//
// MTCX register definitions.
//

typedef struct
{
    unsigned long Minor     :8;
    unsigned long Major     :4;
    unsigned long Reserve   :20;
} MTCX_VERSION_REG;

typedef struct
{
    unsigned long DisplayNo         :4; // display number, see MTCX_DISPLAY_* defines
    unsigned long DisplayFound      :1; // 0 = no display, 1 = display found
    unsigned long Reserve1          :3;
    unsigned long RunLedState       :1; // Run LED: 0 = off, 1 = on
    unsigned long Reserve2          :23;
} MTCX_CONFIG_2ND_REG;

typedef struct
{
    unsigned long Switches          :8; // Mode/Node switches
    unsigned long Reserve1          :4;
    unsigned long PanelCount        :1; // 0 = 16 panels, 1 = 1 panel
    unsigned long Reserve2          :19;
} MTCX_CONFIG_SPECIALS_REG;

typedef struct
{
    unsigned long Addr      :14;    // Address/Size depending on command; Flash address / 16
    unsigned long Dir       :1;     // 1 = Read, 0 = Write (see MTCX_DIR_x constants)
    unsigned long Reserve   :1;
    unsigned long DevNum    :4;     // see MTCX_DEVNUM_x constants
    unsigned long Target    :4;     // see MTCX_DEV_x constants
    unsigned long Command   :8;     // see MTCX_CMD_x constants
```

```
} MTCX_CMD_PARAM_REG;

typedef struct
{
    unsigned long Error     :8;     // Valid if Status = 011b (see MTCX_ERR_x constants)
    unsigned long Status    :3;     // Command service status:
                                    // 000b = free
                                    // 001b = request finished
                                    // 010b = reserved
                                    // 011b = request failed
                                    // 100b-111b = reserved
    unsigned long Lock      :1;     // 0 = command service free, 1 = locked
    unsigned long Reserve   :20;
} MTCX_CMD_STATUS_REG;

// This structure contains the status byte of the MTCX_CMD_STATUS_REG.
typedef struct
{
    unsigned char Status    :3;
    unsigned char Lock      :1;
    unsigned char Reserve   :4;
} MTCX_CMD_STATUS_REG_BYTE1;

//
// MTCX device types.
//
#define MTCX_DEVICE_APC620    0x01
#define MTCX_DEVICE_APC620E   0x02
#define MTCX_DEVICE_PPC700    0x03
#define MTCX_DEVICE_APC810    0x04
#define MTCX_DEVICE_APC820    0x05
#define MTCX_DEVICE_PPC800    0x06
#define MTCX_DEVICE_PPC300    0x07
#define MTCX_DEVICE_PP300_400 0x08
#define MTCX_DEVICE_PP500     0x0A
#define MTCX_DEVICE_APC510    0x0C
#define MTCX_DEVICE_APC511    0x0D
#define MTCX_DEVICE_APC910    0x0E
#define MTCX_DEVICE_PPC900    0x0F

typedef struct
{
    unsigned long DeviceType     : 8;   // 00h and FFh: not supported
    unsigned long MtcxBootArea   : 1;   // MTCX booted from: 0 = low area, 1 = high area
    unsigned long BiosBootArea   : 1;   // BIOS booted from: 0 = backup area, 1 = normal (update)
area
    unsigned long FpgaBootArea   : 1;   // FPGA booted from: 0 = low area, 1 = high area
                                        // PP500/APC51x: I/O board, APC910/PPC900: Display Link
    unsigned long Reserve        : 21;
 } MTCX_HW_INFO_REG;

typedef struct
{
    unsigned long Reserve1      :9;
    unsigned long NmiSkipReset  :7;     // NMI reset skip:
                                        // bit 0 = Watchdog
                                        // bit 2 = Reset switch
                                        // bit 3 = Software reset (command)
    unsigned long SmcStatus     :6;     // SMC status:
                                        // bit 0 = Watchdog
                                        // bit 1 = Power fail
                                        // bit 2 = Reset switch
                                        // bit 3 = Software reset (command)
                                        // bit 4 = Power button
    unsigned long NmiIrqActive  :1;     // 1 = NMI IRQ active
    unsigned long NmiStatusInval:1;     // 1 = NMI status invalid
    unsigned long WatchdogToggle:1;     // set 0 to toggle watchdog
    unsigned long Reserve2      :7;
} MTCX_BASEBOARD_SUPPORT_REG;

typedef struct
```

```
{
    unsigned long Layer     :4;     // 0 to 3
    unsigned long Mode      :4;
    unsigned long Checksum  :8;     // Checksum of key configuration
    unsigned long Status    :1;     // 0 = Fail, 1 = Okay
    unsigned long Reserve   :15;
} MTCX_BASEBOARD_KEYSTATE_REG;

typedef struct
{
    unsigned long Reserve1      :8;
    unsigned long BatState      :2; // CMOS battery state: 00b = good, 11b = bad
    unsigned long PowerButton   :1; // State of power button: 0 = pressed, 1 = not pressed
    unsigned long ResetButton   :1; // State of reset button: 0 = pressed, 1 = not pressed
    unsigned long Reserve2      :4;
    unsigned long ResetCounter  :8;
    unsigned long Reserve3      :8;
} MTCX_BASEBOARD_SPECIALS_REG;

typedef struct
{
    unsigned long Detected   :1;
    unsigned long Linked     :1;
    unsigned long Reserve1   :6;
    unsigned long Flags      :16; // see MTCX_UPS_STATUS_x constants
    long          BatTemp    :8;  // 0 to 127 °C
} MTCX_UPS_STATUS_REG;

typedef struct
{
    unsigned char Detected   :1;
    unsigned char Linked     :1;
    unsigned char Reserve1   :6;
} MTCX_UPS_STATUS_REG_BYTE0;

typedef struct
{
    unsigned short PowerOk      :1;
    unsigned short BatSettings  :1;
    unsigned short OnBat        :1;
    unsigned short LowBat       :1;
    unsigned short BatPolarity  :1;
    unsigned short BatFailure   :1;
    unsigned short BatLifetime  :1;
    unsigned short Shutdown     :1;
    unsigned short Reserve2     :7;
    unsigned short BatTempAlarm :1;
} MTCX_UPS_STATUS_FLAGS;

typedef struct
{
    long          BatCurrent  :16; // pos. value = charge current in mA
                                   // neg. value = discharge current in mA
    unsigned long BatVoltage  :16; // mV
} MTCX_UPS_VALUES_REG2; // Values register for IF UPS on APC910 and PPC900

typedef struct
{
    unsigned long Minor     :8;
    unsigned long Major     :4;
    unsigned long Reserve   :4;
    unsigned long Id        :16;    // see MTCX_VER_ID_x
} MTCX_PANEL_VERSION_REG;

typedef struct
{
    unsigned long Reserve       :8;
    unsigned long KeySwitches   :8; // 00h to ffh
    unsigned long KeyCount      :8; // 0 or 128
    unsigned long LedCount      :8; // 0 or 128
} MTCX_PANEL_SPECIALS_REG;
```

```
typedef struct
{
    unsigned long Detected          :1;
    unsigned long Linked            :1;
    unsigned long Locked            :1;
    unsigned long ScanCodeLock      :1;
    unsigned long SoftwareLock      :1;
    unsigned long Reserve1          :11;
    unsigned long EqualizerAuto     :4;
    unsigned long EqualizerSupport  :1;
    unsigned long Reserve2          :3;
    unsigned long EqualizerUser     :4;
    unsigned long EqualizerMode     :1;
    unsigned long Reserve3          :3;
} MTCX_PANEL_FLAGS_REG;

typedef struct
{
    unsigned char Detected          :1;
    unsigned char Linked            :1;
    unsigned char Locked            :1;
    unsigned char ScanCodeLock      :1;
    unsigned char SoftwareLock      :1;
    unsigned char Reserve           :3;
} MTCX_PANEL_FLAGS_REG_BYTE0; // byte 0 of register

typedef struct
{
    long          Temp      :8;     // 0 to 127
    unsigned long Reserve1  :8;
    unsigned long FanSpeed  :12;    // RPM / 4
    unsigned long Reserve2  :4;
} MTCX_PANEL_TEMPFAN_REG;

//
// MTCX command data definitions.
//

#define MTCX_CMD_DATA_SIZE    16        // Maximum command data size

#pragma pack(1)

typedef struct
{
    unsigned char   Brightness; // brightness (backlight) in percent: 0 to 100,
                                // 255 = not supported
    unsigned char   Reserve1;
    unsigned short  Reserve2;
} MTCX_DISPLAY_SERVICE_DATA;

// Version info command returns same data as version register.
#define MTCX_VERSION_INFO_DATA MTCX_VERSION_REG

typedef struct
{
    unsigned long  DeviceId;
    unsigned short CompatId;
    unsigned long  VendorId;
    char           HwRevision[5]; // incl. 0 byte
    char           Reserve;
} MTCX_MANUFACTURER_INFO_DATA;

typedef struct
{
    unsigned long  DeviceId;
    unsigned short CompatId;
    char           Reserve[2];
} MTCX_PARENT_INFO_DATA;

typedef struct
```

```
{
    char SerialNumber[12]; // incl. 0 byte
    char Reserve[4];
} MTCX_SERIAL_NUMBER_DATA;

typedef struct
{
    unsigned long PowerOnHours; // 1/4 h
    unsigned long PowerOnCycles;
} MTCX_MODULE_STATISTICS_DATA;

typedef struct
{
    unsigned long BacklightOnHours; // 1/4 h
    unsigned long BacklightOnCycles;
} MTCX_DISPLAY_STATISTICS_DATA;

typedef struct
{
    unsigned long FanOnHours; // 1/4 h
    unsigned long FanOnCycles;
} MTCX_FAN_STATISTICS_DATA;

typedef struct
{
    unsigned long OnBatHours; // 1/4 h
    unsigned long OnBatCycles;
} MTCX_UPS_STATISTICS_DATA;

typedef struct
{
    unsigned short BacklightOnCycles;
    unsigned short BacklightOnHours;
} MTCX_PANEL_STATISTICS_DATA;

typedef struct
{
    unsigned short MinTime;  // 0 to 65535 ms
    unsigned short MaxTime;  // 0 to 65535 ms
} MTCX_WATCHDOG_CONFIG_DATA;

typedef struct
{
    unsigned long Reserve;
    unsigned long User;
} MTCX_ID_SERVICE_DATA;

typedef struct
{
    unsigned short LockTime; // 0 to 65535 ms
    unsigned char  Reserve[2];
} MTCX_PANEL_LOCK_TIME_DATA;

typedef struct
{
    unsigned short LockBits; // Bit x = 1: panel x locked
    unsigned short MaskBits; // Bit x = 1: use LockBits bit x
} MTCX_PANEL_LOCK_DATA;

typedef struct
{
    unsigned char Layer; // 0 to 3
    unsigned char Reserve[3];
} MTCX_KEY_LAYER_DATA;

typedef struct
{
    unsigned short LockBits; // Bit x = 1: scan codes of panel x locked
    unsigned short Reserve;
} MTCX_SCANCODE_LOCK_DATA;
```

```
typedef struct
{
    unsigned short Delay; // 10 to 1200 s
    unsigned short Reserve;
} MTCX_UPS_SHUTDOWN_DATA;

typedef struct
{
    unsigned long Status  :8;
    unsigned long Reserve :8;
    unsigned long Flags   :6;
    long Temp             :10; // Temperature value in 0.25 degrees Celsius
} MTCX_MODULE_TEMPERATURE_DATA;

typedef struct
{
    struct
    {
        unsigned short Value :15; // Battery voltage in 10 mV
        unsigned short State :1;  // Battery status: 0 = good, 1 = bad
    } Vbat;
    struct
    {
        unsigned short Value :15; // Input voltage in 100 mV
        unsigned short State :1;
    } Vin;
    unsigned short Wcpu; // System power consumption in 10 mW
    unsigned short Reserve;
} MTCX_MODULE_VOLTAGE_DATA;

typedef struct
{
    unsigned char Status;
    unsigned char Reserve;
    unsigned short FanSpeed; // Fan speed in rpm
} MTCX_MODULE_FAN_DATA;

#pragma pack()

//
// Maximum values.
//
#define MTCX_MAX_PANELS       16        // Maximum number of panels
#define MTCX_MAX_LAYERS       4         // Maximum number of key and LED layers (per panel)
#define MTCX_MAX_KEYS         128       // Maximum number of keys per layer
#define MTCX_MAX_LEDS         128       // Maximum number of LEDs per layer

//
// Key layer modes.
//
#define MTCX_LAYER_SHIFT   0x00         // Layer shift mode
#define MTCX_LAYER_TOGGLE  0x01         // Layer toggle mode
#define MTCX_LAYER_ONESHOT 0x02         // Layer one-shot mode

//
// LED states.
//
#define MTCX_LED_OFF        0           // LED is off
#define MTCX_LED_BLINK_SLOW 1           // LED is slow blinking
#define MTCX_LED_BLINK_FAST 2           // LED is fast blinking
#define MTCX_LED_ON         3           // LED is on

#endif
```

Listing 7: Definitions for the MTCX interface

7.2.2 Error codes for MTCX interface functions

This list contains the error codes for MTCX interface functions.
These error codes are used in the code examples in this document.

```
//
// MTCX function return values.
// TODO: These return values are used in the code examples.
//       You can replace them with your own values.
//

#define MTCX_OKAY           0
#define MTCX_WORKING        1          // MTCX is working
#define MTCX_FAIL_PARAM     -1         // Invalid function parameter
#define MTCX_FAIL_LOCKED    -2         // Command interface is locked
#define MTCX_FAIL_COMMAND   -3         // Command failed
#define MTCX_FAIL_TIMEOUT   -4         // Timeout detected
#define MTCX_FAIL_DETECT    -5         // Panel not detected
#define MTCX_FAIL_LINK      -6         // Panel not linked
#define MTCX_FAIL_DATA      -7         // Invalid data
#define MTCX_FAIL_NOSUPPORT -8         // Not supported
```

Listing 8: Error codes for MTCX interface functions

7.2.3 Reading the maximum number of panels

Reading the MTCX's **Config Specials** register (see page 13) can determine how many panels are being supported by the MTCX.

Information:

The MTCX always supports up to 16 panels on APC910 and PPC900 devices - regardless of how many panels can actually be connected to the device: e.g. if an APC910 doesn't have a Display Link, then theoretically a maximum of 8 Automation Panels is possible.

