21M000-13 E2 - 2004-10-20

Embedded Solutions

MDIS4 under Windows

MEN Driver Interface System

User Manual

mikro elektronik
gmbh · nürnberg

About this Document

This manual is a complete documentation of MDIS4/2004 under Windows.

History

Edition	Comments	Technical Content	Date of Issue
E1	First edition	D. Pfeuffer	2004-06-22
E2	Minor errors corrected	D. Pfeuffer	2004-10-20

Conventions

⚠	This sign marks important notes or warnings concerning proper functionality of the product described in this document. You should read them in any case.
italics	Folder, file and function names are printed in *italics*.
bold	**Bold** type is used for emphasis.
`monospace`	A `monospaced` font type is used for listings, C function descriptions or wherever appropriate.
hyperlink	Hyperlinks are printed in blue color.
🌐	The globe will show you where hyperlinks lead directly to the Internet, so you can look for the latest information online.
IRQ# /IRQ	Signal names followed by "#" or preceded by a slash ("/") indicate that this signal is either active low or that it becomes active at a falling edge.
|	Vertical lines on the outer margin signal technical changes to the previous edition of the document.

Copyright Information

Please recycle

Germany	**France**	**UK**	**USA**
MEN Mikro Elektronik GmbH	MEN Mikro Elektronik SA	MEN Micro Ltd	MEN Micro, Inc.
Neuwieder Straße 5-7	18, rue René Cassin	Whitehall, 75 School Lane	PO Box 4160
90411 Nuremberg	ZA de la Châtelaine	Hartford, Northwich	Lago Vista, TX 78645-4160
Phone +49-911-99 33 5-0	74240 Gaillard	Cheshire UK, CW8 1PF	Phone (512) 267-8883
Fax +49-911-99 33 5-901	Phone +33 (0) 450-955-312	Phone +44 (0) 1477-549-185	Fax (512) 267-8803
E-mail info@men.de	Fax +33 (0) 450-955-211	Fax +44 (0) 1477-549-178	E-mail sales@menmicro.com
www.men.de	E-mail info@men-france.fr	E-mail info@menmicro.co.uk	www.menmicro.com
	www.men-france.fr	www.menmicro.co.uk	

Contents

A 1 General . . . 8
A 1.1 Name Conventions . . . 8
A 1.2 Supported Windows Versions . . . 8
A 1.3 Introduction to MDIS . . . 9
A 1.4 Available Packages . . . 9
A 1.5 How MDIS4 Maps into the Windows NT Architecture . . . 10
A 2 Contents of the Package . . . 12
A 3 Installing MDIS4 on the Host System . . . 15
A 3.1 Installing the System Package . . . 15
A 3.2 Installing a Driver Package . . . 16
A 3.2.1 Hints on Updating and Deinstalling . . . 17
A 4 Installing the Target System . . . 18
A 4.1 VMEbus Systems and Swapping Drivers . . . 19
A 4.2 Choosing the Right Windows Driver Type . . . 20
A 4.3 Installing Windows NT 4.0 Drivers . . . 22
A 4.4 Installing Windows 2000 PnP Drivers . . . 24
A 4.4.1 W2k Driver & PnP Basics . . . 25
A 4.4.2 Providing the Installation Files . . . 26
A 4.4.3 Installing PnP Devices . . . 27
A 4.4.4 Installing Non-PnP Devices . . . 29
A 4.4.5 Reinstalling and Updating W2k PnP Drivers . . . 31
A 4.4.6 Notes for Hardware Device Configuration Changes . . . 32
A 5 NT4 Drivers and Device Configuration . . . 34
A 5.1 Starting and Stopping NT4 Drivers . . . 36
A 5.1.1 Starting Drivers Manually . . . 36
A 5.1.2 Stopping Drivers Manually . . . 36
A 5.2 Driver Dependencies . . . 38
A 5.3 NT4 Driver Standard Parameters . . . 39
A 5.4 NT4 Driver Descriptor Files . . . 41
A 5.4.1 Generating .reg Descriptors for NT4 Drivers . . . 42
A 5.4.2 NT4 Driver-Specific MDIS Keys . . . 46
A 6 W2k Drivers & Device Configuration . . . 52
A 6.1 W2k Device Parameters . . . 53
A 6.2 MDIS4 Device Parameters . . . 58
A 7 Building MDIS4 Applications from C Sources . . . 60
A 7.1 Using NMAKE . . . 61
A 7.2 Using VC++ IDE . . . 62
A 7.2.1 Building an Application . . . 62
A 7.2.2 Cloning a Project File . . . 63

A 8 Writing Applications for MDIS . . . 64
A 8.1 Basics of MDIS API Libraries . . . 65
A 8.2 C/C++ Specifics . . . 66
A 8.2.1 Using Static MDIS API Libraries . . . 66
A 8.2.2 Using MDIS API DLLs . . . 67
A 8.2.3 Visual C++ Notes . . . 67
A 8.3 Visual Basic Specifics . . . 68
A 8.3.1 VB Declaration Files . . . 68
A 8.3.2 Multithreading . . . 68
A 8.3.3 MAPIVB – VB Example MDIS4 Application . . . 69
A 8.4 Delphi Specifics . . . 69
A 8.4.1 Delphi Import Units . . . 69
A 8.5 Measurement Studio . . . 70
A 8.5.1 General . . . 70
A 8.5.2 Customizing Your Project . . . 70
A 8.5.3 Writing Code with MDIS4 . . . 72
A 8.6 MDISNT Test and Configuration Utility . . . 74
A 8.6.1 Using MDISNT . . . 74

A 9 Solving Problems . . . 76
A 9.1 Gathering Information . . . 76
A 9.1.1 Viewing Event-Log Entries . . . 76
A 9.1.2 Displaying the Used Driver Parameters (NT4) . . . 76
A 9.1.3 Displaying the Used Resources . . . 77
A 9.1.4 Getting Revision Information on MDIS Modules . . . 77
A 9.1.5 Getting .sys/.dll File Information . . . 78
A 9.1.6 Examining Dependencies of Executables . . . 78
A 9.1.7 Viewing the PCI Configuration Space . . . 78
A 9.1.8 Displaying Debug Output from Checked Modules . . . 78
A 9.2 Problems and Solutions . . . 79
A 9.2.1 NT4 Driver Does Not Start . . . 79
A 9.2.2 NT4 Driver Does Not Stop . . . 79
A 9.2.3 Device Driver Does Not Work . . . 80
A 9.2.4 Strings of Event-log Entries are Missing . . . 80
A 9.2.5 W2k Device Cannot be Opened . . . 80
A 9.2.6 Cannot Link C/C++ Application with Static MDIS API Libraries . . . 80

A 10 Performance . . . 81
A 10.1 MDIS-API Calls without Hardware Access . . . 81
A 10.1.1 NT4 Drivers on 200MHz D1 CPU . . . 81
A 10.1.2 NT4/W2k Drivers on 1.2GHz F7N CPU . . . 82
A 10.2 MDIS-API Calls with Hardware Access . . . 84
A 10.2.1 NT4 Drivers on 200MHz D1 CPU . . . 84
A 10.2.2 NT4/W2k Drivers on 1.2GHz F7N CPU . . . 85

A 11 Development Tools and Resources 86
A 11.1 Development Tools 86
A 11.2 Literature 87
A 11.3 Resources on the Web 87

B 1 MBUF Device I/O 88
B 1.1 Channels 88
B 1.2 Channel I/O 88
B 1.2.1 Channel I/O Modes 88
B 1.2.2 Channel Direction 88
B 1.3 Block I/O 89
B 1.3.1 Driver Buffers 89
B 1.3.2 Block I/O Modes 90
B 1.4 Buffer Events 95

B 2 Status Codes 97
B 2.1 Status Code Types 97
B 2.2 Common Status Codes 98
B 2.2.1 MDIS Kernel Status Codes 98
B 2.2.2 Input Buffer Management Status Codes 101
B 2.2.3 Output Buffer Management Status Codes 102
B 2.2.4 Device Driver Status Codes 104
B 2.2.5 Board Handler Status Codes 105

B 3 Error Codes 106
B 3.1 Operating System Specific Errors 106
B 3.2 MDIS Kernel Errors 106
B 3.3 Device Driver Errors 107
B 3.4 Board Handler Errors 107
B 3.5 Descriptor Errors 108
B 3.6 ID PROM Errors 108
B 3.7 Operating System Service Errors 108
B 3.7.1 PCI System Specific Error Codes 109
B 3.7.2 VMEbus Specific Error Codes 110
B 3.8 Buffer Management Errors 110
B 3.9 PLD Loader Errors 110
B 3.10 CPU Handler (Bus Mapper) Errors 111
B 3.11 BBIS Kernel Errors 111

B 4 MDIS Device Descriptors 112
B 4.1 General 112
B 4.1.1 Devices and Device Descriptors 112
B 4.1.2 Boards and Board Descriptors 113
B 4.2 Descriptor Format 114
B 4.3 Device Descriptor Keys 115
B 4.3.1 Additional Descriptor Keys for PCI Devices (PC•MIP Modules) 116

B 4.4 Board Descriptor Keys 118
B 4.4.1 VMEbus M-Module Carrier Boards 119
B 4.4.2 CompactPCI M-Module Carrier Boards 120
B 4.4.3 Standard PCI M-Module Carrier Boards 121
B 4.4.4 PC•MIP Carrier Boards. 122
B 4.5 Driver Debugging 124
B 4.5.1 Debug Level 124
B 4.5.2 Debug Settings 125

Part A MDIS4 under Windows

A 1 General

In this manual, Windows NT refers to all Windows NT based operating systems (Windows NT 4.0, Windows 2000, Windows XP) whereas Windows NT 4.0 refers indeed only to the Windows NT 4.0 operating system.

Since the MDIS4 User Guide might not be updated for each new release of the MDIS4 System Package for Windows NT/2000/XP/Embedded, please read the uncompressed *readme.txt* file for the latest news of the current release. The file is located in the main ZIP file of the package.

A 1.1 Name Conventions

Windows NT (or just NT)
MS Windows NT based operating system (NT 4.0, Windows 2000, Windows XP)

Windows NT 4.0 (or just NT4)
MS Windows NT 4.0 operating system only

Windows 2000 (or just W2k)
MS Windows 2000

Windows XP (or just XP)
MS Windows XP

Windows XP Embedded (or just XPe)
MS Windows XP Embedded

A 1.2 Supported Windows Versions

MDIS for Windows provides two different Windows driver types to support the following NT based Microsoft operating systems:

Windows NT 4.0 drivers for:

- Windows NT Workstation 4.0
- Windows NT Embedded 4.0[1]

Windows 2000 PnP drivers for:

- Windows 2000 Professional
- Windows XP Professional
- Windows XP Home Edition
- Windows XP Embedded[1]

[1] Currently, MEN driver packages contain no ready built component definition files (*.kdf/ .sld* files) for Windows NT/XP Embedded. However, the component definition files can be created using the MS Component Designer.
For Windows XP Embedded you can convert the *.inf* files from the Windows 2000 Plug & Play drivers into component definition (*.sld*) files by either importing *.inf* files into MS Component Designer, or using MS EConvert (for details, refer to the MS MSDN Library).

However, with some limitations, it is also possible to use the Windows NT 4.0 drivers for Windows 2k/XP. Refer to Chapter A 4.2 Choosing the Right Windows Driver Type on page 20.

A 1.3 Introduction to MDIS

MDIS, the MEN Driver Interface System, is a framework to develop device drivers for almost any kind of I/O hardware. A properly written driver runs on all operating systems supported by MDIS. Operating systems currently supported include Windows, VxWorks, OS-9, Linux and QNX.

MDIS4 is the fourth major revision of MDIS and is the first revision that offers full platform independence. Earlier revisions were limited to run under MS-DOS and OS-9 and were fixed to support M-Module mezzanines.

Typical I/O hardware supported by MDIS device drivers:

- Binary I/O
- Analog I/O
- Motion controllers
- Fieldbus controllers (CAN, Profibus etc.)
- Other specialized hardware like watchdogs, hardware monitors, etc.

And this hardware is typically located on:

- M-Module mezzanines
- PC•MIP mezzanines
- PMC Modules
- Other PCI hardware
- Chips on CPU boards and FPGA units

MDIS drivers can be used for all the types of hardware listed above, because in these cases the driver function can be presented to the application using the MDIS standard API. There are some device types, like network and disk I/O, where the MDIS API cannot be used because the operating system already supports this kind of device. For these devices, you still need to develop a specific driver for each operating system.

A 1.4 Available Packages

Apart from the MDIS4 System Package for Windows NT/2000/XP/Embedded, MEN supplies the following driver packages for Windows:

- MDIS4 driver packages for Windows that contain the sources of MDIS4 low-level drivers and example programs as well as the corresponding ready-to-use, built object code for Windows.
- Native driver packages for Windows which contain ready-to-use, built object code of native drivers for Windows.