```
// Return maximum number of supported panels.
int MtcxGetMaxPanelCount()
{
    MTCX_CONFIG_SPECIALS_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_CONFIG_SPECIALS_ADDR);
    if (reg.PanelCount == 1) // 1 panel
        // For example PP300/400 supports only panel number 0.
        return 1;

    return MTCX_MAX_PANELS;
}
```

Listing 9: MtcxGetMaxPanelCount – Reading the maximum number of panels

The definitions used can be found on page 59.

7.2.4 Switching panels

By setting the MTCX **Panel Switch** register (see page 49) you can select the panel on which data and settings can later be accessed via the **Panel Service** area of the MTCX.

Note: The panel is switched in every panel-specific function in this document's code examples. Depending on the application, it is also possible to remove this from these functions and perform this switch elsewhere, e.g. at the start of the application.

```
// Switch to panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxSwitchToPanel(unsigned int PanelNumber)
{
    // Check how many panels are supported.
    // TODO: you can remove this check if the function is not used
    // on a device that supports only one panel
    if (MtcxGetMaxPanelCount() == 1)
    {
        if (PanelNumber != 0)
            return MTCX_FAIL_PARAM;
    }
    // TODO: you can remove this branch if the function is not used
    // on a device that supports more than one panel
    else // 16 panels
    {
        if (PanelNumber >= MTCX_MAX_PANELS)
            return MTCX_FAIL_PARAM;
        WRITE_PORT_UCHAR(MTCX_PANEL_SWITCH_ADDR + 3, (unsigned char)PanelNumber); // write byte 3
only
    }
    return MTCX_OKAY;
}
```

Listing 10: MtcxSwitchToPanel – Switching panels

The definitions used can be found on page 59.

With the additional evaluation of the “linked” information in the MTCX **Panel Flags** register (see page 53), a function can be created for switching to a linked panel. For performance reasons, only the lowest byte of the **Panel Flags** register is accessed in **MtcxSwitchToLinkedPanel**.

```
// Switch to linked panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxSwitchToLinkedPanel(unsigned int PanelNumber)
{
    int retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG_BYTE0 reg;

        *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_PANEL_FLAGS_ADDR); // read byte 0 only
        if (! reg.Detected) // never detected
            return MTCX_FAIL_DETECT;
        if (! reg.Linked) // once detected, but not linked now?
            return MTCX_FAIL_LINK;
    }
    return retvalue;
}
```

Listing 11: MtcxSwitchToLinkedPanel – Switching to a connected panel

The definitions used can be found on page 59.
The **MtcxReadCommand** function can be found on page 67

7.2.5 Basic functions for the MTCX command interface

These functions are necessary for processing MTCX commands (basic functions) and are called by **MtcxReadCommand** and **MtcxWriteCommand**.

```
// Lock the command interface.
// Return 0 if interface is already locked and
// 1 if interface is locked successfully now.
int MtcxLockCommand()
{
    unsigned long reg = READ_PORT_ULONG(MTCX_CMD_STATUS_ADDR);

    if (((MTCX_CMD_STATUS_REG*)&reg)->Lock)
        return 0;

    ((MTCX_CMD_STATUS_REG*)&reg)->Lock = 1;
    ((MTCX_CMD_STATUS_REG*)&reg)->Error = 0;
    ((MTCX_CMD_STATUS_REG*)&reg)->Status = 0;
    WRITE_PORT_ULONG(MTCX_CMD_STATUS_ADDR, reg);
    return 1;
}

// Unlock the command interface.
void MtcxUnlockCommand()
{
    unsigned long reg = READ_PORT_ULONG(MTCX_CMD_STATUS_ADDR);
    ((MTCX_CMD_STATUS_REG*)&reg)->Error = 0;
    ((MTCX_CMD_STATUS_REG*)&reg)->Status = 0;
    ((MTCX_CMD_STATUS_REG*)&reg)->Lock = 0;
    WRITE_PORT_ULONG(MTCX_CMD_STATUS_ADDR, reg);
}

// Set (request) a command.
//
// Parameters
//   Dir
//     [in] Specifies the direction: 1 = read, 0 = write.
//   Command
//     [in] Specifies the command code (see MTCX_CMD_x constants).
//   Addr
//     [in] Specifies the address (depends on command).
//   DevNum
//     [in] Specifies the device number (see MTCX_DEVNUM_x constants).
//   Target
//     [in] Specifies the target device (see MTCX_DEV_x constants).
//   Data
//     [in/out] Points to the buffer that contains/receives the
//     command data. Can be NULL if no data is required.
//   Size
//     [in] Specifies the data size in bytes: 0 to 16.
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxSetCommand(unsigned int Dir,
                   unsigned int Command,
                   unsigned int Addr,
                   unsigned int DevNum,
                   unsigned int Target,
                   void *Data,
                   unsigned int Size)
{
    MTCX_CMD_PARAM_REG param;

    // Lock command interface.
    if (! MtcxLockCommand())
        return MTCX_FAIL_LOCKED;

    // Write command data.
    if (Data && Size)
    {
```

```
        int Port = MTCX_CMD_DATA0_ADDR;
        unsigned char *p = (unsigned char*)Data;

        while (Size-- > 0)
            WRITE_PORT_UCHAR(Port++, *p++);
    }

    // Write command parameters.
    param.Dir = Dir;
    param.Command = Command;
    param.Addr = Addr;
    param.DevNum = DevNum;
    param.Target = Target;

    WRITE_PORT_ULONG(MTCX_CMD_PARAM_ADDR, *(unsigned long*)&param);
    return MTCX_OKAY;
}

// Read the command status and error code.
//
// Parameters
//   ErrorCode
//     [out] Points to a variable that receives the
//     command error when the function returns with
//     MTCX_FAIL_COMMAND (see MTCX_ERR_x constants).
//
// Return MTCX_OKAY at success, MTCX_WORKING if the
// command interface is busy and MTCX_FAIL_x at failure.
int MtcxGetCommandStatus(unsigned char *ErrorCode)
{
    int retvalue;
    unsigned char regbyte;

    regbyte = READ_PORT_UCHAR(MTCX_CMD_STATUS_ADDR + 1); // read byte 1 only
    retvalue = MTCX_WORKING;
    switch (((MTCX_CMD_STATUS_REG_BYTE1*)&regbyte)->Status)
    {
        case 0x01:
            retvalue = MTCX_OKAY;
            break;
        case 0x03:
            retvalue = MTCX_FAIL_COMMAND;
            break;
    }
    *ErrorCode = READ_PORT_UCHAR(MTCX_CMD_STATUS_ADDR); // read byte 0 only
    return retvalue;
}

// Wait for command response.
//
// Parameters
//   ErrorCode
//     [out] Points to a variable that receives the
//     command error when the function returns with
//     MTCX_FAIL_COMMAND (see MTCX_ERR_x constants).
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxWaitForCommandResponse(unsigned char *ErrorCode)
{
    int retvalue;

    while ((retvalue = MtcxGetCommandStatus(ErrorCode)) == MTCX_WORKING)
    {
        // TODO: add timeout handling and terminate loop
        //       after timeout with retvalue = MTCX_FAIL_TIMEOUT
        ;
    }
    return retvalue;
}
```

Listing 12: Basic functions for the MTCX command interface

The definitions used can be found on page 59.

7.2.6 Reading data with the MTCX command

This function is used to issue an MTCX command for reading data. It will be used in later code examples.

```
// Read data via MTCX command.
//
// Parameters
//   Command
//     [in] Specifies the command code (see MTCX_CMD_x constants).
//   Addr
//     [in] Specifies the address (depends on command).
//   DevNum
//     [in] Specifies the device number (see MTCX_DEVNUM_x constants).
//   Target
//     [in] Specifies the target device (see MTCX_DEV_x constants).
//   Data
//     [out] Points to a buffer that receives the
//     command data. Can be NULL if no data is required.
//   Size
//     [in] Specifies the data size in bytes: 0 to 16.
//   ErrorCode
//     [out] Points to a variable that receives the
//     command error when the function returns with
//     MTCX_FAIL_COMMAND (see MTCX_ERR_x constants).
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxReadCommand(unsigned int Command,
                    unsigned int Addr,
                    unsigned int DevNum,
                    unsigned int Target,
                    void *Data,
                    unsigned int Size,
                    unsigned char *ErrorCode)
{
    int retvalue;

    if (Size > MTCX_CMD_DATA_SIZE)
        return MTCX_FAIL_PARAM;

    // TODO: acquire command synchronization object here
    //       (if multiple commands can be requested parallel)

    retvalue = MtcxSetCommand(MTCX_DIR_READ, Command, Addr, DevNum, Target, 0, 0);
    if (retvalue == MTCX_OKAY)
    {
        retvalue = MtcxWaitForCommandResponse(ErrorCode);
        if (retvalue == MTCX_OKAY)
        {
            int Port = MTCX_CMD_DATA0_ADDR;
            unsigned char *p = (unsigned char *)Data;

            while (Size-- > 0)
                *p++ = READ_PORT_UCHAR(Port++);
        }
        MtcxUnlockCommand();
    }

    // TODO: release command synchronization object here

    return retvalue;
}
```

Listing 13: MtcxReadCommand – Reading data with the MTCX command

The definitions used can be found on page 59.

7.2.7 Writing data with the MTCX command

This function issues an MTCX command for writing data. It will be used in later code examples.

```
// Write data via MTCX command.
//
// Parameters
//   Command
//     [in] Specifies the command code (see MTCX_CMD_x constants).
//   Addr
//     [in] Specifies the address (depends on command).
//   DevNum
//     [in] Specifies the device number (see MTCX_DEVNUM_x constants).
//   Target
//     [in] Specifies the target device (see MTCX_DEV_x constants).
//   Data
//     [in] Points to a buffer that contains the
//     command data. Can be NULL if no data is required.
//   Size
//     [in] Specifies the data size in bytes: 0 to 16.
//   ErrorCode
//     [out] Points to a variable that receives the
//     command error when the function returns with
//     MTCX_FAIL_COMMAND (see MTCX_ERR_x constants).
//
// Return MTCX_OKAY at success and MTCX_FAIL_x at failure.
int MtcxWriteCommand(unsigned int Command,
                     unsigned int Addr,
                     unsigned int DevNum,
                     unsigned int Target,
                     void *Data,
                     unsigned int Size,
                     unsigned char *ErrorCode)
{
    int retvalue;

    if (Size > MTCX_CMD_DATA_SIZE)
        return MTCX_FAIL_PARAM;

    // TODO: acquire command synchronization object here
    //       (if multiple commands can be requested parallel)

    retvalue = MtcxSetCommand(MTCX_DIR_WRITE, Command, Addr, DevNum, Target, Data, Size);
    if (retvalue == MTCX_OKAY)
    {
        retvalue = MtcxWaitForCommandResponse(ErrorCode);
        MtcxUnlockCommand();
    }

    // TODO: release command synchronization object here

    return retvalue;
}
```

Listing 14: MtcxReadCommand – Writing data with the MTCX command

The definitions used can be found on page 59.

7.3 Panel functions

The panel functions are performed by the MTCX.

Depending on the model, one or more Automation Panels can be linked to an APC910 and PPC900. These Automation Panels are addressed via a panel number starting at 0 (when connected to the Monitor / Panel port) or 8 (when connected to the optional Display Link = Monitor / Panel option).

A PPC900 has a display unit, which can also be addressed like a panel using the panel number 15. For technical reasons, not all panel functions are supported for this panel:

- No firmware version
- No statistic values
- No device information
- No temperature
- No equalizer

You can, however, perform the following functions for all panels:

- Reading panel flags (panel "detected", panel "linked", panel "locked", scan codes "locked") – This also makes it possible to determine the number of panels.
- Reading/setting the panel lock time
- Reading/setting the panel lock

The following code examples show how to use these functions.

7.3.1 Checking if a panel is "supported"

It is possible to check whether a panel is supported by attempting to switch to that panel.

The following C code example shows how to check whether a panel is "supported".

```
// Check if panel is supported.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//     Note: PP300/400 supports only panel number 0.
//
// Return 1 if panel is detected and 0 if not.
int IsPanelSupported(unsigned int PanelNumber)
{
    int retvalue;
    int Supported = 0;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        Supported = 1;

    // TODO: release synchronization object for panel data here

    return Supported;
}
```

Listing 15: IsPanelSupported – Checking if a panel is "supported"

The definitions used can be found on page 59.
The **MtcxSwitchToPanel** function can be found on page 68.

7.3.2 Checking if a panel is "detected"

You can also check whether a panel has been detected via the MTCX **Panel Flags** register (see page 53).

The following C code example shows how to check whether a panel has been detected.

```
// Check if panel is detected.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return 1 if panel is detected and 0 if not.
int IsPanelDetected(unsigned int PanelNumber)
{
    int retvalue;
    int Detected = 0;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG_BYTE0 reg;

        *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_PANEL_FLAGS_ADDR); // read byte 0 only
        Detected = reg.Detected;
    }

    // TODO: release synchronization object for panel data here

    return Detected;
}
```

Listing 16: IsPanelDetected – Checking if a panel is "detected"

The definitions used can be found on page 59.
The **MtcxSwitchToPanel** function can be found on page 68.

7.3.3 Checking if a panel is "linked"

You can also check whether a panel is linked (i.e. communication with the panel is functional) using the MTCX **Panel Flags** register (see page 53).

The following C code example shows how to check whether a panel is linked.