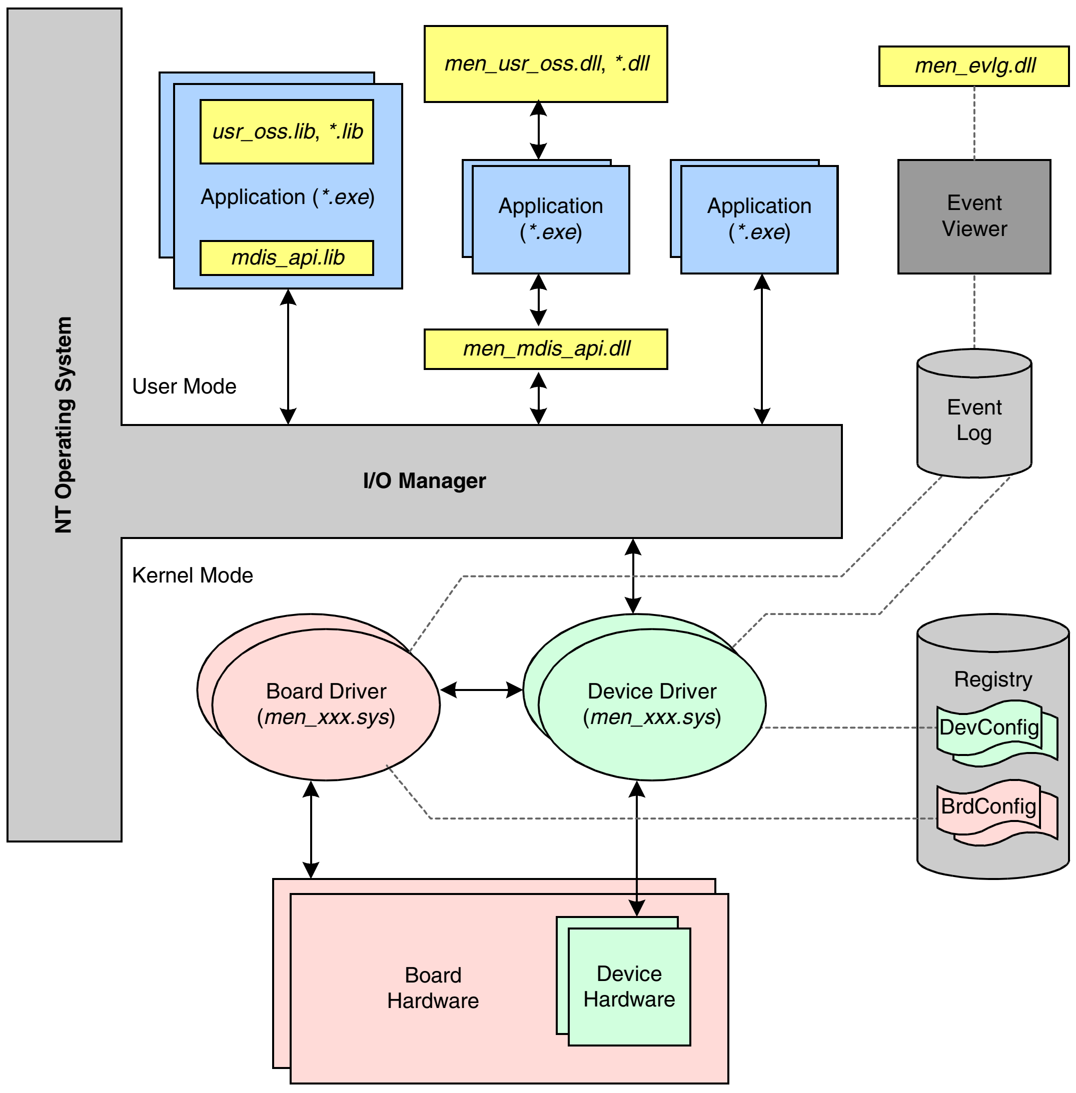
Note: The OS-independent MDIS4 low-level driver packages that contain only the sources are no longer required.

A 1.5 How MDIS4 Maps into the Windows NT Architecture

Under Windows NT, user application code runs in user mode (ring 3 of an i386-based CPU), whereas operating system code (such as system services and device drivers) runs in kernel mode (ring 0 of an i386-based CPU).

Since there is no direct access from applications to physical memory and I/O ports, all MDIS drivers run in kernel mode. The Windows NT I/O Manager converts all input/output requests from user-mode threads into properly sequenced calls to the MDIS device drivers.

Figure A1. *MDIS4 Module Overview for Windows NT*



MDIS for Windows uses two different types of drivers:

- Board drivers
- Device drivers

A board driver deals with the hardware of a board (e. g. CPU board or mezzanine carrier board), whereas a device driver deals with the hardware of a device (e. g. M-Module, PC•MIP, onboard device) which is plugged or located on a board.

What's Specific to Board Drivers

- There is a special board driver (e. g. *men_d201.sys*) for each board type (e. g. D201 M-Module carrier board).
- A board driver normally comprises an operating system-independent BBIS handler and some libraries.
- A board driver doesn't provide an interface to applications. Only device drivers communicate with a board driver.
- The board drivers must be started in the system before the device drivers and must be stopped after the device drivers.

What's Specific to Device Drivers

- There is a special device driver (e. g. *men_m50.sys*) for each device type (e. g. M50 M-Module).
- A device driver normally comprises an operating system-independent MDIS low-level driver and some libraries. A device driver can also be a native Windows driver for hardware similar to standard PC components (e. g. serial ports).
- An application can access a device driver via the MDIS-API functions (*M_open*, *M_read*, etc.). A native Windows driver (e. g. for serial ports) can be accessed through functions of the Win32 API (*CreateFile*, *ReadFile*, etc.).

The Application Interface

The MDIS-API library and other MDIS API libraries (USR-OSS, etc.) can be statically linked to an application. Application programs can also use the DLL versions of the MDIS API libraries (prefixed with *men_*) at runtime, which allows the development of non-C applications (e. g. Visual Basic or Delphi programs).

Note: The static and the DLL version of the MDIS-API library version 3.0 and later require the *men_winspec.dll* for the Windows 2000 Plug & Play driver support (refer to Chapter A 8 Writing Applications for MDIS on page 64). For simplicity, the *men_winspec.dll* is not shown in Figure A1, MDIS4 Module Overview for Windows NT, on page 10.

A 2 Contents of the Package

The MDIS4 System Package for Windows NT/2000/XP/Embedded distribution contains:

- An installation program for the MDIS4 System Package.
- Uncompressed files.
- Compressed files (included in cabinet files *.cab*).

Uncompressed Files

- ***readme.txt*** A text file with the latest news of the current distribution
- ***history.txt*** A text file with the revision history of the MDIS4 System Package
- ***tree.txt*** A text file with the file tree of the compressed files (excluding the MDIS4 User Guide)

The uncompressed files are located in the main ZIP file of the package.

Compressed Files

- The MDIS4 User Guide (this file).
- All MDIS API libraries (static libraries as well as DLLs) required to build application programs for MDIS.
- All currently available board drivers for Windows.
- *DESCGEN* — The MDIS descriptor generator, which converts common descriptor files (*.dsc*) in *.reg* files for the Windows NT registry.
- *MDISNT* — A test and configuration utility for MDIS (including sources).
- *MAPIVB* — A Visual Basic test and example program for MDIS, which demonstrates the usage of MDIS API DLLs and MDIS drivers with Visual Basic (including sources).
- The MDIS4 Package Installer — A program that simplifies the installation procedure of driver packages for Windows.

The compressed files are included in the cabinet files (*.cab*) of the package.

Example Folder Tree of the MDIS4 System Package for Windows NT/ W2000/XP/Embedded

```
|-NT                     === Windows NT4/W2k/XP stuff ===
|  |
|  |-DRIVERS             --- Driver folder ---
|  |  |-BBIS                 BBIS board drivers
|  |  |  `-D201              D201 BBIS
|  |  |     |-DOC            Documentation
|  |  |     `-DRIVER         Driver sources
|  |  |
|  |  `-MDIS_LL              MDIS device drivers
|  |     `-MT                MT test driver
|  |        |-DOC            Documentation
|  |        |-DRIVER         Driver sources
|  |        `-TOOLS          Tools
|  |           `-MT_BENCH    MT_BENCH tool sources
:  :
:  :
|  |-INCLUDE             --- Include folder ---
|  |  |-COM
|  |  |  `-MEN               Common headers
|  |  `-NATIVE
|  |     `-MEN               Native headers
:  :
:  :
|  |-LIBSRC              --- Library folder ---
|  |  `-MDIS_API             MDIS-API library
|  |     `-DOC               Documentation
:  :
:  :
|  |-MAKETMPL            --- Makefile templates ---
|  |     `...
:  :
:  :
|  |-OBJ                 --- Object folder ---
|  |  |-DLL                  DLLs and corresponding import LIBs
|  |  |  `-MEN
|  |  |     `-I386
|  |  |        |-CHECKED     Checked builds (debug)
|  |  |        `-FREE        Free builds (non-debug)
|  |  |
|  |  |-EXE                  Executables
|  |  |  `...
|  |  |-LIB                  Static libraries
|  |  |  `...
|  |  `-SYS                  NT4 drivers
|  |     `...
:  :
:  :
```

```
|  |-TOOLS              --- Tools folder ---
|  |  |-MDISAPP             MDISAPP example program
|  |  |  |-DOC              Documentation
|  |  |  |-NMAKE            NMAKE specific files
|  |  |  `-VC               VC specific files
|  |  |
|  |  `-MDISNT              MDISNT test program
:  :
:  :
|  `-VB                 --- Visual Basic folder ---
|     `-MAPIVB              MAPIVB example program
|
`-W2K                  === Windows W2k/XP stuff ===
   |
   |-OBJ               --- Object folder ---
   |  |-DLL                 DLLs and corresponding import LIBs
   |  |  `-MEN
   |  |     |-CHK
   |  |     |  `-I386       Checked builds (debug)
   |  |     `-FRE
   |  |        `-I386       Free builds (non-debug)
   |  |
   |  `-SYS                 W2k drivers
   |     `...
   |
   `-TARGET_INSTALL     --- Target install folder ---
```

Note: Basically, all common sources are located in a subfolder named *COM*. However, the MDIS4 Package Installer places some Windows-specific files under the COM subfolders.

A 3 Installing MDIS4 on the Host System

The host system is the Windows NT/2000/XP based PC on which you want to develop your application. If you would like to develop your application directly on your target system (without a separate host), you can perform the host and target installation on your target system. However, this user manual proceeds with the assumption that you are using two separate physical systems as host and target.

The MDIS4 host installation comprises the installation of the MDIS4 System Package for Windows NT/2000/XP/Embedded as well as the installation of all the driver packages required for your application.

Note: The MDIS4 host installation does not install drivers that can be used on the host. It merely provides access to the drivers, libraries, executables and documentation in a folder tree.

A 3.1 Installing the System Package

The system package is contained in a single ZIP file (usually *13m00006.zip*).

To install the system package, proceed as follows:

- ☑ Check the *readme.txt* file of the system package for additional installation hints.
- ☑ Extract the ZIP file to an arbitrary empty folder (e. g. *C:\TEMP*). Then execute *setup.exe* in this temporary folder.
- ☑ After the welcome dialog you are prompted for the destination folder (usually *C:\WORK*).
  Note: From now on, *%WORK%* in this manual refers to your destination folder.
- ☑ Use the default selection of the components to be installed.

The installer performs the following steps:

- The setup program copies all selected components into the destination folder.
- Some environment variables (prefixed by *MEN_*) are set.
- A program folder is created in the *Start* menu (default: *Programs\MDIS4 for Windows*). From this program folder, called "MDIS4 program folder", you can browse all documentation files included in the system package. You can also execute the MDIS4 Package Installer program from here.

A 3.2 Installing a Driver Package

Use the MDIS4 Package Installer located in *%WORK%\NT\MDIS_INSTALLER\* to install an MDIS4 driver package for Windows or a native driver package for Windows.

Usually, a driver package consists of a single ZIP file (e. g. *13m06670.zip*) and a PDF user manual (e. g. *21m066-01.pdf*). If you want to install several driver packages, you have to install the packages one by one.

Note: If you have received the package via e-mail, make sure that the ZIP file does not contain further ZIP files, otherwise you may have received more than one article bound together in a single ZIP file. In this case, you have to unzip the single ZIP file manually and must install each article ZIP file with the related documentation files (if any) separately.

To install an MDIS4 package, proceed as follows:

☑ Move the driver package ZIP file and any related documentation files (*.pdf, .html, .txt*) to an arbitrary empty folder (e.g. *C:\MEN_MDIS_PKGS\13M066-70*). Take care to put files of only one article into the empty folder.

☑ Execute the MDIS4 Package Installer program from the MDIS4 program folder via the *Start* menu.

☑ Specify the path where the driver package is located and follow the setup instructions.

The MDIS4 Package Installer performs the following steps:

- Copy the content of the Driver Package to the *%WORK%* folder.
  Thereby, all Windows 2000/XP package description files (*.xml, .inf*) and Windows 2000 Plug & Play Drivers (*.sys*, free build) will be copied to folder *%WORK%\W2K\TARGET_INSTALL.*
- Search all folders of the installed package for *program*.mak* files. If a *program*.mak* file was found in a subfolder, the NMAKE\makefile—if it does not already exist—will be copied into this subfolder.
- Put links to the documentation files included in the package to the MDIS4 program folder.

A 3.2.1 Hints on Updating and Deinstalling

You can update an already installed MDIS4 System Package or driver package by installing a newer version over your current installation. The setup program copies the files from the package into the *%WORK%* folder and overwrites only the same files. No other files in the *%WORK%* tree will be overwritten or removed.

For a clean *%WORK%* folder structure it is sometimes recommended to remove the current MDIS4 installation before installing a newer MDIS4 System Package version. You can also specify a different *%WORK%* location for a new MDIS4 System Package installation, but consider that the MDIS4 Package Installer always uses the *%WORK%* location of the last system package installation as destination path for the driver packages.

You can deinstall the MDIS4 System Package via the *Windows Control Panel* ➤ *Add/Remove Programs* dialog. This deinstallation procedure removes only the MDIS4 System Package and not the driver packages installed through the MDIS4 Package Installer. Therefore, if you have installed driver packages and you want to remove the entire MDIS4 installation from your computer you must perform the following steps:

- ☑ Back up your development work from the *%WORK%* tree.
- ☑ Deinstall the MDIS4 System Package using the *Add/Remove Programs* dialog.
- ☑ Remove the *%WORK%* folder from your disk.
- ☑ Remove the MDIS4 program folder from the *Start* menu.

Note: Only the MDIS4 System Package setup program makes some entries in the registry which will be removed by the *Add/Remove Programs* deinstallation. The MDIS4 Package Installer does not store any additional information in the registry.

A 4 Installing the Target System

The installation of the target system on which you want to run your applications depends primarily on

- the target's Windows version
- the target's hardware configuration
- the applications that you want to use on the target.

The installation for a target test system comprises:

- The installation of all necessary drivers on the target system. This includes all device drivers for the devices you want to use as well as the board drivers for the boards where the devices reside. The following sub chapters describe driver installation in detail.
- The installation of *men_winspec.dll* (required) and some MDIS API DLLs (optional) if the MDIS4 programs are not statically linked with the installed MDIS API libraries. Refer to Chapter A 8 Writing Applications for MDIS on page 64.
- The installation of the desired MDIS4 tools, test and example programs. To do this, just copy the provided executables from the host to an arbitrary folder on the target.

For the target installation, you have to take all the required files (drivers, programs, DLLs, etc.) from your MDIS4 host installation. Therefore we recommend to establish a network connection between the host and the target during the development phase rather than using any exchangeable media. For your application distribution, the installation procedure of your entire application including all the necessary drivers, DLLs, etc., is up to you.

Note: To avoid confusion between a physical hardware device and a virtual software device we will use the terms hw-device and sw-device where necessary.

Note: MDIS4 for Windows does not require any special Windows NT Service Pack. However, we recommend to install the latest Windows Service Pack available from Microsoft on the target system.

A 4.1 VMEbus Systems and Swapping Drivers

MDIS for Windows supports VMEbus access only for MEN VMEbus systems (e.g. MEN A13) under W2k/XP but not under NT4.

Targets running Windows NT are always x86 based CPUs (Little Endian oriented) and have mostly a (Compact)PCI bus where the MDIS supported devices (e.g. M-Module carrier boards) reside.

However, a target system may provide a VMEbus that is connected to the x86 CPU via a PCI-to-VMEbus bridge. This bus bridge may or may not swap the data between the two buses because of the "Big/Little Endian Hell".

Swapping means that each 16-bit or 32-bit access is byte-swapped and that the offsets to registers are not as expected on this kind of CPU. Swapping can be made by hardware (e.g. a PCI-to-VME bus bridge) or software (e.g. the swapped variants of an MDIS driver).

The MDIS4 System Package includes the standard board drivers (e.g. *men_a201.sys*) plus swapped variants (e.g. *men_a201_sw.sys*) for all supported VMEbus boards. The Windows MDIS driver packages include the standard device driver (e.g. *men_m66.sys*) and usually also a swapped variant (e.g. *men_m66_sw.sys*).

If swapped variants of the MDIS drivers are required or not, depends on the used VMEbus system (PCI-to-VMEbus bridge) and the corresponding drivers. Please refer to the documentation of your VMEbus system.

No swapped MDIS driver variants are required for the currently available MEN VMEbus systems (e.g. MEN A13).

In general, if you are using the swapped variant of a carrier board driver (e.g. *men_a201_sw.sys*) then you also have to use the swapped variants of the device drivers (e.g. *men_m66_sw.sys*) for the devices (e.g. M66 M-Module) plugged on the carrier (e.g. A201 VMEbus carrier).

If the swapping variant of a Windows 2000 PnP board driver was installed, the Found New Hardware Wizard will search for swapping variants of the subsequent required device drivers rather than for non-swapping variants.

A special VME4WIN driver from MEN must be installed for the used VMEbus system prior to the installation of any drivers for devices residing on the VMEbus. Please contact MEN to obtain the suitable VME4WIN Software Package for your MEN VMEbus system.

Refer to the documentation of the used VME4WIN driver and VMEbus carrier board driver for further information about the driver requirements and installation procedure.

A 4.2 Choosing the Right Windows Driver Type

MEN provides two different driver types: Windows NT 4.0 and Windows 2000 Plug & Play (PnP) drivers.

What the two Driver Types have in Common

- The drivers are designed as kernel-mode drivers that can directly control and access hardware devices.
- The execution of a driver can be preempted by higher-priority threads or interrupted by interrupts.
- The configuration data for the drivers are stored in the Windows registry.
- The drivers use Windows's event-log service to enter detailed error messages and other information into the Windows event log. The messages can be viewed using the Windows Event Viewer.
- Besides the normally used free build version of a driver, there is a checked build version which can produce debug print output for error debugging. The debug strings can be viewed using one of the free debug monitor programs. Refer to Chapter A 9.1.8 Displaying Debug Output from Checked Modules on page 78.

⚠ The drivers are **not** designed for multiprocessor systems, therefore your target system must be a uniprocessor system.

⚠ You must be logged on as an administrator or as a member of the Administrators group in order to install a driver.

Windows NT 4.0 Drivers or Windows 2000 PnP Drivers?

Bevor the target installation, you have to decide which driver type applies to your target system. The general rule is:

- If the target runs Windows NT 4.0 you must use the provided Windows NT 4.0 drivers.
- If the target runs Windows 2000 or Windows XP you should use the provided Windows 2000 PnP drivers.

However, in some cases and under certain circumstances it may be better to use the provided Windows NT 4.0 drivers for a target running Windows 2000 or Windows XP. To make a decision, you have to consider the following facts:

- The provided Windows NT 4.0 drivers runs only under Windows 2000/XP if Windows is installed without ACPI Support. To verify this, open *Computer Management* then select *Device Manager* and expand the *Computer* tree. If ACPI support is installed you will see an appropriate description. If Windows 2000/XP is installed without ACPI you will see the name *Standard PC*.
- For the Windows NT 4.0 driver installation, you have to create configuration files (*.reg*) which depend on the target's hardware configuration. Among others, you have to specify a unique fixed device name for each device in the configuration file.
- The Windows 2000 PnP drivers provide an easy Plug & Play installation without any necessary configuration tasks before driver installation. Usually, all installed hw-devices will be automatically recognized. An auto-generated alterable unique device name will be assigned to each installed sw-device.

- The driver and device configuration can be modified with a registry editor (NT4 drivers) or via a property sheet (W2k PnP drivers). Win32 user mode application can modify the configuration via special Win32 calls (refer to the MSDN library for further information).
- If you have the intention to migrate from the Windows NT 4.0 drivers to the Windows 2000 PnP drivers, you have to re-link all MDIS applications that are statically linked with the MDIS-API library with MDIS-API library version >= 3.0.

Target Running Windows 2000/XP

It is not possible to mix both driver types (NT4/W2k) for a board and the corresponding devices. That means an NT4 device driver (e.g. for an M-Module) requires an NT4 board driver (e.g. for the carrier board where the M-Module resides). The same goes for W2k drivers.

Generally, you should not use MDIS4 NT4 drivers and MDIS4 W2k drivers together on one target system. Only if no native W2k driver is available for a device (e.g. a serial interface) should a native NT4 driver for this device be mixed with other W2k drivers on target systems without installed ACPI support. In this case, the device serviced by an NT4 driver cannot be plugged on a carrier serviced by a W2k board driver. This means you will require separate carrier boards for devices serviced by NT4 device drivers and for devices serviced by W2k device drivers.

If you intend to mix the NT4 and W2k version of the same driver (e.g. for two D201 carrier boards) you must rename the NT4 driver (e.g. *men_d201.sys*) to a unique name (e.g. *men_d201nt4.sys*). Furthermore, you have to change the *HW_TYPE* (e.g. *D201*) in the corresponding descriptor file (e.g. *d201_min.dsc*) to the chosen unique name (e.g. *D201NT4*). Otherwise, you will get name conflicts.

A 4.3 Installing Windows NT 4.0 Drivers

For the event-log feature, you have to perform the following step once for your target:

Copy the *men_evlg.dll* file from the *%WORK%\W2K\TARGET_INSTALL* path of your host system to the *%SystemRoot%\System32* system path of the target system. This DLL will be used by the Windows Event Viewer to display the MEN specific descriptions that correspond to the event entries logged by the MDIS drivers.

To install a driver on your target system, proceed as follows:

☑ Modify the driver-specific common meta descriptor (*.dsc*) according to your target's hardware configuration, then use the *DESCGEN* descriptor generator to generate the *.reg* Windows NT descriptor from the modified *.dsc* descriptor. For further information about descriptor files and how they can be created, refer to Chapter A 5.4 NT4 Driver Descriptor Files on page 41.

☑ Copy the driver executable image file (*.sys*) from the *%WORK%\NT\OBJ\SYS\MEN\I386\FREE* path of your host system to the *%SystemRoot%\System32\Drivers* system path of the target system.

☑ Insert the driver parameters from the generated Windows NT descriptor into the registry. To do this, you must copy the generated *.reg* file to an arbitrary path (e. g. *C:\MEN\REG*) of the target system. To enter the information into the registry the file can simply be double-clicked or run from the command prompt.

N.B.:

If you plan to clone driver parameters from an already installed target system to another target with an exported *.reg* file from the *regedit* registry editor, please observe the following issues:

- Pay attention to copy the corresponding registry entries under *\Registry\Machine\System\CurrentControlSet\Services\EventLog\System\men_** as well.
- Don't copy the *Enum* subkey from the source target to the destination target. (See Chapter Subkey Enum on page 40).

☑ After adding a new driver key to the registry, you must reboot Windows NT to register the new driver. However, it is not necessary to reboot Windows NT after modifications of driver parameters in the registry for an already registered driver!

☑ If the driver is not configured to start automatically, you have to start the driver manually before you are able to access the devices belonging to the driver. Refer to Chapter A 5.1 Starting and Stopping NT4 Drivers on page 36.

Note on *.ini* Files

If the driver comes with common meta descriptor files (*.dsc*) it is strongly recommended to generate a Windows NT descriptor file (*.reg*) and to proceed as described above. However, some older native Windows NT drivers come only with *.ini* files, which contain the driver parameters. Chapter A 5.4 NT4 Driver Descriptor Files on page 41 describes how you can enter the content of *.ini* files into the registry.

Note on Board Drivers for CompactPCI/PCI M-Module Carrier Boards

In the board descriptor, the location of each CompactPCI or PCI board (e. g. C203, D201, F201) on the PCI bus must be specified. If you use descriptor key ***PCI_BUS_SLOT*** to specify the geographical location of the board on a PCI bus, you need additional parameters in the registry under the so-called PCI key (*HKEY_LOCAl_MACHINE\SOFTWARE\MEN\PCI*). For a detailed description of the PCI key, refer to Chapter A 5.4.2.6 CompactPCI/PCI M-Module Carrier Board Driver Keys on page 48.

Note on Deinstalling Drivers

Normally it is not necessary to deinstall a driver, because you can disable any driver as described in Chapter A 5.3 NT4 Driver Standard Parameters on page 39. However, you can deinstall a driver on your target system as follows:

☑ Remove the driver executable image file (*.sys*) from the system path of the target system (*%SystemRoot%\System32\Drivers*).

☑ Remove the driver parameters from the registry. To do this, use a registry editor, go to the services key (*HKEY_LOCAL_MACHINE\HARDWARE\SYSTEM\CurrentControlSet\Services*) and remove the driver key (e. g. *men_m66*).

A 4.4 Installing Windows 2000 PnP Drivers

This chapter describes some necessary basics about Windows 2000 PnP Drivers, the Windows 2000/XP PnP mechanism and the procedure of the Windows 2000 PnP Driver installation for the MEN hardware on your target system.

Please be aware that some described procedures are different for Windows 2000 and Windows XP as annotated.

The Device is the Starting Point

The Windows 2000 PnP Driver installation sets the focus on the devices and not on the drivers. This means that the installation of a driver is initiated by the installation of a sw-device: You have to install a sw-device for each installed hw-device you want to use. If you are adding a hw-device to your target system, a new sw-device must be installed but not necessarily a new driver.

Furthermore, you are able to disable/enable an installed sw-device but Windows 2000 PnP Drivers cannot be manually started/stopped like Windows NT 4.0 drivers.

Device Manager

The Windows 2000/XP *Device Manager* is the essential tool to manage devices. It provides you with information about how the hardware on your computer is installed and configured, and which drivers are used for the devices. Through *Device Manager* you can verify a device installation, update drivers, modify device settings, and troubleshoot problems.

The *Device Manager* is integrated in the *Computer Management*, a collection of administrative tools. To open *Computer Management*:

☑ Right-click on the *My Computer* desktop icon and select *Manage.*

Or

☑ Open the *Control Panel*, choose *Administrative Tools* and then *Computer Management*.

Unsigned Drivers

The provided Windows 2000 PnP drivers are generally not digitally signed by the MS Windows Hardware Quality Labs (WHQL) because most of our drivers belong to MEN-specific driver classes that are not supported by WHQL signing. Furthermore, MEN has no intention to use the cost- and time-consuming WHQL driver signing.

For unsigned drivers of certain driver classes you might see a warning during driver installation or update. In this case you have to use option *Continue Anyway* to complete the driver installation.

Note: Windows 2000/XP offer a *Control Panel* option to allow all device drivers to be installed without warnings, regardless of whether they have a digital signature. If you want to set this option, refer to the operating system's *Help*.

A 4.4.1 W2k Driver & PnP Basics

For a better understanding of the driver installation, you should know some basics about the Windows 2000/XP PnP mechanism.

PnP and non-PnP Devices

Devices can be roughly divided into two groups: PnP and non-PnP devices.

A PnP device is a hw-device that can be automatically detected by a so-called bus driver that is responsible for the bus/interface where the device resides. If a new PnP device was installed, Windows detects the device automatically and searches for a matching driver. If a proper driver was found on the computer, a corresponding sw-device will be installed without any user interaction. Otherwise, Windows starts up the *Found New Hardware Wizard*, which prompts the user for a driver.

A non-PnP device is a hw-device that cannot be automatically detected. If a non-PnP device was installed, you must use the *Add/Remove Hardware Wizard* (W2k) or *Add Hardware Wizard* (XP) in *Control Panel* to tell Windows what type of device you are installing. The *Add(/Remove) Hardware Wizard* may ask you to select the correct driver for the device.

MEN PnP Devices

Usually, the following type of MEN hw-devices are PnP devices and will be detected by Windows:

- PC•MIPs, PMC modules, PCI onboard devices
- (Compact)PCI carrier boards for M-Modules
- ISA PNP onboard devices
- M-Modules, after the PnP driver installation of the corresponding carrier

The installation of PnP devices is described in Chapter A 4.4.3 Installing PnP Devices on page 27.

MEN PnP Devices that Require a User-Initiated Installation

For a few PnP hw-devices a user-initiated installation (as for non-PnP devices) is required. This is the case whenever one hw-device is used for standard tasks (serviced by a standard driver) and also for OEM-specific features which require special drivers. In this case, Windows installs its standard driver for the hw-device but does not know that further drivers are required for the OEM-specific features (virtual hw-devices). The user will not be promted to install further drivers for the hw-devices. In this case a bus driver which enumerates the virtual hw-devices must be manually installed via the *Add(/Remove) Hardware Wizard.*

For example:

On MEN's F7/F7N/D4/EM02(EM05) CPU Board, the ICH82801(ALI1535) chipset contains among others the registers for the watchdog functionality.

Since Windows installs its own driver for the chipset during the Windows setup, no further drivers will be installed by PnP for the chipset.

The user first has to install the generic ISA(PCI) BBIS bus driver for the watchdog through the *Add/Remove Hardware Wizard*. The MDIS4 driver packages for the F7/F7N/D4/EM02(EM05) watchdog includes the proper *.inf* file for manual installation of the ISA(PCI) BBIS driver. The ISA(PCI) BBIS driver itself is included in the

MDIS4 System Package for Windows. After installation of this bus driver, the *Found New Hardware Wizard* appears for the virtual watchdog device and asks for the proper watchdog device driver.

MEN non-PnP Devices

The following type of MEN hw-devices are non-PnP devices and require manual installation via the *Add(/Remove) Hardware Wizard*:

- ISA devices
- VMEbus devices

The installation of non-PnP devices is described in Chapter A 4.4.4 Installing Non-PnP Devices on page 29.