```
// Check if panel is linked.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return 1 if panel is linked and 0 if not.
int IsPanelLinked(unsigned int PanelNumber)
{
    int retvalue;
    int Linked = 0;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG_BYTE0 reg;

        *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_PANEL_FLAGS_ADDR); // read byte 0 only
        Linked = reg.Linked;
    }

    // TODO: release synchronization object for panel data here

    return Linked;
}
```

Listing 17: IsPanelLinked – Checking if a panel is "linked"

The definitions used can be found on page 59.
The **MtcxSwitchToPanel** function can be found on page 68.

7.3.4 Checking if a panel is "locked"

You can also check whether a panel has been locked via the MTCX **Panel Flags** register (see page 53).

The following C code example shows how to check whether a panel has been locked.

```
// Check if panel is locked.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return 1 if panel is locked and 0 if not.
int IsPanelLocked(unsigned int PanelNumber)
{
    int retvalue;
    int Locked = 0;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG_BYTE0 reg;

        *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_PANEL_FLAGS_ADDR); // read byte 0 only
        Locked = reg.Locked;
    }

    // TODO: release synchronization object for panel data here

    return Locked;
}
```

Listing 18: IsPanelLocked – Checking if a panel is "locked"

The definitions used can be found on page 59.
The **MtcxSwitchToPanel** function can be found on page 68.

7.3.5 Checking if scan codes are "locked"

You can also check whether the scan codes of a panel have been locked via the MTCX **Panel Flags** register (see page 53).

The scan codes are defined in the key configuration. A key configuration can be created with the B&R Key Editor and loaded onto the device.

The following C code example shows how to check whether a panel's scan codes are locked.

```
// Check if scan codes are locked.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return 1 if scan codes are locked and 0 if not.
int AreScanCodesLocked(unsigned int PanelNumber)
{
    int retvalue;
    int ScanCodeLock = 0;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG_BYTE0 reg;

        *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_PANEL_FLAGS_ADDR); // read byte 0 only
        ScanCodeLock = reg.ScanCodeLock;
    }

    // TODO: release synchronization object for panel data here

    return ScanCodeLock;
}
```

Listing 19: AreScanCodesLocked – Checking if scan codes are "locked"

The definitions used can be found on page 59.
The **MtcxSwitchToPanel** function can be found on page 68.

7.3.6 Reading/setting the panel lock time

The lock time of a panel can be read and set using the MTCX's **Key Service** command (see page 22).

The panel lock time specifies how long input from keys or resistive touch screens on other panels is locked while data is being input on a particular panel. The panel lock time is usually set in the key configuration created in the B&R Key Editor.

The following C code example shows how to read the lock time.

```
// Get lock time of panel.
//
// Parameters
//     [out] Points to a variable that receives the
//     lock time in ms: 0 to 65535.
//
// Return 0 at success and -1 at failure.
int GetPanelLockTime(unsigned short *LockTime)
{
    MTCX_PANEL_LOCK_TIME_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_KEY_SERVICE, 0, 3, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *LockTime = data.LockTime;
    return 0;
}
```

Listing 20: GetPanelLockTime – Reading the panel lock time

The following C code example shows how to set the lock time.

Information:

Specify either 0 (no lock) or values starting at 500 ms for the lock time.

Information:

When the key configuration is reprogrammed or reloaded or the system is rebooted, the panel lock time is reset to the value set in the key configuration.

```
// Set lock time of panel.
//
// Parameters
//   LockTime
//     [in] Specifies the lock time in ms: 0 ro 65535.
//
// Return 0 at success and -1 at failure.
int SetPanelLockTime(unsigned short LockTime)
{
    MTCX_PANEL_LOCK_TIME_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    memset((void*)&data, 0x00, sizeof(data));
    data.LockTime = LockTime;

    // Note: the lock time is reset to the key configuration value
    //       at reboot of the device or when a new key configuration
    //       is downloaded (and the scan codes are "restarted").
    if ((retvalue = MtcxWriteCommand(MTCX_CMD_KEY_SERVICE, 0, 3, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 21: SetPanelLockTime – Setting the panel lock time

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.3.7 Reading/setting the panel lock status

The locking of a panel can be read and set with the MTCX **Key Service** command (see page 22).

Input on a panel via keys or resistive touch screen is usually locked automatically if data is being input on another connected panel and the panel lock time is set in the key configuration or with the Key Service command. After the panel lock time has expired, it is possible for input take place again.

The following C code example shows how to read the lock status of a connected panel.

Locked = 1 indicates that the panel is locked. Input made via the keypad and resistive touch screen is not processed. Locked = 0 indicates that the panel is not locked.

A faster way to see whether a panel is locked is via the MTCX **Panel Flags** register (see page 53 and **MtcxIsPanelLocked** code example on page 79).

```
// Get lock state of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Locked
//     [out] Points to a variable that receives the
//     lock state: 1 = locked, 0 = unlocked.
//
// Return 0 at success and -1 at failure.
int GetPanelLock(unsigned int PanelNumber, int *Locked)
{
    MTCX_PANEL_LOCK_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_KEY_SERVICE, 0, 4, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    // Note: LockBits bit x represents the lock state of panel x:
    //       bit value 0 = lock OFF, 1 = lock ON.
    *Locked = (data.LockBits & (1<<PanelNumber)) ? 1 : 0;
    return 0;
}
```

Listing 22: GetPanelLock – Reading the panel lock status

The following C code example shows how to set the lock status of a connected panel.

When Lock is 1, input via keys or the resistive touch screen is locked. It is then also not possible to determine the state of the keys by reading the key matrix. If Lock = 0, keypad and resistive touch screen input is not locked.

```
// Set lock state of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Lock
//     [in] Specifies the lock state: 1 = locked, 0 = unlocked.
//
// Return 0 at success and -1 at failure.
int SetPanelLock(unsigned int PanelNumber, int Lock)
{
    MTCX_PANEL_LOCK_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    // Note: LockBits bit x represents the lock state of panel x:
    //       bit value 0 = lock OFF, 1 = lock ON.
    //       A bit is evaluated by the MTCX only when the corresponding
    //       bit in MaskBits is set 1.
    if (Lock)
    {
        // Lock specified panel.
        data.LockBits = (1<<PanelNumber);
        data.MaskBits = (1<<PanelNumber);
    }
    else
    {
        // Unlock specified panel.
        data.LockBits = ~(1<<PanelNumber);
        data.MaskBits = (1<<PanelNumber);
    }

    // This overrides the automatic panel lock.
    if ((retvalue = MtcxWriteCommand(MTCX_CMD_KEY_SERVICE, 0, 4, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 23: SetPanelLock – Setting the panel lock status

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.4 Reading versions

The following code examples show how to read the BIOS and firmware versions installed on the PC and connected Automation Panels.

7.4.1 Reading the BIOS version

The BIOS version can be read by looking for and evaluating the B&R vendor string in the BIOS ROM.

The BIOS ROM is located in memory at the addresses F0000h through FFFFFh. The B&R vendor string is structured as follows:

```
„Bernecker + Rainer  Industrie-Elektronik Qv.yy“
„Bernecker + Rainer  Industrie-Elektronik Sv.yy“
```

Q is the BIOS code on an APC910 and PPC900 device with QM77 chipset and S is the BIOS code for a HM76 chipset, with v.yy representing the BIOS version.

The following C code example shows how to read the BIOS code and version.

```
// Read BIOS ID and version.
//
// Parameters
//   Id
//     [out] Points to a variable that receives the BIOS ID.
//   Version
//     [out] Points to a buffer that receives the
//     BIOS version string: for example "1.14".
//   Size
//     [in] Specifies the size of the version buffer in bytes.
//
// Return 0 at success and -1 at failure.
int GetBiosVersion(char *Id, char *Version, int Size)
{
    const char *BrVendorString = "Bernecker + Rainer  Industrie-Elektronik ";
    unsigned long BiosMem;

    // Search the BIOS memory block for the B&R vendor string.
    for (BiosMem = 0x000F0000; BiosMem < 0x000FFFF0; BiosMem++)
    {
        int i;

        for (i = 0; BrVendorString[i] != '\0'; i++)
        {
            if (READ_REGISTER_UCHAR(BiosMem + i) != BrVendorString[i])
                break;
        }
        if (BrVendorString[i] == '\0')
        {
            BiosMem += i;
            *Id = READ_REGISTER_UCHAR(BiosMem++);
            for (i = 0; i < Size; i++)
                *Version++ = READ_REGISTER_UCHAR(BiosMem++);
            return 0;
        }
    }

    return -1;
}
```

Listing 24: GetBiosVersion – Reading the BIOS version

7.4.2 Reading the MTCX version

It is possible to read the MTCX version from the MTCX's **Version** register (see page 12).

The following C code example shows how to read the MTCX version.

```
// Read version of MTCX.
//
// Parameters
//   Major
//     [out] Points to a variable that receives the
//     major version number: 0 to 255.
//   Minor
//     [out] Points to a variable that receives the
//     minor version number: 0 to 99.
void GetMtcxVersion(unsigned char *Major, unsigned char *Minor)
{
    MTCX_VERSION_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_VERSION_ADDR);
    *Major = reg.Major;
    *Minor = reg.Minor;
}
```

Listing 25: GetMtcxFpgaVersion – Reading the MTCX FPGA version

The definitions used can be found on page 59.

7.4.3 Reading the Display Link FPGA version

The version of the Display Link FPGA firmware can be determined from the firmware's header. It is possible to read this header using the MTCX's **Module Header** command (see page 18). It has the following structure:

```
//
// Strings used in FPGA firmware header.
//
#define APC910_FPGA_ID_LINK "LSDL"

#pragma pack(1)
//
// FPGA firmware header.
//
// Each FPGA firmware has a 16 byte header at address 0x10.
// This header contains the firmware version and update counter.
//
typedef struct
{
    char Id[4];                 // 0x10: ID "LSDL"
    char Version[4];            // 0x14: version: "vvyy", where vv is major and yy is minor
    unsigned long Counter;      // 0x18: update counter; incremented at each update
    unsigned short Reserve2;    // 0x1C: unused; 0
    unsigned short CRC;         // 0x1E: 16-bit CRC of header
} APC910_FPGA_HEADER;
#pragma pack()
```

Listing 26: APC910_FPGA_HEADER data structure – APC910 FPGA header

The following C code example shows how to read the Display Link FPGA version.

```
// Read version of Display Link FPGA.
//
// Parameters
//   Major
//     [out] Points to a variable that receives the
//     major version number: 0 to 255.
//   Minor
//     [out] Points to a variable that receives the
//     minor version number: 0 to 99.
//
// Return 0 at success and -1 at failure.
int GetLinkFpgaVersion(unsigned char *Major, unsigned char *Minor)
{
    APC910_FPGA_HEADER header;
    int retvalue;
    unsigned char ErrorCode;
    unsigned int version;

    // Read active Display Link FPGA firmware header.
    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_HEADER, 0, MTCX_DEVNUM_LINK, 0,
        &header, sizeof(header), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here.
        return -1;
    }

    // Check Display Link FPGA firmware header.
    if (strncmp(header.Id, APC910_FPGA_ID_LINK, strlen(APC910_FPGA_ID_LINK)) != 0)
    {
        // TODO: add your error handling here.
        return -1;
    }

    // Get version from Display Link FPGA firmware header.
    if (sscanf(header.Version, "%4d", &version) != 1)
    {
        // TODO: add your error handling here.
        return -1;
    }

    *Major = version >> 8;
    *Minor = version & 0xff;
    return 0;
}
```

Listing 27: GetLinkFpgaVersion – Reading the Display Link FPGA version

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.4.4 Reading the version of the scan code data

It is possible to read the scan code data version using the MTCX's **Version Info** command (see page 18).

Information:

The version of the scan code data only refers to the format of the data structure and is independent of the version that can be set in the B&R Key Editor when setting the key configuration. This version must be read directly from the key configuration data in flash memory (not included in this implementation guide).

The following C code example shows how to read the version of the scan code data.

```
// Read version of scan code data structure.
//
// Parameters
//   Major
//     [out] Points to a variable that receives the
//     major version number: 0 to 255.
//   Minor
//     [out] Points to a variable that receives the
//     minor version number: 0 to 99.
//
// Return 0 at success and -1 at failure.
//
// Remarks
//   This function returns the supported version of the
//   scan code data structure and NOT the file version!
//   The file version must be read from the Flash memory.
int GetScanCodeVersion(unsigned char *Major, unsigned char *Minor)
{
    MTCX_VERSION_INFO_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_VERSION_INFO, 0, MTCX_DEVNUM_SCAN, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here.
        return -1;
    }

    *Major = data.Major;
    *Minor = data.Minor;
    return 0;
}
```

Listing 28: GetScanCodeVersion – Reading the version of the scan code data

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.4.5 Reading the AP Link FPGA version

The version of the AP Link FPGA of a connected Automation Panel can be read using the MTCX's **Panel Version** register (see page 51).

Information:

- **This function is supported only for panels 0 to 14. The display unit of a PPC900 (panel 15) does not have an AP Link FPGA.**
- **The version of the base unit is returned for AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).**

The following C code example shows how to read the version of the AP Link FPGA.

```
// Read version of AP link FPGA.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Major
//     [out] Points to a variable that receives the
//     major version number: 0 to 255.
//   Minor
//     [out] Points to a variable that receives the
//     minor version number: 0 to 99.
//
// Return 0 at success and -1 at failure.
int GetPanelLinkFpgaVersion(unsigned int PanelNumber, unsigned char *Major, unsigned char *Minor)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_VERSION_REG reg;

        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_VERSION_ADDR);
        *Major = reg.Major;
        *Minor = reg.Minor;
    }

    // TODO: release synchronization object for panel data here

    return retvalue;
}
```

Listing 29: GetPanelLinkFpgaVersion – Reading the AP Link FPGA version

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.5 Reading hardware properties

The following code examples show how you can read the properties of the hardware being used.

7.5.1 Read device type (APC910, PPC900)

The MTCX **Hardware Info** register (see page 29) can be read to type of the device being used.

The following C code example shows how to read the device type.

```
// Get the type of the device.
//
// Return MTCX device type etc. (see MTCX_DEVICE_* defines).
int GetDeviceType()
{
    MTCX_HW_INFO_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_HW_INFO_ADDR);
    return reg.DeviceType;
}
```

Listing 30: GetDeviceType – Reading the device type (APC910, PPC900)

The definitions used can be found on page 59.

7.5.2 Reading the panel type (AP900, AP800, etc.)

The MTCX **Panel Version** register (see page 51) can be used to determine whether the panel is an AP900, AP800 or AP800 extension unit.

The following C code example shows how to read the panel type.

```
// Get the type of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//
// Return panel type and -1 at failure (e.g. if panel is not linked).
int GetPanelType(unsigned int PanelNumber)
{
    int retvalue;
    int PanelType = -1;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_VERSION_REG reg;

        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_VERSION_ADDR);
        PanelType = reg.Id;
    }

    // TODO: release synchronization object for panel data here

    return PanelType;
}
```

Listing 31: GetPanelType – Reading the panel type (AP900, AP800, etc.)

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.6 Reading device information from the PC

The following code examples show how to read B&R-specific device information (e.g. serial number or hardware revision). This also includes reading the information from optional components such as the UPS IF option.

7.6.1 Reading a PC module's device ID

A PC module's device ID corresponds to the product number used internally at B&R and can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's device ID.

```
// Get device ID of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   DeviceId
//     [out] Points to a variable that receives
//     the device ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int GetModuleDeviceId(int ModuleNumber, unsigned long *DeviceId)
{
    int retvalue;
    unsigned char ErrorCode;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x100, ModuleNumber, 0,
        DeviceId, sizeof(*DeviceId), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    return 0;
}
```

Listing 32: GetApciHwDeviceId – Reading a PC module's device ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.2 Reading a PC module's compatibility ID

A PC module's compatibility ID is used to detect hardware changes important for device-specific software and can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's compatibility ID.

```
// Get compatibility ID of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   CompatId
//     [out] Points to a variable that receives
//     the compatibility ID: 0000h to FFFFh.
//
// Return 0 at success and -1 at failure.
int GetModuleCompatId(int ModuleNumber, unsigned short *CompatId)
{
#pragma pack(1)
    struct
    {
        unsigned long VendorId;
        unsigned short CompatId;
    } data;
#pragma pack()
    int retvalue;
    unsigned char ErrorCode;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x200, ModuleNumber, 0,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    *CompatId = data.CompatId;
    return 0;
}
```

Listing 33: GetModuleCompatId – Reading a PC module's compatibility ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.3 Reading a PC module's vendor ID

The PC module's vendor ID for a B&R product is always 0 and can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's vendor ID.

```
// Get vendor ID of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   VendorId
//     [out] Points to a variable that receives
//     the vendor ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int GetModuleVendorId(int ModuleNumber, unsigned long *VendorId)
{
#pragma pack(1)
    struct
    {
        unsigned long VendorId;
        unsigned short CompatId;
    } data;
#pragma pack()
    int retvalue;
    unsigned char ErrorCode;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x200, ModuleNumber, 0,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    *VendorId = data.VendorId;
    return 0;
}
```

Listing 34: GetModuleVendorId – Reading a PC module's vendor ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.4 Reading a PC module's hardware revision

A PC module's hardware revision can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's hardware revision. Specify a variable with at least 5 characters for HwRevision.

```
// Get hardware revision of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   HwRevision
//     [out] Points to a buffer that receives the hardware revision.
//     The buffer must be large enough to hold 4 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 5.
//
// Return 0 at success and -1 at failure.
int GetModuleHwRevision(int ModuleNumber, char *HwRevision, int Size)
{
    int retvalue;
    unsigned char ErrorCode;

    if (Size < 5)
        return -1;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x300, ModuleNumber, 0,
        HwRevision, 5, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    HwRevision[4] = '\0';
    return 0;
}
```

Listing 35: GetModuleHwRevision – Reading a PC module's hardware revision

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.5 Reading a PC module's serial number

A PC module's serial number can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's serial number. Specify a variable with at least 12 characters for SerialNumber.

```
// Get serial number of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   SerialNumber
//     [out] Points to a buffer that receives the serial number.
//     The buffer must be large enough to hold 11 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 12.
//
// Return 0 at success and -1 at failure.
int GetModuleSerialNumber(int ModuleNumber, char *SerialNumber, int Size)
{
    int retvalue;
    unsigned char ErrorCode;

    if (Size < 12)
        return -1;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x400, ModuleNumber, 0,
        SerialNumber, 12, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    SerialNumber[11] = '\0';
    return 0;
}
```

Listing 36: GetModuleSerialNumber – Reading a PC module's serial number

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.6 Reading a PC module's model number

A PC module's model number can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's model number. Specify a variable with at least 41 characters for ModelNumber.

```
// Get model number of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   ModelNumber
//     [out] Points to a buffer that receives the model number.
//     The buffer must be large enough to hold 40 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 41.
//
// Return 0 at success and -1 at failure.
int GetModuleModelNumber(int ModuleNumber, char *ModelNumber, int Size)
{
    int retvalue;
    unsigned char ErrorCode;

    if (Size < 41)
        return -1;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x500, ModuleNumber, 0,
        ModelNumber, 16, &ErrorCode)) != MTCX_OKAY ||
        (retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x501, ModuleNumber, 0,
        ModelNumber+16, 16, &ErrorCode)) != MTCX_OKAY ||
        (retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x502, ModuleNumber, 0,
        ModelNumber+32, 9, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    ModelNumber[40] = '\0';
    return 0;
}
```

Listing 37: GetModuleModelNumber – Reading a PC module's model number

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.7 Reading a PC module's parent device ID

A PC module's parent device ID specifies the device ID of the PC module from which the current device is "derived" (e.g. when using a customer-specific variant of a standard device). The parent device ID can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's parent device ID. ID = FFFFh indicates that a parent device does not exist.

```
// Get parent device ID of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   DeviceId
//     [out] Points to a variable that receives
//     the device ID: 00000000h to FFFFFFFFh (= no parent).
//
// Return 0 at success and -1 at failure.
int GetModuleParentDeviceId(int ModuleNumber, unsigned long *DeviceId)
{
#pragma pack(1)
     struct
     {
            unsigned long DeviceId;
            unsigned short CompatId;
     } data;
#pragma pack()
     int retvalue;
     unsigned char ErrorCode;

     if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x700, ModuleNumber, 0,
         &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
     {
         // TODO: add your error handling here
         return -1;
     }
     *DeviceId = data.DeviceId;
     return 0;
}
```

Listing 38: GetModuleParentDeviceID – Reading a PC module's parent device ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.6.8 Reading a PC module's parent compatibility ID

A PC module's parent compatibility ID specifies the compatibility ID of the PC module from with the current device is "derived" (e.g. when using a custom variant of a standard device). The parent compatibility ID can be read using the MTCX's **Module Info** command (see page 29).

The following C code example shows how to read a PC module's compatibility ID. ID = FFFFh indicates that a parent device does not exist.

```
// Get parent compatibility ID of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   CompatId
//     [out] Points to a variable that receives
//     the compatibility ID: 0000h to FFFFh (= no parent).
//
// Return 0 at success and -1 at failure.
int GetModuleParentCompatId(int ModuleNumber, unsigned short *CompatId)
{
#pragma pack(1)
    struct
    {
        unsigned long DeviceId;
        unsigned short CompatId;
    } data;
#pragma pack()
    int retvalue;
    unsigned char ErrorCode;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_INFO, 0x700, ModuleNumber, 0,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    *CompatId = data.CompatId;
    return 0;
}
```

Listing 39: GetModuleParentCompatId – Reading a PC module's parent compatibility ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.7 Reading device information from a panel

The following code examples show how to read B&R-specific device information (e.g. serial number or hardware revision) from a connected Automation Panel.

The information read in the following code examples is part of the Automation Panel's factory settings and is provided in the following data structure (the start offset in the factory settings is 80h):

```
#define BR_INT_DATA_PANEL_HEADER_ID "\xff\x55\xaa\xff\x00\x00\xff\xff"
#define BR_INT_DATA_PANEL_HEADER_SIZE 8

#pragma pack(1)

typedef struct
{
    unsigned char   Header[8];              // hex: FF 55 AA FF 00 00 FF
    unsigned char   Reserve1[2];
    unsigned long   DeviceId;               // old model number (big-endian format!)
    unsigned short  CompatId;               // compatibility ID (big-endian format!)
    unsigned long   VendorId;               // B&R = 0 (big-endian format!)
    char            HwRevision[5];          // string with 4 char's
    char            SerialNumber[12];       // string with 11 char's
    char            ModelNumber[41];        // string with 40 char's
    unsigned char   Reserve2[2];
} BR_INT_DATA_PANEL; // start offset 80h
```

Listing 40: BR_INT_DATA_PANEL data structure - Panel device information

Information:

Factory settings of the display unit of a PPC900 (panel 15) can be read with the Module Info command (see page 29 or code examples on page 93). AP800 Extension Units do not have their own factory settings.

This data can be read using the following function:

```
// Read internal data of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Data
//     [out] Points to a buffer that receives the B&R internal data.
//
// Return 0 at success and -1 at failure.
int ReadPanelIntData(unsigned int PanelNumber, BR_INT_DATA_PANEL *Data)
{
    int retvalue;
    MTCX_PANEL_VERSION_REG reg;
    int i;

    if (PanelNumber > 15)
        return -1;
    if (sizeof(BR_INT_DATA_PANEL) % MTCX_CMD_DATA_SIZE) // safety check
        return -1;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_VERSION_ADDR);

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
        return -1;
```

```
    // Don't send Flash command to built-in display because MTCX does not
    // respond on some platforms...
    if (reg.Id == MTCX_VER_ID_DISP)
       return -1;

    // Read B&R internal data from flash memory.
    for (i = 0; i < sizeof(BR_INT_DATA_PANEL); i += MTCX_CMD_DATA_SIZE)
    {
        unsigned char ErrorCode;

        retvalue = MtcxReadCommand(MTCX_CMD_FLASH_SERVICE, (0x80 + i) / MTCX_CMD_DATA_SIZE,
            PanelNumber, MTCX_DEV_PANELSET, (unsigned char*)Data + i, MTCX_CMD_DATA_SIZE,
&ErrorCode);
        if (retvalue != MTCX_OKAY)
        {
            // TODO: add your error handling here
            return -1;
        }
    }

    if (memcmp(Data->Header, BR_INT_DATA_PANEL_HEADER_ID, BR_INT_DATA_PANEL_HEADER_SIZE) != 0)
    {
        // the flash memory does not contain valid data
        return -1;
    }

    return 0;
}
```

Listing 41: ReadPanelIntData – Reading internal B&R panel data

Information:

Reading the Factory Settings data takes longer than another MTCX commands due to the flash / EEPROM access required.

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.7.1 Reading a panel's device ID

The device ID of a connected Automation Panel corresponds to its product number used internally at B&R and can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's device ID.

```
// Get device ID of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   DeviceId
//     [out] Points to a variable that receives
//     the device ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int GetPanelDeviceId(unsigned int PanelNumber, unsigned long *DeviceId)
{
    BR_INT_DATA_PANEL data;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    *DeviceId = swapl(data.DeviceId);
    return 0;
}
```

Listing 42: GetPanelDeviceId – Reading a panel's device ID

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.
The **swapl** function can be found on page 58.

7.7.2 Reading a panel's compatibility ID

The compatibility ID of a connected Automation Panel can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's compatibility ID.

```
// Get compatibility ID of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   CompatId
//     [out] Points to a variable that receives
//     the compatibility ID: 0000h to FFFFh.
//
// Return 0 at success and -1 at failure.
int GetPanelCompatId(unsigned int PanelNumber, unsigned short *CompatId)
{
    BR_INT_DATA_PANEL data;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    *CompatId = swapw(data.CompatId);
    return 0;
}
```

Listing 43: GetPanelCompatId – Reading a panel's compatibility ID

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.
The **swapw** function can be found on page 58.

7.7.3 Reading a panel's vendor ID

The vendor ID of a connected Automation Panel can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's vendor ID.

```
// Get vendor ID of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   VendorId
//     [out] Points to a variable that receives
//     the vendor ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int GetPanelVendorId(unsigned int PanelNumber, unsigned long *VendorId)
{
    BR_INT_DATA_PANEL data;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    *VendorId = swapl(data.VendorId);
    return 0;
}
```

Listing 44: GetPanelVendorId – Reading a panel's vendor ID

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.
The **swapl** function can be found on page 58.

7.7.4 Reading a panel's hardware revision

The hardware revision of a connected Automation Panel can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's hardware revision. Specify a variable with at least 5 characters for HwRevision.

```
// Get hardware revision of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   HwRevision
//     [out] Points to a buffer that receives the hardware revision.
//     The buffer must be large enough to hold 4 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 5.
//
// Return 0 at success and -1 at failure.
int GetPanelHwRevision(unsigned int PanelNumber, char *HwRevision, int Size)
{
    BR_INT_DATA_PANEL data;

    if (Size < 5)
        return -1;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    strncpy(HwRevision, data.HwRevision, Size);
    HwRevision[4] = '\0';
    return 0;
}
```

Listing 45: GetPanelHardwareRev – Reading a panel's hardware revision

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.

7.7.5 Reading a panel's serial number

The serial number of a connected Automation Panel can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's serial number. Specify a variable with at least 12 characters for SerialNumber.