A 4.4.2 Providing the Installation Files

The files required for the installation of Windows 2000 PnP drivers are located in the host's target install folder (*%WORK%\W2K\TARGET_INSTALL*) and consist of common installation files and driver package specific installation files:

The common installation files are installed on the host during the system package installation and comprise:

- *men_evlg.dll*— MEN event log DLL
- *men_mdis_clinst.dll*— MEN class installer DLL
- *men_qt-mt.dll*— QT C++ class library

The driver package specific installation files are installed on the host during the driver package installation and comprise:

- *<article no.>.inf* (e.g. *13m03606.inf*) — Driver Package installation file
- *<article no.>.xml* (e.g. *13m03606.xml*) — Driver Package description file
- *<driver name e.g. men_m36>.sys*— Windows 2000 PnP Driver

Note: A driver package may contain several *.inf/.xml/.sys* files.

Note: Common board drivers (e.g. C204, D203, ..) have no description files.

For the target installation, you have to provide the installation files from your host's target install folder to the target. However, the common installation files must only be supplied during the installation of the first PnP device belonging to the setup class "MDIS devices" (e.g. an M-Module) and also during the installation of the first PnP device belonging to the setup class "BBIS boards" (e.g. an M-Module carrier). On the first installation, the corresponding device setup class (MDIS devices or BBIS boards) will be registered and the common installation files are installed on the target. Therefore, the common installation files are not necessary during the installation of further MDIS devices or BBIS boards.

If the target has access to the host via a network connection (recommended during your development phase): Share the host's *%WORK%\W2K\TARGET_INSTALL* folder with the target or copy the content of the host's target installation directory to an arbitrary folder on the target.

If you have no network connection between the host and target: Move the required installation files of the driver package you want to install and the common installation files—if necessary—to the target using an exchangeable medium.

A 4.4.3 Installing PnP Devices

The *Found New Hardware Wizard* will guide you to the installation of a PnP device if the required driver for the new plugged hw-device could not be found on the target.

If you are plugging a further hw-device of the same model as an already present hw-device and a sw-device for the present hw-device was installed you will not be prompted by the *Found New Hardware Wizard* because the new hw-device will be installed automatically.

Windows 2000

When the *Found New Hardware Wizard* appears:

☑ Click *Next*.

When the *Install Hardware Device Drivers* window appears:

☑ Take over the default selection *Search for a suitable driver for my device (recommended)* and click *Next*.

When the *Locate Driver Files* window appears:

☑ Uncheck all options except *Specify a location* and click *Next*.

When the drive selection window appears:

☑ Browse to the location of the driver installation files and click *OK*.

If Windows has found more than one matching driver, the *Driver Files Search Results* window appears with the message "*The wizard also found other drivers that are suitable for this device*:"

☑ Select *Install one of the other drivers*, otherwise the first driver found will be installed (this may be the wrong driver) and you will not be prompted to select the correct driver.

☑ Click *Next*.

Now the *Driver Files Found* window appears:

☑ Select the correct driver from the list. Please ignore the confusing *Recommended driver* message that appears if the first listed driver is selected.

☑ Click *Next*.

If Windows has found only one matching driver the *Driver Files Search Results* window appears without a message about other suitable drivers:

☑ Click *Next*.

When the *Completing the Found New Hardware Wizard* window appears:

☑ Click *Finish*.

Windows XP

When the Found New Hardware Wizard appears:

☑ Select *Install from a list or specific location (Advanced).*

☑ Click *Next*.

When the *Please choose your search and installation options* window appears:

☑ Select *Search for the best driver in these locations.*

☑ Select *Include this location in the search* and browse to the location of the driver installation files.

☑ Click *Next*.

If Windows has found only one matching driver, this will be installed immediately.

If Windows has found more than one matching driver, the *Please select the best match for your hardware from the list below* window appears:

☑ Select the correct driver from the list. Please ignore the warning "*This driver is not digitally signed*".

☑ Click *Next*.

When the *Completing the Found New Harware Wizard* window appears:

☑ Click *Finish*.

A 4.4.4 Installing Non-PnP Devices

The *Add/Remove Hardware Wizard* (W2k) or *Add Hardware Wizard* (XP) will guide you through the installation of non-PnP devices.

The installation of Non-PnP devices is necessary for bus drivers as described in Chapter MEN PnP Devices that Require a User-Initiated Installation on page 25.

If you are installing a bus driver, then the bus driver will inform Windows about all new found PnP devices on its served bus. Therefore the *Found New Hardware Wizard* may apper after the installation of a non-PnP device and you have to perform the installation for each new found PnP device according to Chapter A 4.4.3 Installing PnP Devices on page 27.

Windows 2000

☑ To open the *Add Hardware Wizard*, go to *Start* ➢ *Settings* ➢ *Control Panel* ➢ *Add/Remove Hardware*.

When the *Add/Remove Hardware Wizard* appears:

☑ Click *Next*.

When the *Choose a Hardware Task* window appears:

☑ Take over the default selection *Add/Troubleshoot a device*.

☑ Click *Next*.

When the *Choose a Hardware Device* window appears:

☑ Select *Add a new device* from the list.

☑ Click *Next*.

When the *Find New Hardware* window appears:

☑ Select *No, I want to select the hardware from a list*.

☑ Click *Next*.

When the *Hardware Type* window appears:

☑ Select *Other devices* from the list.

☑ Click *Next*.

When the *Select a Device Driver* window appears:

☑ Click *Have Disk...*

☑ Browse to the location of the driver installation files.

☑ Select *MEN Mikro Elektronik* as manufacturer.

☑ Select the model of the hardware device you want to install.

☑ Click *Next*.

When the *Start Hardware Installation* window appears:

☑ Click *Next*.

When the *Completing the Add/Remove Hardware Wizard* window appears:

☑ Click *Finish*.

Windows XP

☑ To open the *Add Hardware Wizard*, go to *Start* ➤ *Control Panel* ➤ *Add Hardware*.

When the *Add Hardware Wizard* appears:

☑ Click *Next*.

☑ Wait while the wizard searches and press the *Cancel* button on each *Welcome to the Found New Hardware Wizard* window that will appear. A few *Welcome...* windows may appear and you have to wait until the wizard has finished the search.

When the *Is the hardware connected?* window appears:

☑ Select *Yes, I have already connected the hardware*.

☑ Click *Next*.

When the *The following hardware is already installed on your computer* window appears:

☑ Select *Add a new hardware device* from the list.

☑ Click *Next*.

When the *The wizard can help you to install other hardware* window appears:

☑ Select *Install the hardware that I manually select from a list (Advanced)*.

☑ Click *Next*.

When the *From the list below, select the type of hardware you are installing* window appears:

☑ Select *Show All Devices* from the list.

☑ Click *Next*.

When the *Select the device driver you want to install for this hardware* window appears:

☑ Click *Have Disk...*

☑ Browse to the location of the driver installation files.

☑ Select *MEN Mikro Elektronik* as manufacturer.

☑ Select the model of the hardware device you want to install.

☑ Click *Next*.

When the *The wizard is ready to install your hardware* window appears:

☑ Click *Next*.

When the *Completing the Add Hardware Wizard* window appears:

☑ Click *Finish*.

A 4.4.5 Reinstalling and Updating W2k PnP Drivers

You must manually initiate reinstalling or updating of drivers through the *Device Manager*.

The *Upgrade Device Driver Wizard* (W2k) or *Hardware Update Wizard* (XP) guides you through the process of reinstalling or updating a driver. Thereby, you have to perform the same procedure as during the *Found New Hardware Wizard* installation. See Chapter A 4.4.3 Installing PnP Devices on page 27.

A 4.4.5.1 Reinstalling a Driver for a PnP Device

Reinstalling a driver is required if the installation for a PnP device with the *Found New Hardware Wizard* was canceled and the sw-device is not yet installed.

In the *Device Manager*, a yellow question mark will appear next to the incompletely installed sw-device under the *Other Devices* group. Double-click on the device you want to reinstall, then click the *Reinstall Driver* button on the *General* tab to start the *Wizard*.

A 4.4.5.2 Updating a Driver for a PnP or non-PnP Device

Updating a driver is required if you want to upgrade the driver to the latest version.

In the *Device Manager*, double-click on the device you want to update, then select the *Driver Tab*. The driver information shown shows the driver currently being used by the system. Click *Update Driver* to start the *Wizard*.

Note: You can also perform a manual "quick and dirty" driver update, if you know all the files belonging to the driver. This can be useful if you want to use the checked version of a driver for debugging for some time. To do this, just copy the checked driver version (e. g. *%WORK%\W2K\OBJ\SYS\MEN\CHK\I386\men_d201.sys*) from the host to the target’s driver’s path (*%System-Root%\System32\Drivers*) and then reboot the target.

A 4.4.6 Notes for Hardware Device Configuration Changes

The following examples point out how hw-device configuration changes affect the sw-device's presence and installation. This PnP behavior could be an advantage but also a drawback for your target application.

You should consider this behavior if you intend to alter the hw-device configuration on your target.

Example 1—Plugging a new hw-device type

If a new type of hw-device was plugged (e.g. the first M66 M-Module):

- The *Found New Hardware Wizard* prompts you for the sw-device installation.
- The corresponding sw-device will be named <dev-type>_1 (e.g. *m66_1*).

Example 2—Plugging a further hw-device of an existing hw-device type

If a further hw-device of the same type as an already installed hw-device was plugged (e.g. a second M66 M-Module):

- The new sw-device is automatically installed.
- The corresponding sw-device will be named <dev- type>_2..n (e.g. *m66_2*).

Example 3—Removing or replacing a hw-device

If a hw-device (e.g. an M66 M-Module) was removed or replaced by a hw-device of another type (e.g. an M36 M-Module):

- The corresponding sw-device (e.g. *m66_1*) vanishes from the *Device Manager* tree.
- The name (e.g. *m66_1*) and settings (e.g. descriptor parameter *ID_CHECK*) of the sw-device remain in the registry.

Example 4—Replugging a hw-device

If a hw-device (e.g. an M66 M-Module) was replugged to its old location:

- The corresponding sw-device (e.g. *m66_1*) reappears in the *Device Manager* tree.
- The old name (e.g. *m66_1*) and settings (e.g. descriptor parameter *ID_CHECK*) of the sw-device will be taken over from the registry.

Example 5—Moving a carrier with plugged mezzanines

If a carrier board (e.g. a D201 M-Module carrier) with plugged hw-devices (e.g. two M66 M-Modules) was moved from one slot to another:

- The *Found New Hardware Wizard* prompts you for the sw-device installation of the moved carrier board.

Note: No auto-installation happens because the carrier may not be definitely recognized and you have to select the right carrier board type manually.

- The old carrier board sw-device with its instance number (e.g. *#1*) and device parameters vanish from the *Device Manager* tree and the newly installed sw-device for the carrier board appears (e.g. *#2*).
- All sw-devices (e.g. *m66_1* and *m66_2*) related to the plugged hw-device on the carrier at the previous location vanish from the *Device Manager* tree. The names (*m66_1* and *m66_2*) and settings (e.g. descriptor parameters) of the vanished sw-devices remain in the registry and will be re-used if the carrier board is again moved to the previous location.
- New sw-devices will be installed (automatically) for the plugged hw-device on the carrier and therefore the device names will be changed (e.g. to *m66_3*, *m66_4*). The device settings (e.g. descriptor parameters) of the new sw-devices will be set to the defaults.

Example 6—Device name pitfall

If you change a device name to a name of a sw-device that was not uninstalled (e.g. *m66_2*) and whose related hw-device was removed (the corresponding sw-device *m66_2* is not visible) and then replug the removed hw-device, a second sw-device with the same name (e.g. *m66_2*) will appear in the *Device Manager* tree.

In this case, you have to change the name of one of the two (*m66_2*) sw-devices to a unique device name. However, to prevent you from accessing the wrong device, in this case the MDIS-API *M_open()* function will return the error `0x04B0` "*The specified device name is invalid*".

A 5 NT4 Drivers and Device Configuration

This chapter gives you detailed information on how Windows NT 4.0 drivers can be started/stopped and which registry entries are involved in the driver configuration. Furthermore, the Windows NT 4.0 driver-specific descriptor files and their generation are explained.

Configuration Data

The driver and device configuration data comprise:

- Windows NT 4.0 driver-specific parameters, described in Chapter A 5.3 NT4 Driver Standard Parameters on page 39.
- MDIS4 standard descriptor parameters, described in Chapter B 4 MDIS Device Descriptors on page 112.
- Low-level driver-specific descriptor parameters, described in the driver's user manual.
- Windows NT 4.0 driver-specific MDIS4 parameters, described in Chapter A 5.4.2 NT4 Driver-Specific MDIS Keys on page 46.

Configuration Location & Organization

The configuration data is located in the Windows registry and arranged like a folder tree with "registry keys" (like a directory folder) and "registry values" (like a file). Each registry value uses one of the registry-specific data types (e.g. *REG_DWORD*) and stores the actual configuration data.

For each driver, a main registry key will be created during the driver installation under *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services*. The main key is named after the driver's name (e.g. *men_m22*). The Windows NT 4.0 driver-specific parameters (e.g. *Start*, *ErrorControl*, ...) are directly stored under the driver's main key. Device-specific parameters (e.g. *VALID*) are stored under a device sub-key (e.g. *m22_1*) that specifies the device name. All MDIS4 standard descriptor parameters (e.g. *BOARD_NAME*) and parameters belonging to the low-level driver (e.g. *CHANNEL_0\INACTIVE*) are located under the *Parameters* subkey beneath the device-related key.