```
// Get serial number of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   HwRevision
//     [out] Points to a buffer that receives the hardware revision.
//     The buffer must be large enough to hold 11 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 12.
//
// Return 0 at success and -1 at failure.
int GetPanelSerialNumber(unsigned int PanelNumber, char *SerialNumber, int Size)
{
    BR_INT_DATA_PANEL data;

    if (Size < 12)
        return -1;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    strncpy(SerialNumber, data.SerialNumber, 12);
    SerialNumber[11] = '\0';
    return 0;
}
```

Listing 46: GetPanelSerialNumber – read serial number of a panel

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.

7.7.6 Reading a panel's model number

The model number of a connected Automation Panel can be read from the factory settings of the panel using the MTCX's **Flash / EEPROM Service** command (see page 24).

The following C code example shows how to read a connected Automation Panel's model number. Specify a variable with at least 41 characters for ModelNumber.

```
// Get model number of panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   ModelNumber
//     [out] Points to a buffer that receives the model number.
//     The buffer must be large enough to hold 40 characters
//     and a 0-byte.
//   Size
//     [in] Specifies the size of the buffer in bytes: minimum is 41.
//
// Return 0 at success and -1 at failure.
int GetPanelModelNumber(unsigned int PanelNumber, char *ModelNumber, int Size)
{
    BR_INT_DATA_PANEL data;

   if (Size < 41)
        return -1;

    if (ReadPanelIntData(PanelNumber, &data) != 0)
        return -1;

    strncpy(ModelNumber, data.ModelNumber, 41);
    ModelNumber[40] = '\0';
    return 0;
}
```

Listing 47: GetPanelModelNumber – Reading a panel's model number

The definitions used can be found on page 59.
The **ReadPanelIntData** function can be found on page 101.

7.8 Reading statistics

It is possible to read the following statistics:

- A PC module's operating hours and power-on cycles
- Operating hours and power-on cycles (of the backlight) of the display unit of a PPC900 or an Automation Panel
- Operating hours and power-on cycles of a fan kit
- CMOS battery status

Statistics on the PC are managed by the MTCX. On an Automation Panel, the statistics are updated by the panel itself.

The following code examples show how to read statistics (e.g. hours of operation) of PC modules and connected Automation Panels.

7.8.1 Reading a PC module's operating hours and power-on cycles

A PC module's operating hours and power-on cycles can be read using the MTCX's **Module Statistics** command (see page 32).

The following C code example shows how to read a PC module's operating hours and power-on cycles.

```
// Read statistics values of a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   PowerOnHours
//     [out] Points to a variable that receives the
//     power-on hours: 0 to 4194303.
//   PowerOnCycles
//     [out] Points to a variable that receives the
//     power-on cycles: 0 to 16777212.
//
// Return 0 at success and -1 at failure.
int GetModuleStatistics(int ModuleNumber, unsigned long *PowerOnHours, unsigned long
*PowerOnCycles)
{
    MTCX_MODULE_STATISTICS_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_STATISTICS, 0x100, ModuleNumber, 0,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *PowerOnHours = data.PowerOnHours / 4;
    *PowerOnCycles = data.PowerOnCycles;
}
```

Listing 48: GetModuleStatistics – Reading a PC module's operating hours and power-on cycles

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.8.2 Reading the display unit's operating hours and power-on cycles

A display unit's operating hours and power-on cycles (of the backlight) can be read using the MTCX's **Module Statistics** command (see page 32).

The following C code example shows how to read a display unit's operating hours and power-on cycles.

```
// Read statistics values of the display unit (PPC900 only).
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 1.
//   BacklightOnHours
//     [out] Points to a variable that receives the
//     backlight-on hours: 0 to 4194303.
//   BacklightOnCycles
//     [out] Points to a variable that receives the
//     backlight-on cycles: 0 to 16777212.
//
// Return 0 at success and -1 at failure.
int GetDisplayStatistics(int ModuleNumber, unsigned long *BacklightOnHours, unsigned long
*BacklightOnCycles)
{
    MTCX_DISPLAY_STATISTICS_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_STATISTICS, 0x100, ModuleNumber, 1,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *BacklightOnHours = data.BacklightOnHours / 4;
    *BacklightOnCycles = data.BacklightOnCycles;
    return 0;
}
```

Listing 49: GetDisplayStatistics – Reading a display unit's operating hours and power-on cycles

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.8.3 Reading the number of operating hours and power-on cycles from the PC fans.

The PC fan's operating hours and power-on cycles can be read using the MTCX's **Module Statistics** command (see page 32).

Each PC module can have up to four fans. This command will return an error for fans that do not exist.

Note: Normally all fans of a module return the same statistics values – unless in case of a fan error (for example standstill).

The following C code example shows how to read a PC fan's operating hours and power-on cycles.

```
// Read statistics values of the fans on a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 8 or 12 (PPC900 only).
//   FanNumber
//     [in] Specifies the fan number: 0 to 3.
//   FanOnHours
//     [out] Points to a variable that receives the
//     fan operating hours: 0 to 4194303.
//   FanOnCycles
//     [out] Points to a variable that receives the
//     fan turn-on cycles: 0 to 16777212.
//
// Return 0 at success and -1 at failure.
int GetFanStatistics(int ModuleNumber, int FanNumber, unsigned long *FanOnHours, unsigned long
*FanOnCycles)
{
    MTCX_FAN_STATISTICS_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_STATISTICS, 0x200, ModuleNumber, FanNumber,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *FanOnHours = data.FanOnHours / 4;
    *FanOnCycles = data.FanOnCycles;
    return 0;
}
```

Listing 50: GetFanStatistics – Reads the number of operating hours and power-on cycles from the PC fan.

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.8.4 Reading "On Battery" hours and cycles from the UPS.

The UPS "On Battery" hours and cycles can be read using the MTCX's **Module Statistics** command (see page 32).

The following C code example shows how to read the UPS "On Battery" hours and cycles.

```
// Read statistics values of the UPS (battery).
//
// Parameters
//   OnBatHours
//     [out] Points to a variable that receives the
//     "on battery" hours: 0 to 4194303.
//   OnBatCycles
//     [out] Points to a variable that receives the
//     "on battery" cycles: 0 to 16777212.
//
// Return 0 at success and -1 at failure.
int GetUpsStatistics(unsigned long *OnBatHours, unsigned long *OnBatCycles)
{
    MTCX_UPS_STATISTICS_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_STATISTICS, 0x100, MTCX_DEVNUM_APC910_IF1, 3,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *OnBatHours = data.OnBatHours / 4;
    *OnBatCycles = data.OnBatCycles;
    return 0;
}
```

Listing 51: GetUpsStatistics – Reads "On Battery" hours and cycles from the UPS.

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.8.5 Reading a panel's operating hours and power-on cycles

A connected Automation Panel's operating hours and power-on cycles (of the backlight) can be read with the MTCX's **Statistics Info** command (see page 25).

Information:

- **This function is supported only for panels 0 to 14. The statistics values of the display unit of a PPC900 (panel 15) can be read with the Module Statistics command (see page 32 and code example on page 109).**
- **This function is not supported for AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).**

The following C code example shows how to read a connected Automation Panel's number of operating hours and power-on cycles.

```
// Get statistics values of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   BacklightOnHours
//     [out] Points to a variable that receives the
//     backlight-on hours: 0 to 65535.
//   BacklightOnCycles
//     [out] Points to a variable that receives the
//     backlight-on cycles: 0 to 65535.
//
// Return 0 at success and -1 at failure.
int GetPanelStatistics(unsigned int PanelNumber, unsigned short *BacklightOnHours, unsigned short
*BacklightOnCycles)
{
    MTCX_PANEL_STATISTICS_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    if ((retvalue = MtcxReadCommand(MTCX_CMD_STATISTICS_INFO, 0, PanelNumber, MTCX_DEV_PANEL,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *BacklightOnHours = data.BacklightOnHours;
    *BacklightOnCycles = data.BacklightOnCycles;
    return 0;
}
```

Listing 52: GetPanelStatistics – Reading a panel's operating hours and power-on cycles

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.8.6 Reading the CMOS battery status

The CMOS battery status can be read via the MTCX **Baseboard Specials** register (see page 45). The battery status is provided as an integer value: 0 = good, 3 = bad

The following C code example shows how to read the CMOS battery status.

```
// Read CMOS battery state.
//
// Parameters
//   State
//     [out] Points to a variable that receives the
//     state of the CMOS battery:
//     0 = good,
//     2 = not available (older platforms only),
//     3 = bad.
void GetBatteryState(int* State)
{
    MTCX_BASEBOARD_SPECIALS_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_BASEBOARD_SPECIALS_ADDR);
    *State = reg.BatState;
}
```

Listing 53: GetBatteryState – Reading the CMOS battery status

The definitions used can be found on page 59.

7.9 Reading temperature values

It is possible to read the following temperature values:

- Temperature values of a PC module
- Temperature of a connected Automation Panel
- Temperature of the UPS battery (see page 154)

All temperatures are collected and provided by the MTCX.

The following code examples show how to read the temperatures of a PC module and connected Automation Panels.

7.9.1 Reading a PC module's temperature values

A PC module's temperatures can be read using the MTCX's **Module Temperature** command (see page 31).

Each PC module can have up to four temperature sensors. This command will return an error for temperature sensors that do not exist.

The following C code example shows how to read a PC module's temperatures.

```
// Read temperature from a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   SensorNumber
//     [in] Specifies the sensor number: 0 to 3.
//   Temp
//     [out] Points to a variable that receives
//     the temperature in degrees Celsius.
//
// Return 0 at success and -1 at failure.
//
// Remarks: See to the hardware manual for a
// description of the temperature sensors.
int GetModuleTemp(int ModuleNumber, int SensorNumber, float *Temp)
{
    MTCX_MODULE_TEMPERATURE_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_TEMPERATURE, 0, ModuleNumber, SensorNumber,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *Temp = (float)(data.Temp >> 6) / 4;
    return 0;
}
```

Listing 54: GetModuleTemp – Reading a PC module's temperature values

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.9.2 Reading a panel's temperature

The temperature of a linked Automation Panel (ambient temperature of the display) can be read via the MTCX **Panel TempFan** register (see page 51).

Information:

- **This function is supported only for panels 0 to 14. The temperature of the display unit of a PPC900 (panel 15) is returned as 0 always with this function and can be read with the Module Temperature command (see page 31 and code example on page 115109).**
- **No valid temperature provided for AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).**

The following C code example shows how to read the temperature of a connected Automation Panel.

For performance reasons, only the lowest byte of the **Panel TempFan** register is accessed in **MtcxGetPanelTemp**.

```
// Read panel temperature.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Temp
//     [out] Points to variable that receives the
//     panel temperature in degrees Celsius: 0 to 127.
//
// Return 0 at success and -1 at failure.
int GetPanelTemp(unsigned int PanelNumber, char *Temp)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        *Temp = (char)READ_PORT_UCHAR(MTCX_PANEL_TEMPFAN_ADDR); // read byte 0 only

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
        return -1;
    return 0;
}
```

Listing 55: GetPanelTemp – Reading a panel's temperature

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.10 Reading fan speed

The following code examples show how to read the fan speeds.

7.10.1 Reading a PC module's fan speed

A PC module's fan speed can be read using the MTCX's **Module Fan** command (see page 38).

Each PC module can have up to four fans. This command will return an error for fans that do not exist.

The following C code example shows how to read a PC module's fan speed.

```
// Read fan speed from a module.
//
// Parameters
//   ModuleNumber
//     [in] Specifies the module number: 0 to 15.
//   FanNumber
//     [in] Specifies the fan number: 0 to 3.
//   FanSpeed
//     [out] Points to a variable that receives
//     the fan speed in rpm.
//
// Return 0 at success and -1 at failure.
int GetModuleFanSpeed(int ModuleNumber, int FanNumber, unsigned short *FanSpeed)
{
    MTCX_MODULE_FAN_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_FAN, 0, ModuleNumber, FanNumber,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *FanSpeed = data.FanSpeed;
    return 0;
}
```

Listing 56: GetModuleFanSpeed – Reading a PC module's fan speed

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.10.2 Reading a panel's fan speed

The fan RPMs of a linked Automation Panel can be read via the MTCX **Panel TempFan** register (see page 51).

Information:

- **This function is supported only for panels 0 to 14. The fan speed of the display unit of a PPC900 (panel 15) is returned as 0 always with this function.**
- **No valid fan speed provided for AP800 extension units. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).**

The following C code example shows how to read the fan speed of a connected Automation Panel.

```
// Read panel fan speed.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   FanSpeed
//     [out] Points to a variable that receives the
//     fan speed in rpm: 0 to 16380.
//
// Return 0 at success and -1 at failure.
int GetPanelFanSpeed(unsigned int PanelNumber, unsigned short *FanSpeed)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_TEMPFAN_REG reg;

        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_TEMPFAN_ADDR);
        *FanSpeed = (unsigned short)reg.FanSpeed * 4;
    }

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
        return -1;
    return 0;
}
```

Listing 57: GetPanelFanSpeed – Reading a panel's fan speed

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.11 Reading voltage values

It is possible to read the following voltage values:

- Battery voltage (Vbat)
- Input voltage (Vin)
- Current system power

All values are collected and provided by the MTCX.

The following code examples show how to read these voltage values.

7.11.1 Reading a PC module's voltage values

A PC module's voltage values can be read using the MTCX's **Module Voltage** command (see page 36).

Note: Voltage values can currently only be read for module number 0 (system unit).

Information:

Reading valid input voltage (Vin) and system power (Wcpu) requires MTCX version 1.01 or higher on APC910 and MTCX version 1.02 or higher on PPC900. Older MTCX versions return invalid values. Vin is always 0.

The following C code example shows how to read voltage values.

```
// Get voltages.
//
// Parameters
//   Vbat
//     [out] Points to a variable that receives the
//     battery voltage in Volt.
//     Note: Vbat returns the minimum value that
//           can be measured when VbatState = bad.
//   VbatState
//     [out] Points to a variable that receives the
//     battery voltage status: 0 = good, 1 = bad.
//   Vin
//     [out] Points to a variable that receives the
//     input voltage in Volt.
//   Wcpu
//     [out] Points to a variable that receives the
//     system power consumption in Watt.
//
// Return 0 at success and -1 at failure.
int GetVoltages(float *Vbat, int *VbatState, float *Vin, float *Wcpu)
{
    MTCX_MODULE_VOLTAGE_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    if ((retvalue = MtcxReadCommand(MTCX_CMD_MODULE_VOLTAGE, 0, 0, 0,
        &data, sizeof(data), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *Vbat = (float)data.Vbat.Value / 100;
    *VbatState = data.Vbat.State;
    *Vin = (float)data.Vin.Value / 10;
    *Wcpu = (float)data.Wcpu / 100;
    return 0;
}
```

Listing 58: GetVoltages – Reading voltage values

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.

7.12 Display functions

One or more Automation Panels can be linked to an APC910 and PPC900. The following functions can be executed for an Automation Panel's display:

- Reading/setting the display brightness
- Reading/setting the display equalizer value

The following functions can be executed for the display unit of a PPC900:

- Reading/setting the display brightness

Display functions are executed by the MTCX.

The following code examples show how you can use the display functions.

7.12.1 Reading/setting the display brightness

The display brightness of the display unit of a PPC900 and a connected Automation Panel (backlighting) can be read and manually set using the MTCX **Display Service** command (see page 27).

The brightness setting is only in effect until the system is restarted. If the brightness should be in effect the next time the system is restarted, it must be stored in the application and set again after the next system start.

The following C code example shows how to read the display brightness.

```
// Get display brightness (backlight).
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Percent
//     [out] Points to a variable that receives the
//     brightness in percent: 0 to 100.
//
// Return 0 at success and -1 at failure.
int GetDisplayBrightness(unsigned int PanelNumber, unsigned char *Percent)
{
    MTCX_DISPLAY_SERVICE_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    // Read display settings.
    retvalue = MtcxReadCommand(MTCX_CMD_DISPLAY_SERVICE, 0x00, PanelNumber, MTCX_DEV_PANEL,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    // Brightness not supported?
    if (data.Brightness == 255)
        return -1;

    *Percent = data.Brightness;
    return 0;
}
```

Listing 59: GetDisplayBrightness – Reading the display brightness

The following C code example shows how the display brightness can be set (even if the panel is not connected).

```
// Set display brightness (backlight).
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Percent
//     [in] Specifies the brightness in percent: 0 to 100.
//
// Return 0 at success and -1 at failure.
int SetDisplayBrightness(unsigned int PanelNumber, unsigned char Percent)
{
    MTCX_DISPLAY_SERVICE_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    if (Percent > 100)
        return -1;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    // Read display settings.
    retvalue = MtcxReadCommand(MTCX_CMD_DISPLAY_SERVICE, 0x00, PanelNumber, MTCX_DEV_PANEL,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    // Brightness not supported?
    if (data.Brightness == 255)
        return -1;

    // Write display settings.
    data.Brightness = Percent;
    retvalue = MtcxWriteCommand(MTCX_CMD_DISPLAY_SERVICE, 0x00, PanelNumber, MTCX_DEV_PANEL,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 60: SetDisplayBrightness – Setting the display brightness

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.12.2 Reading/setting the equalizer

A connected Automation Panel's equalizer can be read and set using the MTCX's **Panel Flags** register (see page 53).

The equalizer is integrated into Automation Panel devices and adapts the DVI signal to various cable lengths. The equalizer value is automatically calculated based on the cable length. It is possible to set a different equalizer value in order to obtain the best possible display quality (e.g. in case of low-quality cables or poor DVI signal quality).

An equalizer value set by the user is only in effect until the system is restarted. If the equalizer value should be in effect the next time the system is restarted, it must be stored in the application and set again after the next system start.

Information:

The display unit of a PPC900 (panel 15) and AP800 extension units do not support equalizers. You can tell if you are dealing with an AP800 extension unit by checking the Panel Version register (see page 51 and the code example on page 92).

The following C code example shows how to read a display's equalizer value.

```
// Get display equalizer value.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   EqualizerMode
//     [out] Points to a variable that receives the
//     equalizer mode: 0 = automatic or 1 = user specific.
//   EqualizerValue
//     [out] Points to a variable that receives the
//     equalizer value: 0 (strong) to 15 (weak).
//     If EqualizerMode is 0, this is the automatic
//     equalizer value. If EqualizerMode is 1, this
//     is the user specified equalizer value.
//
// Return 0 at success and -1 at failure.
//
// Remarks
//   AP900: Requires SDL firmware version 1.04 or higher.
int GetDisplayEqualizer(unsigned int PanelNumber, int *EqualizerMode, unsigned char
*EqualizerValue)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG reg;

        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_FLAGS_ADDR);
        if (reg.EqualizerSupport)
        {
            *EqualizerMode = reg.EqualizerMode;
            if (reg.EqualizerMode)
                *EqualizerValue = reg.EqualizerUser;
            else
                *EqualizerValue = reg.EqualizerAuto;
            WRITE_PORT_ULONG(MTCX_PANEL_FLAGS_ADDR, *(unsigned long*)&reg);
        }
        else
            retvalue = MTCX_FAIL_NOSUPPORT;
    }

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    return 0;
}
```

Listing 61: GetDisplayEqualizer – Reading a display's equalizer value

The following C code example shows how to set a display's equalizer value.

```
// Set display equalizer value.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   EqualizerMode
//     [in] Specifies the equalizer mode:
//     0 = automatic or 1 = user specific.
//   EqualizerValue
//     [in] Specifies the equalizer value: 0 (strong) to 15 (weak).
//     If EqualizerMode is 1, this value overrides the
//     automatic equalizer value. If EqualizerMode is 0,
//     this value is ignored and the automatic equalizer
//     value is used.
//
// Return 0 at success and -1 at failure.
//
// Remarks
//   AP900: Requires firmware version 1.04.
int SetDisplayEqualizer(unsigned int PanelNumber, int EqualizerMode, unsigned char EqualizerVal-
ue)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        MTCX_PANEL_FLAGS_REG reg;

        *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_PANEL_FLAGS_ADDR);
        if (reg.EqualizerSupport)
        {
            reg.EqualizerMode = EqualizerMode;
                   if (reg.EqualizerMode)
                reg.EqualizerUser = EqualizerValue;
            WRITE_PORT_ULONG(MTCX_PANEL_FLAGS_ADDR, *(unsigned long*)&reg);
        }
        else
            retvalue = MTCX_FAIL_NOSUPPORT;
    }

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    return 0;
}
```

Listing 62: SetDisplayEqualizer – Setting a display's equalizer value

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.13 Keypad functions

One or more Automation Panels can be linked to an APC910 and PPC900. Each Automation Panel and the display unit of a PPC900 can have up to 128 matrix keys, which can be operated in 4 key levels and for which you can perform the following functions:

- Reading the number of keys
- Reading the key matrix
- Reading the key switches
- Locking/unlocking scan codes
- Reading the status of the key configuration
- Reading the key layer mode
- Reading and setting the number of key layers

Key functions are executed by the MTCX.

The following code examples show how you can operate and evaluate the matrix keys.

7.13.1 Reading the number of keys

The number of keys a panel supports can be read via the MTCX **Panel Specials** register (see page 52).

Information:

For technical reasons, all B&R devices that are able to support keys currently report the number of keys as 128. This does not necessarily mean that the device actually has 128 keys. There may in fact be fewer or no keys at all present on the device. It can only be assumed that the device does not support keys if the number of keys returned is 0.

The following C code example shows how to read the number of supported keys.

For performance reasons, only the higher byte of the **Panel Specials** register is accessed.

```
// Get supported key count of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   KeyCount
//     [out] Points to variable that receives the
//     maximum number of supported matrix keys: 0 to 128.
//
// Return 0 at success and -1 at failure.
int GetKeyCount(unsigned int PanelNumber, unsigned char *KeyCount)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        *KeyCount = READ_PORT_UCHAR(MTCX_PANEL_SPECIALS_ADDR + 2); // read byte 2 only

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 63: GetKeyCount – Reading the number of keys

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.13.2 Reading the key matrix

The key states of the display unit of a PPC900 and a linked Automation Panel can be read via the MTCX **Panel Key Matrix** registers (see page 54).

Note: Key states can also be read if the system doesn't have a valid key configuration.

The following C code example shows how to read the status of a connected panel's key matrix.

```
// Read key matrix of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   KeyMatrix
//     [out] Points to a buffer that receives the
//     key matrix: 1 bit per key; 0 = key released, 1 = key pressed.
//   Size
//     [in] Specifies the size of the buffer in bytes: 1 to 16.
//
// Return 0 at success and -1 at failure.
int GetKeyMatrix(unsigned int PanelNumber, unsigned char *KeyMatrix, int Size)
{
    int retvalue;

    if (Size < 1 || Size > 16)
        return -1;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
    {
        int Port = MTCX_PANEL_KEY_MATRIX0_ADDR;

        while (Size-- > 0)
            *KeyMatrix++ = ~READ_PORT_UCHAR(Port++); // make pressed keys 1 instead of 0
    }

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    return 0;
}
```

Listing 64: GetKeyMatrix – Reading the status of the key matrix

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.13.3 Reading the key switches

The states of a panel's key switches can be read via the MTCX **Panel Specials** register (see page 52).

Key switch states can only be read when the system has a valid key configuration. A key configuration can be created with the B&R Key Editor and loaded onto the device.

Information:

The key configuration must be created with B&R Key Editor version 2.50 or higher.

The following C code example shows how to read the status of a connected panel's key switches.

```
// Read key switch states of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   KeySwitches
//     [out] Points to a variable that receives the
//     key switches: 00h to FFh.
//
// Return 0 at success and -1 at failure.
int GetKeySwitches(unsigned int PanelNumber, unsigned char *KeySwitches)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        // make closed contacts 1 instead of 0
        *KeySwitches = ~READ_PORT_UCHAR(MTCX_PANEL_SPECIALS_ADDR + 1); // read byte 1 only

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    return 0;
}
```

Listing 65: GetKeySwitches – Reading the status of key switches

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.13.4 Read/set scan code lock for matrix keys

The scan code lock of a panel's matrix keys can be read and set with the MTCX **Key Service** command (see page 22).

This function can be used to lock the scan codes, for example when the operating system should not respond to keystrokes while the key matrix is being read for testing purposes.

The following C code example shows how to check whether the scan codes for a panel's matrix keys are locked or unlocked.

A faster way to read whether a panel's scan codes are locked is via the MTCX **Panel Flags** register (see page 53 and **MtcxAreScanCodesLocked** code example on page 79).

```
// Get scan code lock state of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Locked
//     [out] Points to a variable that receives the
//     lock state: 1 = locked, 0 = unlocked.
//
// Return 0 at success and -1 at failure.
int GetScanCodeLock(unsigned int PanelNumber, int *Locked)
{
    MTCX_SCANCODE_LOCK_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    // Read lock bits.
    retvalue = MtcxReadCommand(MTCX_CMD_KEY_SERVICE, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *Locked = (data.LockBits & (1<<PanelNumber)) ? 1 : 0;
    return 0;
}
```

Listing 66: GetScanCodeLock – Reading the status of a scan code lock

The following C code example shows how to lock and unlock scan codes for a panel's matrix keys.

If Lock = 0, the scan codes are unlocked and the assigned key codes are sent to the operating system whenever a key is pressed. If Lock does not equal 0, the scan codes are locked and the operating system does not receive any key codes. In this case, it is only possible to read the state of the keys.

```
// Set scan code lock state of a panel.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Lock
//     [in] Specifies the lock state: 1 = locked, 0 = unlocked.
//
// Return 0 at success and -1 at failure.
int SetScanCodeLock(unsigned int PanelNumber, int Lock)
{
    MTCX_SCANCODE_LOCK_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    // Read, modify, write scan code lock bits.
    retvalue = MtcxReadCommand(MTCX_CMD_KEY_SERVICE, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
    if (Lock)
        data.LockBits |= (1<<PanelNumber);
    else
        data.LockBits &= ~(1<<PanelNumber);
    retvalue = MtcxWriteCommand(MTCX_CMD_KEY_SERVICE, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 67: SetScanCodeLock – Setting the status of a scan code lock

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.13.5 Reading the status of the key configuration

The status of the key configuration in the flash memory (valid/invalid) can be determined by reading the MTCX **Baseboard KeyState** register (see page 44). A key configuration can be created with the B&R Key Editor and loaded onto the device.

The following C code example shows how to read the status of the key configuration.

```
// Read the state of the key configuration.
//
// Parameters
//   Valid
//     [out] Points to a variable that receives the
//     state of the key configuration:
//     1 indicates the data is valid,
//     0 indicates the data is not valid (for example has a
//       wrong checksum or no data is programmed into the Flash memory).
void GetKeyCfgState(int *Valid)
{
    MTCX_BASEBOARD_KEYSTATE_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_BASEBOARD_KEYSTATE_ADDR);
    *Valid = reg.Status ? 1 : 0;
}
```

Listing 68: GetKeyCfgState – Reading the status of a key configuration

The definitions used can be found on page 59.

7.13.6 Reading the key layer mode

The mode of the key level can be read via the MTCX **Baseboard KeyState** register (see page 44).

The keys on a B&R device can be operated on four different layers, which means each key can send up to four different key codes. The layer can be switched with special layer keys. These layer keys and the mode for switching the key layer are configured using the B&R Key Editor.

The following C code example shows how to read the key layer mode.