Example for the *men_m22.sys* driver, viewed with the *regedt32* registry editor:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\men_m22
Start: REG_DWORD: 0x3
Group: REG_SZ: MEN_MODULE_DRIVER
DependOnGroup: REG_MULTI_SZ: MEN_BOARD_DRIVER
ErrorControl: REG_DWORD: 0x1
Type: REG_DWORD: 0x1
m22_1
      VALID: REG_DWORD: 0x1
      ...
      Parameters
         BOARD_NAME: REG_SZ: D201_1
         ...
         CHANNEL_0
            INACTIVE: REG_DWORD: 0x1
            ...
```

Configuration Modification

The configuration data can be modified before driver installation in the descriptor file (*.dsc/.reg*) or after driver installation in the registry using a registry editor. To view, create, modify or delete registry keys and values, use one of the two Windows NT 4.0 registry editors:

- *regedt32.exe*
  This registry editor is tied tightly to the Windows NT platform and understands certain data types that are unique to Windows NT.
- *regedit.exe*
  A registry editor useful for searching a textual value in the registry and to import and export registry files (*.reg*).

A 5.1 Starting and Stopping NT4 Drivers

Windows NT 4.0 kernel mode drivers are started according to information in the Windows NT registry. If the *Start* parameter is set to *Automatic* (`0x02`) the driver will be started during system start-up. The driver start may also depend on other drivers (refer to Chapter A 5.2 Driver Dependencies on page 38). However, you can start and stop drivers manually:

A 5.1.1 Starting Drivers Manually

If the *Start* value is not set to *Disabled* (`0x04`), you can start the driver manually

- from the command prompt, using command *net start <drivername>* or
- from the Device Control Panel Applet or
- from an application, using the Win32 *StartService* function.

A 5.1.2 Stopping Drivers Manually

If the driver is not in use, you can stop it manually

- from the command prompt, using command *net stop <drivername>* or
- from the Device Control Panel Applet or
- from an application, using the Win32 *ControlService* function.

If you try to stop a driver and the driver is already in use (a handle is still open to a device serviced by the driver) the driver stop fails. In this case, as soon as all handles to the device are closed the driver unloads automatically.

Example #1, "device driver is being used by an application"

- Driver *men_m22* is started.
- A handle from an application to device *m22_1* is opened.
- Try to stop the driver from the command prompt using command *net stop men_m22*.
  ⇨ The following error message appears:

  ```
  The men_m22 service could not be stopped.
  ```

- Close all opened handles to the devices serviced by the *men_m22* driver.
  ⇨ The driver stops.

Example #2, "board driver is being used by a device driver"

- Board driver *men_d201* is started (serviced board: *d201_1*).
- Device driver *men_m22* is started (serviced device: *m22_1* located on *d201_1*).
- Try to stop the board driver: *net stop men_d201*
  ⇨ The following error message appears:

  ```
  The men_d201 service could not be stopped.
  ```

- Try to stop the device driver: *net stop men_m22*
  ⇨ The following success message appears:

  ```
  The men_m22 service was stopped successfully.
  ```

  ⇨ The *men_d201* driver is also stopped.

The behavior described in example #2 is only valid for device drivers built with MK library rev. 1.17 or higher and board drivers built with BK library rev. 1.8 or higher. You can query the library revisions of a driver through the *MDISNT* utility. (Refer to Chapter A 9.1.4 Getting Revision Information on MDIS Modules on page 77.)

Note: If the drivers use older MK or BK libraries, you may get a blue screen if you try to stop a board driver which is still being used by a device driver. Therefore, be careful if you want to stop a board driver!

A 5.2 Driver Dependencies

Commonly, MDIS drivers specify the following driver dependencies under their driver key in the registry:

Board Driver:	`Group=MEN_BOARD_DRIVER`
Device Driver:	`Group=MEN_MODULE_DRIVER, DependOnGroup=MEN_BOARD_DRIVER`

At start-up, when a device driver with a *DependOnGroup=MEN_BOARD_DRIVE*R is started (automatically or manually), Windows NT will attempt to start all the board drivers in the prerequisite group (with a *Group=MEN_BOARD_DRIVER*) that have not already been started and are not explicitly marked as *Disabled (Start=0x04)*. If any of the board drivers starts, the device driver with the dependency will also be started. If none of the board drivers with a *Group=MEN_BOARD_DRIVER* starts successfully, the device driver with the dependency will not be started and Windows NT enters an appropriate entry in the Windows NT event log.

Note: It is not necessary to enter the driver groups *MEN_BOARD_DRIVER* and *MEN_MODULE_DRIVER* into the registry group list (*HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Service-GroupOrder\List*).

The start condition of a driver can also depend on one special driver:

When a device driver *DRV2* with a *DependOnService=DRV1* attempts to start, Windows NT will try to start *DRV1* (if *DRV1* is not explicitly marked as *Disabled*) before *DRV2*. *DRV2* will only be started if *DRV1* has started successfully.

Note: *DRV1* and *DRV2* are placeholders for driver names.
Example: The *men_lm78_f2.sys* device driver requires the *men_z8536_f2.sys* device driver. Therefore you must specify 'men_z8536_f2' as *DRV1* and 'men_lm78_f2' as *DRV2* to force that *men_lm78_f2.sys* will only be started if the *men_z8536_f2.sys* is running.

A 5.3 NT4 Driver Standard Parameters

This section only describes the standardized Windows values used by MEN's Windows NT 4.0 drivers. The MDIS-related entries are described in Chapter A 5.4 NT4 Driver Descriptor Files on page 41 and in Chapter B 4 MDIS Device Descriptors on page 112.

Start

This value indicates if the driver will be started automatically during the system startup, manually on request or if the driver cannot be started:

Table A1. *Registry Entry* Start

Hex Value in Registry	Startup Type in Device Control Panel Applet	Meaning
`0x02`	Automatic	Driver is started during system startup
`0x03`	Manual	Driver is started on request
`0x04`	Disabled	Driver cannot be started

The startup types *Boot* (`0x00`) and *System* (`0x01`) should not be used for MDIS-related drivers, the default value is *Manual* (`0x03`). The *Start* Value can be changed directly in the registry or via the Device Control Panel Applet.

Group

This value specifies a driver group to which the driver belongs. MDIS drivers use the following driver groups:

Table A2. *Registry Entry* Group

String Value in Registry	Used by
MEN_BOARD_DRIVER	board driver
MEN_MODULE_DRIVER	device driver

DependOnGroup

This value identifies a prerequisite driver group or specific driver on which the driver start-up depends. MDIS drivers use the following dependencies:

Table A3. *Registry Entry* DependOnGroup

String Value in Registry	Used by
MEN_MODULE_DRIVER	board driver
(none)	device driver

ErrorControl

This value determines which action the system takes if a driver fails to load successfully at auto-start:

Table A4. *Registry Entry* ErrorControl

Hex Value in Registry	Meaning
`0x00`	Load errors are ignored.
`0x01`	Load errors are logged to the system event log.
`0x02`	Load errors are logged to the system event log. The system is restarted by using Last Known Good configuration. If Last Known Good configuration is booted, load continues.
`0x03`	Load errors are logged to the system event log. The system is restarted by using Last Known Good configuration. If Last Known Good configuration is booted, the boot is aborted.

The default value for MDIS-related drivers is `0x01`.

Type

This value indicates the type of component that this entry represents and must always be `0x01` (kernel-mode driver) for MEN drivers.

Subkey *Enum*

Under the *Enum* subkey, Windows NT stores information about the hardware configuration of the devices controlled by a driver.

The operating system adds the *Enum* subkey during the first start of the related driver. The key **must not** be modified.

Note: If you save the driver's registry entries from a target in a *.reg* file for cloning another target with the same driver parameters, pay attention that the *Enum* subkey is not copied. Otherwise, it may be impossible to start the driver on the cloned target.

A 5.4 NT4 Driver Descriptor Files

Descriptors are operating system-specific files that hold parameters for drivers. For Windows NT, MDIS uses *.reg* files as descriptor files. The content of the descriptors must be entered in the Windows NT registry where the driver-related parameters must be stored.

The Windows NT descriptors are generated from the operating system-independent meta descriptors, which are ASCII files with file extension *.dsc*. A *.reg* descriptor is also an ASCII file that can be modified. However, the common descriptor parameters should only be edited in the *.dsc* files which contain additional comments for the parameters.

Apart from the common descriptor parameters held in the meta descriptors, Windows NT descriptors contain additional NT specific parameters (e. g. *Start*, *Group*, etc.). The descriptor generator automatically inserts the NT specific parameters into the *.reg* file.

The Windows NT specific parameters are described in Chapter A 5.4.2 NT4 Driver-Specific MDIS Keys on page 46. For a complete description of meta descriptors and keys refer to Chapter B 4 MDIS Device Descriptors on page 112.

Note on *.ini* Descriptors

Some older native Windows NT driver packages contain *.ini* files instead of meta descriptors (*.dsc*) or *.reg* descriptors. The *.ini* files hold the same information as the *.reg* files but use a different syntax.

Example of the *m45_min.ini* descriptor:

```
\Registry\Machine\System\CurrentControlSet\Services\men_m45
Type = REG_DWORD 0x00000001
Start = REG_DWORD 0x00000003
Group = REG_SZ MEN_MODULE_DRIVER
DependOnGroup = REG_MULTI_SZ MEN_BOARD_DRIVER
ErrorControl = REG_DWORD 0x00000001
m45_1
      VALID = REG_DWORD 1
      Parameters
         BOARD_NAME = REG_SZ "d201_1"
         DEVICE_SLOT = REG_DWORD 0
         ...

\Registry\Machine\System\CurrentControlSet\Services\EventLog\System\men_m45
EventMessageFile = REG_EXPAND_SZ
"%SystemRoot%\System32\IoLogMsg.dll;%SystemRoot%\System32\men_evlg.dll"
TypesSupported = REG_DWORD 0x00000007
```

Microsoft’s *regini.exe* tool, which is contained in the Microsoft DDK, automatically transfers the registry entries from the *.ini* file to the registry. For licensing reasons, however, MEN is not allowed to supply *regini.exe*.

To insert the content of the *.ini* file into the registry of your target system using *regini.exe*, proceed as follows:

☑ Copy the *.ini* file as well as *regini.exe* to an arbitrary path (e. g. *C:\MEN\REG*) of the target system.

☑ From the command prompt, go to the path where the copied files reside and call *regini.exe* followed by the name of the *.ini* file.
Example:

```
C:\MEN\REG> regini m50_min.ini
```

The registry entries can also be manually inserted into the registry, using the registry editor *regedit32* or *regedit*.

You must adapt the parameters to your system configuration – either in the *.ini* file before making the entry via *regini.exe* or directly in the registry.

A 5.4.1 Generating *.reg* Descriptors for NT4 Drivers

Use the *DESCGEN* descriptor generator to generate a Windows NT descriptor (*.reg*) from a common meta descriptor (*.dsc*). You will find *descgen.exe* in the executable path of your MDIS4 installation (*%WORK%\NT\OBJ\EXE\MEN\I386\FREE*). Use option *-winnt* to generate a *.reg* file for Windows NT.

For convenience, add the *%WORK%\NT\OBJ\EXE\MEN\I386\FREE* path to the *PATH* environment variable of your host system. Then you can call *DESCGEN* from every folder as implemented in the following examples.

To show you how to generate *.reg* descriptors, let’s assume you want to use two M66 M-Modules on one D201 M-Module carrier board.

A 5.4.1.1 Using Descriptor Templates

Usually, each MDIS4 driver comes with descriptor templates (e. g. *xxx_min.dsc*, *xxx_max.dsc*). The *min* file includes only the mandatory descriptor keys, while the *max* file includes all possible keys.

You will most likely need only the mandatory descriptor keys, but for special system and driver configurations you must also use some optional descriptor keys from the *max* descriptor. Refer to the corresponding driver manual for a description of optional driver-specific keys.

Basically, it depends on your personal preferences whether you use the *min* or the *max* descriptor as a template for your own descriptor. You can begin with the *min* descriptor as a starting point for your own descriptor and then add all optional keys you need from the *max* descriptor.

A 5.4.1.2 Generating a Board Descriptor

For a D201 M-Module carrier board, you will find the two meta descriptors to build the board descriptor:

```
%WORK%\NT\DRIVERS\BBIS\D201\DRIVER\COM:
d201_min.dsc
d201_max.dsc
```

☑ Make a copy of one of the templates and name it *d201_my.dsc*:

```
D201_1 {
#---------------------------------------------------------------
# general parameters (don't modify)
#---------------------------------------------------------------
DESC_TYPE = U_INT32 2       # descriptor type (2 = board)
HW_TYPE = STRING D201       # hardware name of device
#---------------------------------------------------------------
# PCI configuration
#---------------------------------------------------------------
PCI_BUS_PATH = BINARY 0x08  # device IDs of bridges to
                            # CompactPCI bus
PCI_BUS_SLOT = U_INT32 3    # CompactPCI bus slot (1 = CPU)
}
```

☑ Before generating the descriptor you must edit *PCI_BUS_PATH* and *PCI_BUS_SLOT* in this file to specify the location of the D201 on the PCI bus in your system. (See Chapter B 4.4.2 CompactPCI M-Module Carrier Boards on page 120.)

☑ Then generate the board descriptor:

```
%WORK%\NT\DRIVERS\BBIS\D201\DRIVER\COM>
descgen d201_my.dsc -winnt
```

This produces the following descriptor:

```
%WORK%\NT\DRIVERS\BBIS\D201\DRIVER\COM
d201_my.reg
```

A 5.4.1.3 Generating a Device Descriptor

After building the board descriptor, you must generate the device descriptor.

After installing the low-level package, you will find meta descriptors in the driver's main folder. For the M66 M-Module, you will find the two meta descriptors to build the device descriptor:

```
%WORK%\NT\DRIVERS\MDIS_LL\M066\DRIVER\COM:
m66_min.dsc
m66_max.dsc
```

☑ Make a copy of one of the templates and name the file *m66_my.dsc*.

☑ Now you have to edit *m66_my.dsc* before generating the descriptor.
For the simple case of the M66 M-Module, you only need to change *BOARD_NAME* and *DEVICE_SLOT* to match the device name and slot of the carrier board where the M66 M-Module is installed. In our example we will build a device descriptor (*m66_my.dsc*) for two M66 M-Modules from one meta descriptor:

```
M66_1 {
#-------------------------------------------------------------
general parameters (don't modify)
#-------------------------------------------------------------
DESC_TYPE = U_INT32 1 # descriptor type (1=device)
HW_TYPE = STRING M066 # hardware name of device
#-------------------------------------------------------------
base board configuration
#-------------------------------------------------------------
BOARD_NAME = STRING D201_1 # device name of base board
DEVICE_SLOT = U_INT32 0    # used slot on base board (0..n)
}

M66_2 {
#-------------------------------------------------------------
general parameters (don't modify)
#-------------------------------------------------------------
DESC_TYPE = U_INT32 1 # descriptor type (1=device)
HW_TYPE = STRING M066 # hardware name of device
#-------------------------------------------------------------
base board configuration
#-------------------------------------------------------------
BOARD_NAME = STRING D201_1 # device name of base board
DEVICE_SLOT = U_INT32 1    # used slot on base board (0..n)
}
```

☑ Then generate the device descriptor:

```
%WORK%\NT\DRIVERS\MDIS_LL\M066\DRIVER\COM>
descgen m66_my.dsc -winnt
```

This produces the following *.reg* descriptor:

```
%WORK%\NT\DRIVERS\MDIS_LL\M066\DRIVER\COM:
m66_my.reg
```

A 5.4.1.4 Generating a Multi-Driver Descriptor

In the preceding example, one *.reg* descriptor was generated for each driver. It is also possible to generate a *.reg* descriptor for several drivers, called a multi-driver descriptor. To do this, you must specify all *.dsc* files that are to be used to build the *.reg* file.

To show you how to generate multi-driver descriptors, let's assume you want to use two M66 M-Modules, one M55 M-Module and one D201 M-Module carrier board.

- ☑ For each driver, make a copy of one of the templates and name the files *m66_my.dsc, m55_my.dsc* and *d201_my.dsc*, and put them into an arbitrary folder (e. g. *C:\MYCONFIG*).
- ☑ Now you have to edit each file to specify the system configuration before generating the descriptor.
- ☑ Then generate the descriptor:

```
C:\MYCONFIG>descgen -winnt d201_my.dsc m66_my.dsc m55_my.dsc
```

This produces the following descriptor:

```
\ MYCONFIG:
d201_my.reg
```

The multi-driver descriptor file will be named after the first specified *.dsc* file (*d201_my*). However, *d201_my.reg* contains the configuration data of all the specified *.dsc* files.

A 5.4.2 NT4 Driver-Specific MDIS Keys

The generated *.reg* descriptors contain additional parameters that are Windows NT specific and are not present in the common meta descriptors. These parameters can be divided into two groups:

- Kernel-mode driver specific keys, which are described in Chapter A 5.3 NT4 Driver Standard Parameters on page 39.
- MDIS-related keys, which are described in this Chapter.

A 5.4.2.5 Device and Board Driver Keys

This section describes Windows NT specific MDIS keys, common to all device drivers and board drivers. The following figure shows the keys in the *regedt32* view:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\men_xxx
...
ADDRESS_SHARING: REG_DWORD: 0x03
DEBUG_LEVEL_ENTRY: REG_DWORD: 0xc0008000
mxx_1
      VALID: REG_DWORD: 0x1
         Parameters
            ...

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog\
System\men_xxx
EventMessageFile: REG_EXPAND_SZ: %SystemRoot%\System32\
IoLogMsg.dll;%SystemRoot%\System32\men_evlg.dll
TypesSupported: REG_DWORD: 0x7
```

VALID

The value indicates if the corresponding device parameters are valid (`0x1`) or not (`0x0`). It can be used to disable an individual device of a driver without removing the device specific keys from the registry.

For example, let's assume that we have two M66 M-Modules with the corresponding device names *m66_1* and *m66_2*. Now, we have to remove one M66 M-Module (*m66_1*) for testing purposes. We will set *VALID=0x00* so that the M66 driver does not try to create the *m66_1* device for the non-existing M66 M-Module:

```
HKEY_LOCAL_MACHINE\HARDWARE\SYSTEM\CurrentControlSet\Services\men_m66
...
m66_1
      VALID: REG_DWORD: 0x0
      ...
m66_2
      VALID: REG_DWORD: 0x1
      ...
```

DEBUG_LEVEL_ENTRY

This value specifies an additional driver specific debug level for the Windows NT specific *DriverEntry* routine which creates the corresponding devices. Like all debug levels, the parameter is only relevant for the checked version of a driver. The default value is *0xC0008000*. For a detailed description of debug levels, refer to Chapter B 4.5 Driver Debugging on page 124.

ADDRESS_SHARING

This value specifies if and how the operating system assigns the address resources used by the driver. The *ADDRESS_SHARING* key is supported by device drivers built with MK library rev. 1.10 or higher and board drivers built with BK library rev. 1.3 or higher. You can query the library revisions of a driver through the *MDISNT* utility. (Refer to Chapter A 9.1.4 Getting Revision Information on MDIS Modules on page 77.)

Table A5. *Driver Key* ADDRESS_SHARING

String Value in Registry	Meaning
`0x01`	Address resources assigned as device-exclusive resources.
`0x02`	Address resources assigned as driver-exclusive resources. The address resources can be shared by several devices of this driver.
`0x03`	Address resources assigned as shared resources. The address resources can be shared by several drivers.
`0xFF`	Address resources are not assigned. Windows NT cannot detect any address conflict between this driver and any other driver that uses the same address resources.

By default, this entry does not exist, and the address resources are assigned as device-exclusive resources (*0x01*). The *ADDRESS_SHARING* parameter should only be used for very special situations, e. g. if Windows NT detects an address conflict between a MEN driver and a third-party driver. This may happen if a VMEbus PC with a PCI-to-VME bridge is used and the driver must assign the same address resources that are assigned by a VME-to-PCI bridge specific driver. In this case you need detailed information about the used address resources and whether it is indicated that more than one device use the same address resources.

EventLog Entry

The entry ..\EventLog\System\men_xxx registers the *men_xxx* driver for the Windows NT event-log service. The respective parameters tell the event-log service that the *IoLogMsg.dll* and *men_evlg.dll* DLLs contain the descriptions belonging to the event entries logged by the *men_xxx* driver.

If the *men_evlg.dll* is not installed, a driver is not registered under ..\EventLog\System\\ or the Windows NT Event Viewer cannot display the event strings that correspond to the entries logged by the driver.

A 5.4.2.6 CompactPCI/PCI M-Module Carrier Board Driver Keys

This information applies to all board drivers for CompactPCI/PCI M-Module carrier boards, e. g.:

- *men_d201.sys*
- *men_c203.sys*, *men_c204.sys*
- *men_f201.sys*, *men_f202.sys*

First of all, remember the PCI-bus basics:

- One system can have up to 256 separate PCI busses (0..255), connected over PCI-to-PCI bridges.
- 32 physical units (0..31, called devices) can be plugged into one bus. Each CompactPCI/PCI slot on which a physical unit can be plugged is assigned to one PCI device ID (unique to the assigned PCI bus).
- Each device can contain up to 8 separate functional units (0..7, called functions).
- Interrupt mechanisms:
  The PCI bus has four equal-priority interrupt request lines (*INTA..INTD*) which are active-low, level-triggered, and shareable. Each function can be connected to only one request line. On a PC architecture, the redirector converts a given request on *INTA..INTD* into a request on one of the *IRQ0..IRQ15* lines. BIOS stores the assigned IRQ number in the interrupt line register of the PCI Configuration Space.
- Device memory:
  Dedicated memory used by PCI functions, which can reside anywhere in a 32-bit address space. Up to six address spaces can be assigned to one function. On a PC architecture, normally the BIOS assigns the requested memory spaces to the PCI function and stores address information in the Base Address Registers of the PCI Configuration Space.
- PCI Configuration Space:
  Each individual function has its own 256-byte storage area for configuration data, called PCI configuration space. The first 64 bytes of any PCI configuration space has a predetermined structure for common information (e. g. about vendor, device type, memory resources, interrupt). The PCI configuration space can be accessed through two special registers in the I/O address space.

The board driver of a CompactPCI/PCI M-Module carrier board must know the location of the supported boards in the CompactPCI/PCI system. Since the described boards use only one PCI function (0), this comprises two pieces of information:

- the number of the PCI bus
- the device number on the PCI bus

The current releases (version 2.0 and higher) of the board drivers named above support the following alternatives to specify the location of the carrier board on the PCI bus:

Two alternative descriptor keys to specify the PCI bus number:

- *PCI_BUS_NUMBER*
- *PCI_BUS_PATH* (requires board driver version 2.0 and higher)

Two alternative descriptor keys to specify the PCI device number:

- *PCI_BUS_SLOT*
- *PCI_DEVICE_ID* (requires board driver version 2.0 and higher)

Refer to Chapter B 4.4.2 CompactPCI M-Module Carrier Boards on page 120 for a detailed description of these keys.

PCI_BUS_SLOT

This key specifies the geographical location of the board on a single PCI bus:

- CompactPCI: On CompactPCI systems, slot 1 (marked by a triangle) is always the system slot where the CPU board is plugged. The neighboring slot is slot 2, and so on.
- PCI: On a standard PC motherboard, the slots are either numbered by the manufacturer or it's up to you to define slot 1 and number all slots sequentially.

If the PCI slots belong to different PCI busses, you should number the slots for each PCI bus separately.

Since the allocation of PCI device IDs to the geographical PCI slots depends on the backplane of the PCI system, *PCI_BUS_SLOT* requires additional parameters in the registry under the following PCI key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\MEN\PCI
bus_0
      mechanicalSlot_1: REG_DWORD: 0xc
      mechanicalSlot_n: REG_DWORD: 0x9
bus_1
      mechanicalSlot_1: REG_DWORD: 0x10
      mechanicalSlot_n: REG_DWORD: 0xf
bus_2
      mechanicalSlot_1: REG_DWORD: 0x10
      mechanicalSlot_n: REG_DWORD: 0xf
...
```

For each PCI bus in the system, two parameters must be specified:

- *mechanicalSlot_1* – must be the PCI device ID of slot 1
- *mechanicalSlot_n* – must be the PCI device ID of any other slot (2, 3, etc.)

If *PCI_BUS_SLOT* is specified, the board driver calculates the PCI device ID from the geographic bus-slot number (*PCI_BUS_SLOT*) and the *mechanicalSlot_..* parameters. The board driver uses *mechanicalSlot_n* only to evaluate if the PCI IDs are assigned to the slots in ascending or in descending order.

You will find the right entries for MEN's D1, D2 and F2 CompactPCI systems as imaged above in the *pci.reg* file under *%WORK%/NT/DRIVERS/BBIS/PCI/pci.reg* of your host system. Normally, the backplane of a D1 system is assigned to PCI bus 1, the backplane of a D2 or F2 system is assigned to PCI bus 2.

If you use a PCI M-Module carrier board (e. g. C203, C204) for a standard PCI motherboard you must know the PCI bus number and the PCI device IDs for at least two PCI slots.

The following example illustrates how you can determine the PCI device IDs for PCI slot 1 and PCI slot n using PCIView, a GUI-based utility from BlueWater Systems (see Chapter A 9.1.7 Viewing the PCI Configuration Space on page 78) that displays the PCI Configuration Space of any PCI device in your system.

Assumed hardware:

- Motherboard with four PCI slots (1..4)
- Network adapter plugged on PCI slot 1
- MEN C204 M-Module carrier board plugged on PCI slot 3 (any slot higher than 1)

Do the following:

☑ Start the PCIView utility.

☑ In a combo-box you can select one of the existing PCI devices, then PCIView displays the corresponding PCI Configuration Space of the selected device.

☑ PCIView shows the Vendor Name, Device Name and Device Type in plain language, so that you can identify the plugged network card and the MEN carrier board:

Table A6. *Example of PCIView Utility*

PCI (Bus) Dev/Func	Vendor Name (Vendor ID)	Device Name (Device ID)	Base Class Sub Class	Corresponding Board
(0) 12/0	Realtek (0x10EC)	NE2000 compatible (0x8029)	Network (0x02) Ethernet (0x00)	Network Adapter
(0) 10/0	PLX Technology (0x10B5)	PCI 9050 Target PCI Interface Chip (0x9050)	Bridge (0x06) Other (0x80)	MEN's C204

Now, you have the following information:

- Both boards are located on PCI bus 0.
- The network adapter on slot 1 has the device ID 12 (0x0C).
- The C204 board on slot 3 has the device ID 10 (0x0A).

☑ Now, you can specify the PCI registry entries for this standard PCI motherboard:

```
HKEY_LOCAL_MACHINE\SOFTWARE\MEN\PCI
bus_0
      mechanicalSlot_1: REG_DWORD: 0xc
      mechanicalSlot_n: REG_DWORD: 0xa
```

Note: The PCI registry parameters can be global or can be defined for each individual Windows NT hardware profile:

- Registry path for global allocation:
  HKEY_LOCAL_MACHINE\SOFTWARE\MEN\PCI\
- Registry path for current Windows NT hardware profile:
  HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Hardware Profiles\Current\Software\MEN\PCI\

The board driver first tries to read the parameters from the path for the current hardware profile. If there are no parameters, they are read from the path for global allocation. The parameters specific to a hardware profile have a higher priority than the global parameters.

A 6 W2k Drivers & Device Configuration

This chapter gives you some information about Windows 2000 PnP Drivers and describes how to configure Windows 2000 PnP drivers and their devices. Furthermore, the Windows 2000 PnP driver-specific parameters are explained.

Bus & Function Drivers

We will not describe the architecture of Windows drivers here, but for a better understanding you should know what type of Windows 2000 PnP drivers are provided:

- MEN Device Drivers (e.g. for M-Modules) are function drivers. All devices created by these drivers belong to the device setup class "MDIS devices".
- MEN Board Drivers (e.g. for carrier boards) are bus drivers. All devices created by these drivers belong to the device setup class "BBIS boards".

As noted in Chapter A 4.4.1 W2k Driver & PnP Basics on page 25, a bus driver is a driver that enumerates all devices residing on a particular bus/interface. For example: Windows 2000/XP comes with the PCI bus driver that scans all PCI buses for newly connected PCI devices during the Windows boot phase. If a new PCI device was found, Windows tries to install a matching function driver for the PCI device. A function driver creates a device instance (the sw-device) for each hw-device that is accessible from your application via the device name. The function driver gets the necessary resources (I/O, memory, interrupt) for their operation from the corresponding bus driver.

For further differences between bus and function drivers, refer to Chapter A 1.5 How MDIS4 Maps into the Windows NT Architecture on page 10.

Configuration Data

The driver and device configuration data comprise:

- Windows 2000 PnP driver-specific parameters, described in Chapter A 6.1 W2k Device Parameters on page 53.
- A subset of the MDIS4 standard descriptor parameters, described in Chapter B 4 MDIS Device Descriptors on page 112.
- Low-level driver specific descriptor parameters, described in the driver's user manual.

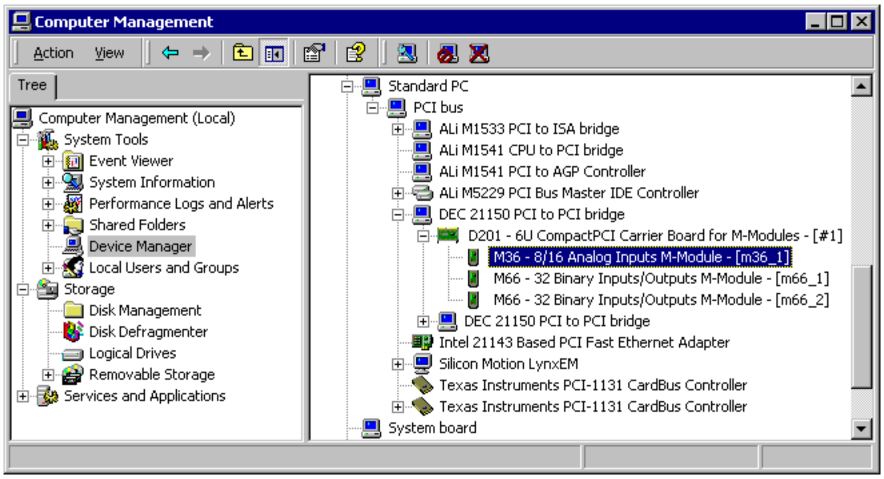
The device configuration can be modified after the driver installation via a device property page, accessible over the Windows *Device Manager.*

Note: It is also possible to change the default device and driver configuration commonly for a hw-device type (not an individual hw-device) in the *.inf* file belonging to the hw-device type. This is not the usual way and requires an extensive knowledge about *.inf* files. MEN does not support this possibility but you can do this on your own risk. Refer to the MSDN Library for further information.

A 6.1 W2k Device Parameters

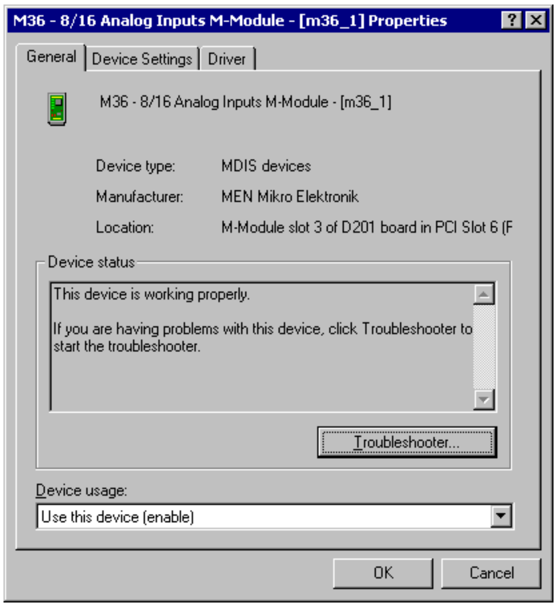
The Windows 2000 PnP driver-specific parameters can be accessed via the *Device Property* page, accessible over the Windows *Device Manager*.

Use the view option *Devices by connection* to easily recognize which device (e.g. M-Module) resides on which board (e.g. carrier board):



```
Standard PC
  PCI bus
    ALi M1533 PCI to ISA bridge
    ALi M1541 CPU to PCI bridge
    ALi M1541 PCI to AGP Controller
    ALi M5229 PCI Bus Master IDE Controller
    DEC 21150 PCI to PCI bridge
      D201 - 6U CompactPCI Carrier Board for M-Modules - [#1]
        M36 - 8/16 Analog Inputs M-Module - [m36_1]
        M66 - 32 Binary Inputs/Outputs M-Module - [m66_1]
        M66 - 32 Binary Inputs/Outputs M-Module - [m66_2]
      DEC 21150 PCI to PCI bridge
    Intel 21143 Based PCI Fast Ethernet Adapter
    Silicon Motion LynxEM
    Texas Instruments PCI-1131 CardBus Controller
    Texas Instruments PCI-1131 CardBus Controller
  System board
```

Double-click on a desired device (e.g. an M36 M-Module plugged on a D201 PCI carrier board) to open the corresponding *Device Property* page:



M36 - 8/16 Analog Inputs M-Module - [m36_1]

Device type: MDIS devices
Manufacturer: MEN Mikro Elektronik
Location: M-Module slot 3 of D201 board in PCI Slot 6 (F

Device status

This device is working properly.

If you are having problems with this device, click Troubleshooter to start the troubleshooter.

Device usage:
Use this device (enable)

The *Device Property* page of MEN's Windows 2000 PnP drivers comprises the following tabs:

- General
- Device Settings
- Driver
- Resources (optional)

The *General*, *Driver* and *Resources* tabs are the usual standardized Windows tabs. Refer to the Windows documentation for further details.

Device Location

The *General* tab provides, among others, the location information of the device. Click on the *Location* information and then scroll right to view the entire location description:

```
M-Module slot 3 of D201 board in PCI Slot 6 (PCI bus 2, device 15,
function 0)
```

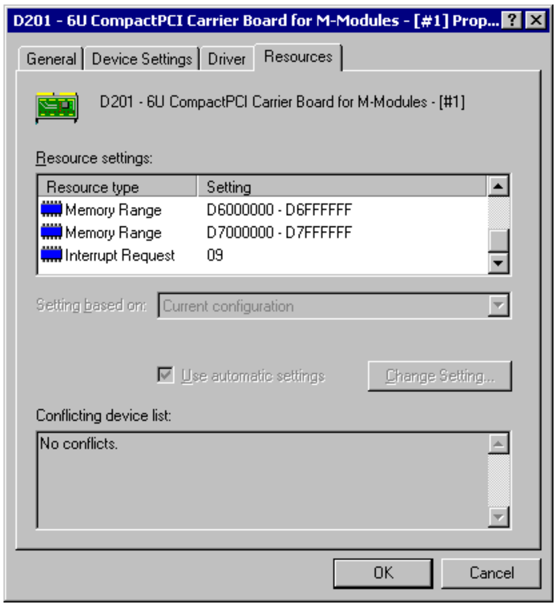
In this example, you can see that the M36 M-Module is plugged in the M-Module slot #3 on a D201 M-Module carrier board which resides on PCI bus 0, device 15, function 0.

For PCI devices, the given PCI slot number (6 in this example) in the location information is normally not usable for CompactPCI systems. For a PCI device residing on a PC motherboard with a properly implemented BIOS it may be an important information on whose physical PCI slot a device is plugged. However, use the exact and always correct PCI bus/device/function information to determine the location of a PCI device.

Resources Information

Although basically all MDIS devices use resources (I/O, memory, interrupt), many *Device Property* pages (e.g. for M-Modules) contain no resource tab. Unfortunately only the MS bus drivers are able to split bus resources between several residing devices. Actually, the *Resources* tab displays all the resources currently assigned to the selected device (not the used resources).

However, you can use the *Resources* tab of the device's board instance (e.g. M-Module carrier board) where the device resides to view all resources assigned to the board and all the belonging devices together:


D201 - 6U CompactPCI Carrier Board for M-Modules - [#1] Prop...

D201 - 6U CompactPCI Carrier Board for M-Modules - [#1]

Resource settings:

Resource type	Setting
Memory Range	D6000000 - D6FFFFFF
Memory Range	D7000000 - D7FFFFFF
Interrupt Request	09

Setting based on: Current configuration

[✓] Use automatic settings

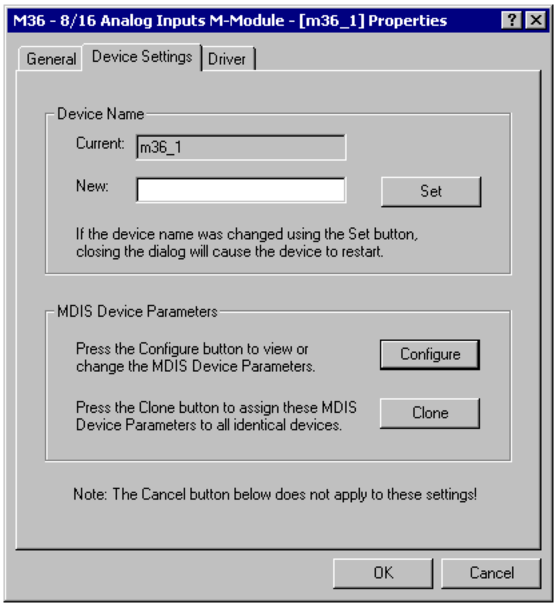
Conflicting device list:

No conflicts.

Device Settings

The *Device Settings* tab designed by MEN depends on the driver type (function/bus driver). The main difference is that all *Device Settings* tabs of a function driver related device have a *Device Name* setting section whereas the *Device Settings* tabs of a bus driver do not.

Device Settings tab of a function driver (e.g. M36 M-Module):



M36 - 8/16 Analog Inputs M-Module - [m36_1] Properties

General | Device Settings | Driver

Device Name

Current: m36_1

New:

Set

If the device name was changed using the Set button, closing the dialog will cause the device to restart.

MDIS Device Parameters

Press the Configure button to view or change the MDIS Device Parameters.

Configure

Press the Clone button to assign these MDIS Device Parameters to all identical devices.

Clone

Note: The Cancel button below does not apply to these settings!

OK | Cancel

Modifying the Device Name

The default device name will be set during the sw-device installation to <dev-model>_1..n (e.g. to *m36_1* for the first M36 M-Module, *m36_2* for the second).

Note: After you have changed the device name, the system may prompt you to restart the computer. This is not necessary.

Windows XP: The new device name will not be shown in the device tree until the *Device Manager* is re-opened. However, the current device name will always be displayed in the "*Current*" text box on the *Device Settings* tab.

The device name can be changed to an arbitrary name via the *Device Settings* tab with the following restrictions:

- The length of the device name is limited to 25 characters.
- The name must not contain any blanks or non-printable characters.
- The name must be unique, i.e. no other MDIS sw-device must have this name.

Cloning MDIS Device Parameters

The *Clone* button of the *MDIS Device Parameters* section allows you to assign all MDIS device parameters of the current device to all installed device of the same device type.

Example: If you have three M36 M-Module devices installed and you need the same *MDIS Device Parameters* configuration for all three devices, just configure the *MDIS Device Parameters* of only one M36 device and then press the *Clone* button of this device. After this, all three M36 devices have the same MDIS device parameters.

The next chapter describes how to configure the MDIS device parameters.

A 6.2 MDIS4 Device Parameters

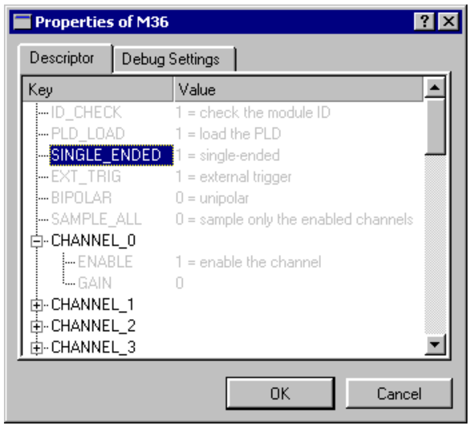
The MDIS4 device parameters comprise a subset of the MDIS4 standard descriptor parameters and the low-level driver specific descriptor parameters.

Viewing and Modifiying MDIS4 Device Parameters

To view or modify MDIS4 device parameters: In the *Device Manager* open the *Device Property* page of the desired device, select the *Device Settings* tab and then press the *Configure* button. The *Properties of xxx* window appears, which you may already know if you have worked with one of MEN's other MDIS4 System Packages (e.g. for Linux, VxWorks, ...) and which uses the same *Properties of xxx* window (within the MDIS Wizard) for configuration tasks on the host before the target installation.

Note: The *Properties of xxx* window is written using Trolltech's QT, a multiplatform, C++ application framework that lets developers write one application that will run on several platforms.

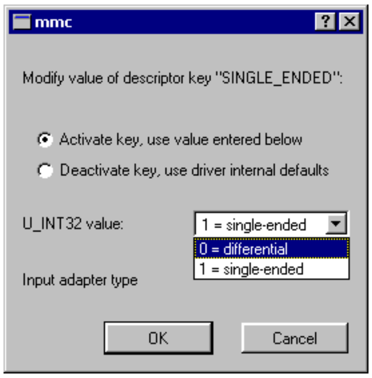
Descriptor Tab



Key	Value
ID_CHECK	1 = check the module ID
PLD_LOAD	1 = load the PLD
SINGLE_ENDED	1 = single-ended
EXT_TRIG	1 = external trigger
BIPOLAR	0 = unipolar
SAMPLE_ALL	0 = sample only the enabled channels
CHANNEL_0	
ENABLE	1 = enable the channel
GAIN	0
CHANNEL_1	
CHANNEL_2	
CHANNEL_3	

The *Properties of xxx* window comprises two tabs: *Descriptor* and *Debug Settings*. The *Descriptor* tab displays in an explorer-like view the MDIS4 device parameters and their current values with a short description.

Double-click on the desired parameter to modify its value. A further, parameter-type dependent window appears that gives you the option to use the default value or to specify another, parameter-specific value.

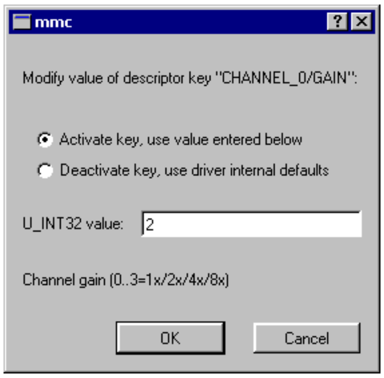


Modify value of descriptor key "SINGLE_ENDED":

(•) Activate key, use value entered below
( ) Deactivate key, use driver internal defaults

U_INT32 value: 1 = single-ended
0 = differential
1 = single-ended

Input adapter type



Modify value of descriptor key "CHANNEL_0/GAIN":

(•) Activate key, use value entered below
( ) Deactivate key, use driver internal defaults

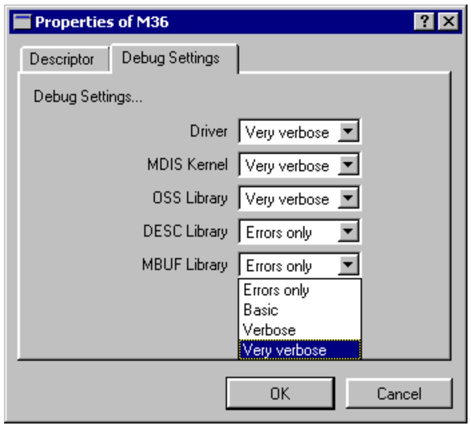
U_INT32 value: 2

Channel gain (0..3=1x/2x/4x/8x)

Changing a parameter value is easily provided with parameter appropriate controls, e.g. a drop-down list box that contains fixed predefined values (left figure above) or a common edit box (right figure above) where you can enter your desired value. Furthermore, the window displays the parameter name, type, description, value explanation and supports value range checking if applicable. Press the *OK* button to assign the specified value or the *Cancel* button to abort.

Debug Settings Tab

The *Debug Settings* tab lists the set debug level of each software module used by the driver belonging to the device.



Driver: Very verbose
MDIS Kernel: Very verbose
OSS Library: Very verbose
DESC Library: Errors only
MBUF Library: Errors only

Use the drop-down list box to change the debug level. The four provided debug levels set the following hex values for the debug descriptor parameters:

Table A7. *Debug Levels for Debug Descriptor Parameters*

Debug Level	Hex value	Enabled Debug Flags
ErrorsOnly	`0xC0008000`	*INTR, NORM, ERROR*
Basic	`0xC0008001`	*INTR, NORM, ERROR, LEV1*
Verbose	`0xC0008003`	*INTR, NORM, ERROR, LEV1, LEV2*
VeryVerbose	`0xC0008007`	*INTR, NORM, ERROR, LEV1, LEV2, LEV3*

Refer to Chapter B 4.5 Driver Debugging on page 124 for further information.

The *Debug Settings* apply only if you are using the checked version of a driver. See Chapter B 4 MDIS Device Descriptors on page 112.

A 7 Building MDIS4 Applications from C Sources

This chapter describes how to build executables from the provided C source code of example programs and tools, shipped with the driver packages. This is useful if you want to use the MEN sources as the starting point for your own C application development.

You will find the ready-to-use built driver example programs and tools in the *%WORK%\NT\OBJ\EXE\MEN\I386\FREE* path of your host system. For test purposes, just copy the executables to an arbitrary path (e. g. *C:\MEN*) of the target system.

No special executables are required for the Windows 2000 PnP drivers, therefore the executables provided under *%WORK%\NT\OBJ\EXE\...* can be used for Windows NT 4.0 as well as Windows 2000 PnP drivers. However, you have to install *men_winspec.dll*, which makes executables independent of the driver type. See Chapter A 8 Writing Applications for MDIS on page 64.

Note: Building Windows drivers and API libraries from the shipped C source code is no longer supported because all driver packages for Windows come with ready-to-use built object code (NT4 drivers, W2000 drivers, API libraries).
To build Windows drivers from MDIS4 low-level drivers, you would require the WinDk library from BSQUARE, a C++ class library used by MEN drivers. Unfortunately, BSQUARE has discontinued WinDk and it is no longer available. However, if you require an MDIS4 Windows driver for your own MDIS4 low-level driver, please contact MEN's support: support@men.de.

Required Compiler

To build MDIS4 example programs or tools (*.exe*) from MEN's C sources, you need Microsoft Visual C++ Professional (4.1 or higher).

You can choose between two build alternatives:

- VC NMAKE — is easy to use and will speed up your work.
- VC IDE — gives you the best overview of your project.

MDIS Example Application

The *%WORK%\NT\TOOLS\MDISAPP* folder contains a simple example MDIS application written in C, called *MDISAPP*, which uses the MDIS-API, USR-OSS and USR-UTL libraries. The application contains the necessary make and project files to build the program using *NMAKE* or from the VC IDE. MDISAPP can also be used as a template for your own MDIS applications.

The following sections describe the two different build processes.

A 7.1 Using NMAKE

NMAKE ships with the Microsoft VC++ Compiler and must be invoked from the command line.

NMAKE uses the *..\NMAKE\makefile* which includes the operating system-independent meta make file *program*.mak*.

For further information about *NMAKE* refer to the Visual C++ documentation.

To build an MDIS4 application using *NMAKE*, proceed as follows:

☑ Open a DOS box.

☑ Change the path to the folder where the *NMAKE\makefile* file of the application resides (e. g. *%WORK%\NT\TOOLS\MDISAPP\NMAKE*).

☑ Invoke *NMAKE*.