The following modes are possible:
0 = Shift mode: A key layer is only active as long as the corresponding layer key is being pressed.
1 = Toggle mode: The key layer is switched each time a layer key is pressed.
2 = One-shot mode: The key layer remains switched after pressing a layer key only until another key is pressed.

```
// Get key layer mode.
//
// Parameters
//   KeyLayerMode
//     [out] Points to a variable that receives the
//     key layer mode: 0 = shift mode, 1 = toggle mode,
//     2 = one-shot mode.
void GetKeyLayerMode(unsigned char *KeyLayerMode)
{
    MTCX_BASEBOARD_KEYSTATE_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_BASEBOARD_KEYSTATE_ADDR);
    *KeyLayerMode = reg.Mode;
}
```

Listing 69: GetKeyLayerMode – Reading the key layer mode

The definitions used can be found on page 59.

7.13.7 Reading/setting the key layer

The current key layer can be read via the MTCX **Baseboard KeyState** register (see page 44).

The keys on a B&R device can be operated on four different layers, which means each key can send up to four different key codes. The layer can be switched with special layer keys. These are configured with the B&R Key Editor.

The following C code example shows how to read the current key layer.

```
// Get key layer.
//
// Parameters
//   KeyLayer
//     [out] Points to a variable that receives the
//     key layer: 0 to 3.
void GetKeyLayer(unsigned char *KeyLayer)
{
    MTCX_BASEBOARD_KEYSTATE_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_BASEBOARD_KEYSTATE_ADDR);
    *KeyLayer = reg.Layer;
}
```

Listing 70: GetKeyLayer – Reading the key layer

The following C code example shows how to set the key layer.

Information:

The key layer is only set if a valid key configuration is present on the device. Nevertheless, keep in mind that this command does not return an error (see page 133)!

```
// Set key layer.
//
// Parameters
//   KeyLayer
//     [in] Specifies the key layer: 0 to 3.
//
// Return 0 at success and -1 at failure.
//
// Remarks
//   Requires a valid key configuration!
int SetKeyLayer(unsigned char KeyLayer)
{
    MTCX_KEY_LAYER_DATA data;
    unsigned char ErrorCode;
    int retvalue;

    if (KeyLayer > 3)
        return -1;

    memset(&data, 0x00, sizeof(data));
    data.Layer = KeyLayer;

    retvalue = MtcxWriteCommand(MTCX_CMD_KEY_LAYER, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 71: SetKeyLayer – Setting the key layer

The definitions used can be found on page 59.
The **MtcxWriteCommand** function can be found on page 74.

7.14 LED functions

One or more Automation Panels can be linked to an APC910 and PPC900. As with the matrix keys, each Automation Panel and the display unit of a PPC900 can have up to 128 LEDs, which can be controlled on 4 levels and for which you can perform the following functions:

- Reading the number of LEDs
- Reading/setting the LED matrix
- Reading/setting individual LEDs
- Reading/setting the Run LED

LED functions are executed by the MTCX.

The following code examples show how to operate and evaluate the matrix LEDs. The following data types are required for this:

```
typedef unsigned char MTCX_LAYER_LED_DATA[16]; // 64 LED states, 2 bits for each LED

typedef struct
{
    unsigned short Offset   :8; // offset of first LED to read: 0 to 63
    unsigned short Size     :6; // number of LEDs to read - 1
    unsigned short Reserve  :2;
} MTCX_LAYER_LED_ADDR;
```

Listing 72: Data types for LED functions

7.14.1 Reading the number of LEDs

The number of LEDs a panel supports can be read via the MTCX **Panel Specials** register (see page 52).

Information:

For technical reasons, all B&R devices that are able to support LEDs currently report the number of LEDs as 128. This does not necessarily mean that the device actually has 128 LEDs. There may in fact be fewer or no LEDs at all present on the device. It can only be assumed that the device does not support LEDs if the number of LEDs returned in 0.

The following C code example shows how to read the number of supported LEDs.

For performance reasons, only the required byte of the **Panel Specials** register is accessed.

```
// Get available LED count.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   LedCount
//     [out] Points to variable that receives the
//     maximum number of supported matrix LEDs: 0 to 128.
//
// Return 0 at success and -1 at failure.
int GetLedCount(unsigned int PanelNumber, unsigned char *LedCount)
{
    int retvalue;

    // TODO: acquire synchronization object for panel data here

    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue == MTCX_OKAY)
        *LedCount = READ_PORT_UCHAR(MTCX_PANEL_SPECIALS_ADDR + 3); // read byte 3 only

    // TODO: release synchronization object for panel data here

    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 73: GetLedCount – Reading the number of LEDs

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.

7.14.2 Reading/setting the status of the LED matrix

The states of a panel's matrix LEDs can be read and set with the MTCX **LED Service** command (see page 26).

As with the keys, LEDs can be operated on four different layers.

The following C code example shows how to read the entire LED matrix of a connected panel.

Each LED is represented by two bits: 00b = LED off, 01b = LED blinks slowly, 10b = LED blinks quickly, 11b = LED on.

```
// Read LED matrix.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Layer
//     [in] Specifies the layer: 0 to 3.
//   LedMatrix
//     [in] Points to a buffer that receives the
//     states of 128 LED (2 bits per LED).
//   Size
//     [in] Specifies the size of the buffer in bytes: 32.
//
// Return 0 at success and -1 at failure.
int GetLedMatrix(unsigned int PanelNumber, unsigned char Layer, unsigned char *LedMatrix, int
Size)
{
    MTCX_LAYER_LED_ADDR LedAddr;
    unsigned char ErrorCode;
    int retvalue;

    if (Layer > 3)
        return -1;

    if (Size != 32)
        return -1;

    LedAddr.Reserve = 0;
    LedAddr.Offset = 0;
    LedAddr.Size = 63;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    // Read LEDs 0 to 63.
    if ((retvalue = MtcxReadCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, LedMatrix, 16, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    LedMatrix += 16;
    LedAddr.Offset = 64;

    // Read LEDs 64 to 127.
    if ((retvalue = MtcxReadCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, LedMatrix, 16, &ErrorCode)) != MTCX_OKAY)
```

```
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 74: GetLedMatrix – Reading the status of the LED matrix

The following C code example shows how to write to a connected panel's entire LED matrix.

```
// Write LED matrix.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Layer
//     [in] Specifies the layer: 0 to 3.
//   LedMatrix
//     [in] Points to a buffer that contains the
//     states of 128 LED (2 bits per LED).
//   Size
//     [in] Specifies the size of the buffer in bytes: 32.
//
// Return 0 at success and -1 at failure.
int SetLedMatrix(unsigned int PanelNumber, unsigned char Layer, unsigned char *LedMatrix, int
Size)
{
    MTCX_LAYER_LED_ADDR LedAddr;
    unsigned char ErrorCode;
    int retvalue;

    if (Layer > 3)
        return -1;

    if (Size != 32)
        return -1;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    LedAddr.Reserve = 0;
    LedAddr.Offset = 0;
    LedAddr.Size = 63;

    // Write LEDs 0 to 63.
    if ((retvalue = MtcxWriteCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, LedMatrix, 16, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    LedMatrix += 16;
    LedAddr.Offset = 64;

    // Write LEDs 64 to 127.
    if ((retvalue = MtcxWriteCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, LedMatrix, 16, &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }
```

```
    return 0;
}
```

Listing 75: SetLedMatrix – Writing to the LED matrix

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.14.3 Reading/setting the status of individual LEDs

The states of a panel's individual LEDs can also be read and set with the MTCX **LED Service** command (see page 26).

As with the keys, LEDs can be operated on four different layers.

The following C code example shows how to read the status of a single LED on a connected panel.

Possible LED states (LedState): 0 = LED off, 1 = LED blinks slowly, 2 = LED blinks quickly, 3 = LED on

```
// Get the state of a LED.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Layer
//     [in] Specifies the layer: 0 to 3.
//   LedNumber
//     [in] Specifies the LED number: 0 to 127.
//   LedState
//     [ount] Points to a variable that receives the
//     LED state: 0 = off, 1 = slow blinking,
//     2 = fast blinking, 3 = on.
//
// Return 0 at success and -1 at failure.
int GetLed(unsigned int PanelNumber, unsigned char Layer, unsigned char LedNumber, unsigned char
*LedState)
{
    MTCX_LAYER_LED_ADDR LedAddr;
    MTCX_LAYER_LED_DATA LedData;
    unsigned char ErrorCode;
    int retvalue;

    if (Layer > 3)
        return -1;
    if (LedNumber >= 128)
        return -1;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    LedAddr.Reserve = 0;
    LedAddr.Offset = LedNumber;
    LedAddr.Size = 0; // 0 = one LED

    if ((retvalue = MtcxReadCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, &LedData, sizeof(LedData), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    *LedState = LedData[0] & 0x3;
    return 0;
}
```

Listing 76: GetLed – Reading the status of individual LEDs

The following C code example shows how to set the status of a single LED on a connected panel.

```
// Set the state of a LED.
//
// Parameters
//   PanelNumber
//     [in] Specifies the panel number: 0 to 15.
//   Layer
//     [in] Specifies the layer: 0 to 3.
//   LedNumber
//     [in] Specifies the LED number: 0 to 127.
//   LedState
//     [in] Specifies the LED state: 0 = off, 1 = slow blinking,
//     2 = fast blinking, 3 = on.
//
// Return 0 at success and -1 at failure.
int SetLed(unsigned int PanelNumber, unsigned char Layer, unsigned char LedNumber, unsigned char
LedState)
{
    MTCX_LAYER_LED_ADDR LedAddr;
    MTCX_LAYER_LED_DATA LedData;
    unsigned char ErrorCode;
    int retvalue;

    if (Layer > 3)
        return -1;
    if (LedNumber >= 128)
        return -1;
    if (LedState > 3)
        return -1;

    // Check if panel is linked.
    retvalue = MtcxSwitchToLinkedPanel(PanelNumber);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    LedAddr.Reserve = 0;
    LedAddr.Offset = LedNumber;
    LedAddr.Size = 0; // 0 = one LED
    LedData[0] = LedState;

    if ((retvalue = MtcxWriteCommand(MTCX_CMD_LED_SERVICE0 + Layer, *(unsigned short*)&LedAddr,
PanelNumber,
        MTCX_DEV_PANEL, &LedData, sizeof(LedData), &ErrorCode)) != MTCX_OKAY)
    {
        // TODO: add your error handling here
        return -1;
    }

    return 0;
}
```

Listing 77: SetLed – Setting the status of individual LEDs

The definitions used can be found on page 59.
The **MtcxSwitchToLinkedPanel** function can be found on page 68.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.14.4 Reading/setting the status of the Run LED

The Run LED can be read using the MTCX's **Config 2nd** register (see page 13).

The following C code example shows how to read the Run LED.

```
// Read Run LED state.
//
// Parameters
//   State
//     [out] Points to a variable that receives the
//     LED state: 0 = off, 1 = on.
void GetRunLed(int *State)
{
    MTCX_CONFIG_2ND_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_CONFIG_2ND_ADDR);

    *State = reg.RunLedState;
}
```

Listing 78: GetRunLed – Reading the status of the Run LED

The following C code example shows how to set the Run LED.

```
// Set Run LED state.
//
// Parameters
//   State
//     [in] Specifies the LED state: 0 = off, 1 = on.
void SetRunLed(int State)
{
    MTCX_CONFIG_2ND_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_CONFIG_2ND_ADDR);

    reg.RunLedState = State;
    WRITE_PORT_ULONG(MTCX_CONFIG_2ND_ADDR, *(unsigned long*)&reg);
}
```

Listing 79: SetRunLed – Setting the status of the Run LED

The definitions used can be found on page 59.

7.15 Operating the watchdog, resetting the software

The watchdog is a life sign monitor that can be turned on or off by a PC application at any time.

Watchdog monitoring is performed by the MTCX.

The watchdog must be configured by the PC application using the MTCX command interface. During its configuration, a time window is set in which the watchdog must be acknowledged.

The watchdog is acknowledged by writing to the watchdog toggle bit. After configuration, the watchdog is enabled by the PC application the first time the toggle bit is written to. The PC application must then acknowledge the watchdog within the allotted time window.

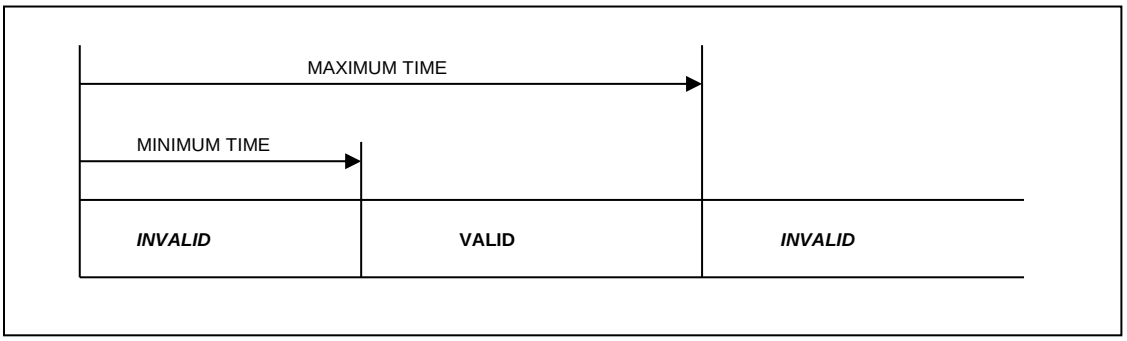

Figure 2: Watchdog times

If a time violation occurs (e.g. due to a program crash or continuous loop in the PC application), the PC's NMI logic is activated by the MTCX. If NMI logic is turned off, the system immediately resets. The watchdog is disabled after every reset.

If needed, you can also perform a software reset of the PC using the Watchdog Service command.

Caution!

Any unsaved data is lost during a reset!

The following code examples show how to operate the watchdog or carry out a software reset of the PC.

7.15.1 Setting the watchdog time window

In order for the watchdog to be used, the time window for acknowledging (toggling) the watchdog must first be set using the MTCX **Watchdog Service** command (see page 25).

The time window is defined by a minimum and maximum time in milliseconds. The minimum time is normally set to 0 milliseconds.

Information:

Set the maximum time as high as possible in order to minimize the load on the system when the watchdog is acknowledged.

The following C code example shows how to read the watchdog time window.

```
// Read minimum and maximum watchdog time.
//
// Parameters
//   MinTime
//     [out] Points to a variable that receives the
//     minimum watchdog time in milliseconds: 0 to 65535.
//   MaxTime
//     [out] Points to a variable that receives the
//     maximum watchdog time in milliseconds: 0 to 65535.
//
// Return 0 at success and -1 at failure.
int GetWatchdogTime(unsigned short *MinTime, unsigned short *MaxTime)
{
    MTCX_WATCHDOG_CONFIG_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_WDOG_SERVICE, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    *MinTime = data.MinTime;
    *MaxTime = data.MaxTime;
    return 0;
}
```

Listing 80: GetWatchdogTime – Reading the watchdog time window

The following C code example shows how to set the watchdog time window.

```
// Set minimum and maximum watchdog time.
//
// Parameters
//   MinTime
//     [in] Specifies the minimum watchdog time
//     in milliseconds: 0 or 65535.
//   MaxTime
//     [in] Specifies the maximum watchdog time
//     in milliseconds: 0 or 65535.
//
// Return 0 at success and -1 at failure.
//
// Remarks
//   Set MinTime and MaxTime to 0 to deactivate
//   the watchdog.
int SetWatchdogTime(unsigned short MinTime, unsigned short MaxTime)
{
    MTCX_WATCHDOG_CONFIG_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    data.MinTime = MinTime;
    data.MaxTime = MaxTime;

    retvalue = MtcxWriteCommand(MTCX_CMD_WDOG_SERVICE, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }
    return 0;
}
```

Listing 81: SetWatchdogTime – Setting the watchdog time window

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.15.2 Toggle watchdog

After the time window has been set, the watchdog is enabled when the toggle bit in the Baseboard Support register (see page 43) is written to. The toggle bit must then be written to inside of the configured time window before the watchdog is deactivated again and a new time window is set.

The toggle bit must always be set to 0 by the PC application. MTCX sets the toggle bit to 1 every millisecond.

Caution!

The PC is rebooted when the toggle bit is not set to 0 within the configured time window. Then data may be lost!

The following C code example shows how to write to the watchdog toggle bit.

```
// Toggle watchdog.
void ToggleWatchdog()
{
    MTCX_BASEBOARD_SUPPORT_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_BASEBOARD_SUPPORT_ADDR);
    reg.WatchdogToggle = 0;
    WRITE_PORT_ULONG(MTCX_BASEBOARD_SUPPORT_ADDR, *(unsigned long*)&reg);
}
```

Listing 82: ToggleWatchdog – Toggling the watchdog

The definitions used can be found on page 59.

7.15.3 Software reset

You can also reset the PC via software using the MTCX **Watchdog Service** command (see page 28).

The following C code example shows how to perform this reset.

```
// Generate a software reset.
// No return at success and -1 at failure.
int SoftwareReset()
{
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxWriteCommand(MTCX_CMD_WDOG_SERVICE, 0x00, 0x02, MTCX_DEV_BASEBOARD,
        0, 0, &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }
    // We should not reach this point...
    return -1;
}
```

Listing 83: SoftwareReset – Resetting the PC

When executed successfully, this function should not return a value.

The definitions used can be found on page 59.
The **MtcxWriteCommand** function can be found on page 74.

7.16 User functions

A User Serial ID can be assigned on an APC910 and PPC900. The user serial ID is a 32-bit value that can be freely defined. The user serial ID allows you to give the B&R device your own identification number. The user serial ID is permanently saved on the B&R device.

The user serial ID is accessed by the MTCX.

The following code examples show how to read and set the user serial ID.

7.16.1 Reading/setting the user serial ID

The User Serial ID can be read and set using the MTCX **Device Info** command (see page 28).

The following C code example shows how to read the user serial ID.

```
// Read User Serial ID.
//
// Parameters
//   Id
//     [out] Points to a variable that receives the
//     User Serial ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int GetUserSerialId(unsigned long *Id)
{
    MTCX_ID_SERVICE_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_DEVICE_INFO, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    *Id = data.User;
    return 0;
}
```

Listing 84: GetUserSerialID – Reading the user serial ID

The following C code example shows how to set the user serial ID.

```
// Set User Serial ID.
//
// Parameters
//   Id
//     [in] Specifies the User Serial ID: 00000000h to FFFFFFFFh.
//
// Return 0 at success and -1 at failure.
int SetUserSerialId(unsigned long Id)
{
    MTCX_ID_SERVICE_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    data.Reserve = 0;
    data.User = Id;

    retvalue = MtcxWriteCommand(MTCX_CMD_DEVICE_INFO, 0, 0, MTCX_DEV_BASEBOARD,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }
    return 0;
}
```

Listing 85: SetUserSerialID – Setting the user serial ID

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.17 UPS functions

The following functions can be performed for an installed UPS IF option:

- Checking if UPS is "detected" and "linked"
- Reading UPS status
- Reading UPS battery voltage, current and temperature
- Reading/writing UPS user settings
- Shutting down UPS

UPS functions are executed by the MTCX.

The following code examples illustrate how to read the UPS status and operating data, how to read and write the UPS user settings and how to shutdown the UPS.

7.17.1 Checking if UPS is "detected"

You can also check whether a UPS has been detected (i.e. is installed) via the MTCX **UPS Status** register (see page 47).

The following C code example shows how to check whether a UPS has been detected.

```
// Check if UPS is detected.
//
// Return 1 if UPS is detected and 0 if not.
int IsUpsDetected()
{
    MTCX_UPS_STATUS_REG_BYTE0 reg;

    *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_UPS_STATUS_ADDR); // read byte 0 only
    return reg.Detected;
}
```

Listing 86: IsUpsDetected – check if UPS is “detected”

The definitions used can be found on page 59.

7.17.2 Checking if UPS is "linked"

You can also check whether a UPS is linked (i.e. if communication with the UPS is functional) using the MTCX **UPS Status** register (see page 47).

The following C code example shows how to check whether a UPS is linked.

```
// Check if UPS is linked
// (that means communication with the UPS is OK).
//
// Return 1 if UPS is linked and 0 if not.
int IsUpsLinked()
{
    MTCX_UPS_STATUS_REG_BYTE0 reg;

    *(unsigned char*)&reg = READ_PORT_UCHAR(MTCX_UPS_STATUS_ADDR); // read byte 0 only
    return reg.Linked;
}
```

Listing 87: IsUpsLinked – check if UPS is “connected”

The definitions used can be found on page 59.

7.17.3 Reading UPS status flags

The current UPS status (e.g. whether battery operation is active) can be read via the MTCX **UPS Status** register (see page 47).

Information:

This function only provides valid values when the UPS is "linked" (see page 151).

The following C code example shows how to read the UPS status. The status is provided in the form of individual flags (see MTCX_UPS_STATUS_x constants).

Information:

Some status bits are not supported by the MTCX (see page 47).

```
// Read UPS status flags.
//
// Parameters
//   Status
//     [out] Points to variable that receives the
//     UPS status flags.
void GetUpsStatusFlags(unsigned short *Flags)
{
    MTCX_UPS_STATUS_REG reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_UPS_STATUS_ADDR);
    *Flags = reg.Flags;
}
```

Listing 88: GetUpsStatusFlags – read UPS status flags

The definitions used can be found on page 59.

7.17.4 Reading UPS battery voltage

The current voltage of the UPS battery can be read via the MTCX **UPS Values** register (see page 48).

Information:

This function only provides valid values when the UPS is "linked" (see page 151).

The following C code example shows how to read the UPS battery voltage in mV.

```
// Read UPS battery voltage.
//
// Parameters
//   Voltage
//     [out] Points to variable that receives the
//     UPS battery voltage in mV.
void GetUpsBatVoltage(unsigned short *Voltage)
{
    MTCX_UPS_VALUES_REG2 reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_UPS_VALUES_ADDR);
    *Voltage = reg.BatVoltage;
}
```

Listing 89: GetUpsBatVoltage – read UPS battery voltage

The definitions used can be found on page 59.

7.17.5 Reading UPS battery current

The UPS charging/discharging current (when the UPS is in battery operation) can be read via the MTCX **UPS Values** register (see page 48).

Information:

This function only provides valid values when the UPS is "linked" (see page 151).

The following C code example shows how to read the UPS battery current in mA.

```
// Read UPS battery current.
//
// Parameters
//   Current
//     [out] Points to variable that receives the
//     UPS battery current in mA.
//     Positive values indicate the charge current.
//     Negative values indicate the discharge current.
void GetUpsBatCurrent(short *Current)
{
    MTCX_UPS_VALUES_REG2 reg;

    *(unsigned long*)&reg = READ_PORT_ULONG(MTCX_UPS_VALUES_ADDR);
    *Current = reg.BatCurrent;
}
```

Listing 90: GetUpsBatCurrent – read UPS battery current

The definitions used can be found on page 59.

7.17.6 Reading UPS battery temperature

The current temperature of the UPS battery can be read via the MTCX **UPS Status** register (see page 47).

Information:

This function only provides valid values when the UPS is "linked" (see page 151).

The following C code example shows how to read the UPS battery temperature in degrees Celsius.

Note: The value -128 is shown if the temperature cannot be read,

```
// Read UPS battery temperature.
//
// Parameters
//   Temp
//     [out] Points to variable that receives the
//     UPS battery temperature in degrees Celsius.
//     -128 indicates the temperature cannot be read.
void GetUpsBatTemp(char *Temp)
{
    *Temp = (char)READ_PORT_UCHAR(MTCX_UPS_STATUS_ADDR + 3); // read byte 3 only
}
```

Listing 91: GetUpsBatTemperature – read UPS battery temperature

The definitions used can be found on page 59.

7.17.7 Reading/writing UPS user settings

The UPS user settings can be read and written using the MTCX **Module Correction** command (see page 34).

Caution!

A Low Battery Shutdown terminates any UPS Shutdown (see page 157) already in progress, which can shorten the system's planned shutdown time!

The following C code example shows how to read the UPS user settings.

Information:

- **If there are no user settings on the UPS, the function returns an error.**
- **Read the description of the Module Correction command (see page 34) for more information about the UPS user settings.**

```
// Get UPS user settings.
//
// Parameters
//   LowBatShutdownTime
//     [out] Points to a variable that receives the
//     low battery shutdown time in seconds.
//
// Return 0 at success and -1 at failure.
int GetUpsUserSettings(unsigned short *LowBatShutdownTime)
{
    struct
    {
        unsigned char  Ok;
        unsigned char  Special;
        unsigned short Counter;
        unsigned short LowBatShutdownTime;
        unsigned short Reserve;
    } data;
    int retvalue;
    unsigned char ErrorCode;

    retvalue = MtcxReadCommand(MTCX_CMD_MODULE_CORR, 0x600, MTCX_DEVNUM_APC910_IF1, 0,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }

    *LowBatShutdownTime = data.LowBatShutdownTime;
    return 0;
}
```

Listing 92: GetUpsUserSettings – read UPS user settings

The following C code example shows how to write the UPS user settings.

```
// Set UPS user settings.
//
// Parameters
//   LowBatShutdownTime
//     [in] Specifies the low battery shutdown time in seconds: 10 to 1200.
//
// Return 0 at success and -1 at failure.
int SetUpsUserSettings(unsigned short LowBatShutdownTime)
{
    struct
    {
        unsigned short LowBatShutdownTime;
        unsigned short Reserve;
    } data;
    int retvalue;
    unsigned char ErrorCode;

    data.LowBatShutdownTime = LowBatShutdownTime;

    retvalue = MtcxWriteCommand(MTCX_CMD_MODULE_CORR, 0x600, MTCX_DEVNUM_APC910_IF1, 0,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }
    return 0;
}
```

Listing 93: SetUpsBatterySettings – write UPS user settings

Information:

The UPS does not have to be restarted for the changed user settings to take effect.

The definitions used can be found on page 59.
The **MtcxReadCommand** function is listed on page 73.
The **MtcxWriteCommand** function can be found on page 74.

7.17.8 Shutting down the UPS

The UPS can be shutdown using the MTCX **UPS Service** command (see page 39).

For example, this function is called up by the Windows UPS service before the system is shutdown. This prevents the system from remaining in UPS operation and draining the UPS battery in the event Windows does not shutdown properly.

The following C code example shows how to shutdown the UPS. A delay time can be entered in seconds.

Caution!

The specified delay time must be longer than the time required by the operating system to shutdown. Otherwise data may be lost. Make sure to also account for the UPS Low Battery Shutdown time (see page 155)!

Information:

- **This function does not return an error when a UPS is not linked (see page 151).**
- **See the description of the UPS Service command for more information about the UPS shutdown behavior.**

```
// Shutdown UPS.
//
// Parameters
//   Delay
//     [in] Specifies the minimum amount of time,
//     in seconds, to wait before turning off the
//     UPS power outlets: 10 to 1200.
//
// Return 0 at success and -1 at failure.
int ShutdownUps(unsigned short Delay)
{
    MTCX_UPS_SHUTDOWN_DATA data;
    int retvalue;
    unsigned char ErrorCode;

    data.Reserve = 0;
    data.Delay = Delay;

    retvalue = MtcxWriteCommand(MTCX_CMD_UPS_SERVICE, 0x01, 0, MTCX_DEV_UPS,
        &data, sizeof(data), &ErrorCode);
    if (retvalue != MTCX_OKAY)
    {
         // TODO: add your error handling here
         return -1;
    }
    return 0;
}
```

Listing 94: ShutdownUps – shutdown UPS

The definitions used can be found on page 59.
The **MtcxWriteCommand** function can be found on page 74.

8 Figure index

9 Table index

10 Listing index

11 Index

M

O

P

R

S

T

U

V

X