```
%WORK%\NT\TOOLS\MDISAPP\NMAKE>nmake [cfg=<configuration>]
```

By default, *NMAKE* links the application with the free builds of the static MDIS API libraries. Optionally, one of four configurations can be specified:

Table A8. *Possible Configurations to build an MDIS4 Application using* NMAKE

cfg=	Meaning
LIB_Release (default)	Links with MDIS API libraries – release build
LIB_Debug	Links with MDIS API libraries – debug build
DLL_Release	Uses MDIS API DLLs – release build
DLL_Debug	Uses MDIS API DLLs – debug build

Depending on the chosen configuration, the free or checked version of the application will be built. The built executable file (e. g. *mdisapp.exe*) will be stored in a subfolder, which is named after the configuration (*LIB_Release*, *LIB_Debug*, *DLL_Release*, *DLL_Debug*).

Note: You can use *nmake [cfg=<configuration>] clean* to perform a clean make for the specified configuration.

A 7.2 Using VC++ *IDE*

Visual C++ uses project files (*.dsp*) to store configuration data such as source files, input libraries, compiler and linker switches for software modules. The *MDISAPP* example application contains the *mdisapp.dsp* file which can be used as a project file template for your own MDIS4 applications.

The basic configuration files for VC++ are the so-called workspace files (*.dsw*). A workspace file contains links to one or more project files. If you try to open a project file directly from VC++ and there is no associated workspace file, VC++ will create a workspace file that contains a link to the project file.

For a detailed description of project files and workspace files see the Visual C++ documentation.

A 7.2.1 Building an Application

To build an MDIS4 application, proceed as follows:

☑ Start VC++ and open the applications workspace file, e. g. *myappl.dsw* (or project file *myappl.dsp*), located in the applications VC subfolder (e.g. *..\TOOLS\MDISAPP\VC*).

☑ Choose the desired configuration: *Build ➢ Set Active Configuration...*

Table A9. *Possible Configurations to build an MDIS4 Application using VC++* IDE

Configuration	Meaning
Win32 MDIS-LIB Release	Links with MDIS API libraries – release build
Win32 MDIS-LIB Debug	Links with MDIS API libraries – debug build
Win32 MDIS-DLL Release	Uses MDIS API DLLs – release build
Win32 MDIS-DLL Debug	Uses MDIS API DLLs – debug build

☑ Build the application: *Build ➢ Rebuild All*.
The application executable file, e. g. *myappl.exe*, will be placed under a sub-folder of the application's *VC* subfolder. The created subfolder will be named after the chosen configuration (*LIB_Release, LIB_Debug, DLL_Release, DLL_Debug*).

A 7.2.2 Cloning a Project File

To create a project file for your application, proceed as follows:

☑ Create a *VC* subfolder in your application folder (i. e. the folder where the sources of your application reside).

☑ Copy the *mdisapp.dsp* project file from *%WORK%\NT\TOOLS\MDISAPP\VC* to your application *VC* path and rename it to your application's name, e. g. *myappl.dsp.*

☑ Open the *myappl.dsp* file using an ASCII editor and replace all *mdisapp* strings by the name of your application, e. g. *myappl*. Use the "Find and Replace" mechanism of your editor with the "case sensitive" option to do this. Save your *myappl.dsp* file.

☑ Start VC++ and use the *File* ➤ *Open Workspace...* dialog to open the *myappl.dsp* project file. VC++ will create a workspace file (e. g. *myappl.dsw*) for your project under your application *VC* path and name it just as the project file.

☑ Use the FileView of the Workspace window to remove the unused (and not available) source files. Add your own source files here (context menu ➤ *Add Files to Project*).

☑ If your application requires an additional driver-specific MDIS API library: Open the *Project Settings* dialog, select the *Link* tab (*Category*: *General*) and add the necessary MDIS API library to the Object/library modules line.

☑ Save the project file (*File* ➤ *Save All*).

A 8 Writing Applications for MDIS

In general, you can access MDIS4 drivers from any programming language that uses DLLs, because the drivers can be accessed via the MDIS API libraries that are available in DLL versions. The main programming language for applications supported by MDIS4 for Windows is C/C++. However, our customers successfully use a lot of other programming languages to write MDIS4 applications, too.

The *men_winspec.dll*

The Windows 2000 PnP driver enhancement of MDIS4 for Windows makes it necessary to install the *men_winspec.dll* file used by the MDIS-API library (the static and the DLL version) on the target. This is because Windows 2000 PnP drivers use interface classes to access a device whereas NT4 drivers use system-wide device names.

men_winspec.dll was developed to avoid that the MDIS-API library and therefore all MDIS applications have to be built specifically for each Windows driver type (NT4/W2K). It encapsulates the device name handling in two different *men_winspec.dll* versions. One for all targets running Windows NT 4.0 and one for all targets running Windows 2000/XP, regardless of the used Windows driver type (NT4/W2K).

The *men_winspec.dll* is required if an application program is linked with the static or using the dynamic (DLL) MDIS-API library version 3.0 or later. The target OS (NT or W2k/XP) specific *men_winspec.dll* version must be installed manually during your MDIS application installation.

Target running Windows NT 4.0

☑ Copy *men_winspec.dll* from the *%WORK%\NT\OBJ\DLL\MEN\I386\FREE* path of your host system to the *%SystemRoot%\System32* system path of the target system.

Target running Windows 2000/XP

☑ Copy *men_winspec.dll* from the *%WORK%\W2K\OBJ\DLL\MEN\FRE\I386* path of your host system to the *%SystemRoot%\System32* system path of the target system.

Switches for MEN Header Files

The MEN header files (e.g. *men_typs.h*) evaluate some switches (defines) (e.g. *WINNT*) that must be set before the files are included from the application's sources. For application development using the MS Visual C++ C/C++ compiler, the required defines are set by default or in the provided *NMAKE* makefiles and Visual C++ project files.

If you intend to use the MEN header files in your own (non Visual C++) development environment you have to make sure that the required defines are set. Examine the provided *NMAKE* makefiles or Visual C++ project files to see which defines are required.

A 8.1 Basics of MDIS API Libraries

- All MDIS API libraries (the static libraries as well as the DLL versions) use the *__stdcall* calling convention.
- All MDIS API libraries and MDIS API DLLs use the multithreaded versions (*LIBCMT.LIB, MSVCRT.DLL*) of the C runtime library and therefore they are safe for multithreading.
- All MDIS API DLLs export their functions according to the *__stdcall* naming convention (e. g. *_UOS_SigInit@4*) and the PASCAL naming convention (e. g. *UOS_SIG_INIT*). For PASCAL upper-case spelling, *XxxYyy* are replaced by *XXX_YYY.* Chapter A 9.1.6 Examining Dependencies of Executables on page 78 describes how you can get the names of all exported functions.
- MDIS API library functions (e. g. *UOS_SigInit* of the USR-OSS library) can use call-back functions that are called by different threads that were created by the library functions themselves. Therefore the used programming language must support multithreading (the free threading model) for these MDIS API library functions.
- If your applications will use the DLL versions of the MDIS API libraries (*men_mdis_api.dll*, *men_usr_oss.dll*, *men_usr_utl.dll*, etc.) you must copy the required DLLs from the *%WORK%\NT\OBJ\DLL\MEN\I386\FREE* path of your host system to the *%SystemRoot%\System32* system path of the target system or to the path where your application executable resides.

The following sections give you some hints on writing MDIS4 applications for C/C++ (especially Visual C++), Visual Basic, Delphi and National's Measurement Studio.

A 8.2 C/C++ Specifics

There are two possibilities for a C/C++ application to call the MDIS API library functions:

- Via the MDIS API libraries (**.lib*), linked statically to the application.
- Via the MDIS API DLLs (*men_*.dll*), used at runtime.

If you intend to use the static MDIS-API library (*mdis_api.lib*), you have to link your C/C++ application with the *men_winspec.lib* import library (located under *%WORK%\NT\OBJ\DLL\MEN\I386\FREE*) that belongs to the *men_winspec.dll* (NT4 and W2K version). See Chapter The men_winspec.dll on page 64 for further information about *men_winspec.lib*.

A 8.2.1 Using Static MDIS API Libraries

You must link your C/C++ application with the static MDIS API libraries that you want to use (at least *mdis_api.lib*).

The static MDIS API libraries are located under *%WORK%\NT\OBJ\LIB\MEN\I386\FREE(CHECKED)*.

A 8.2.1.1 C Runtime Library (CRT) – Multithreading

Since the libraries use functions of the CRT and the static MDIS API libraries **must** be safe for multithreading, it is necessary that you use one of the multithreaded CRTs (*LIBCMT.LIB* or *MSVCRT.DLL*) to build your application.

Do not use the non-multithreaded CRT *LIBC.LIB*, which is linked by default by Visual C++.

A 8.2.1.2 Calling Convention

Since the static MDIS API libraries were built using the *__stdcall* calling convention, a program that will be linked with these libraries must call all MDIS API functions also using the *__stdcall* calling convention.

All MDIS API declarations in header files newer than 07/17/00 (e. g. *mdis_api.h*, *usr_oss.h*, *usr_utl.h*) specify the *__stdcall* calling convention explicitly if define *WINNT* is set as a compiler option for the source files which include the MDIS API header files. In this case, the compiler's common calling convention does not matter. However, for a few older MDIS API header files, it is necessary to use the *__stdcall* calling convention as a common compiler option.

A 8.2.2 Using MDIS API DLLs

For each *men_*.dll*, there is a corresponding *men_*.lib* import library, which can be used for C/C++ programs. Don't confuse the *men_*.lib* import libraries for the **.lib* static libraries.

The import libraries are located under *%WORK%\NT\OBJ\DLL\MEN\I386\FREE(CHECKED)*.

You can link your C/C++ application with the import libraries (e. g. *men_mdis_api.lib*), which belongs to the MDIS API DLLs (e. g. *men_mdis_api.dll*) you want to use at runtime.

A 8.2.2.3 C Runtime Library (CRT) – Multithreading

Since the libraries use functions of the CRT and **must** be designed for multithreaded operation, it is necessary that you use the multithreaded CRT *MSVCRT.DLL* to link your application, because the MDIS API DLLs also use this CRT.

Do not use *LIBC.LIB* or *LIBCMT.LIB*.

A 8.2.2.4 Calling Convention

Since the MDIS API DLLs were built using the *__stdcall* calling convention, a program that wants to use the functions exported from the DLLs must call all MDIS API functions also using the *__stdcall* calling convention.

A 8.2.3 Visual C++ Notes

- The default calling convention in Visual C++ is *__cdecl*.
- If the proper calling convention is not set (see Chapter A 8.2.1.2 Calling Convention on page 66) or the library which contains the appropriate function was not found (see Chapter A 8.2.1.1 C Runtime Library (CRT) – Multithreading on page 66), Visual C++ stops with the following error message:

```
xxx.obj : error LNK2001: unresolved external symbol _YYY
```

- To change the calling convention select *Project* ➢ *Settings* ➢ *C/C++ Tab*, choose category *Code Generation* and select the desired calling convention (e. g. *__stdcall*).
- To choose the CRT select *Project* ➢ *Settings* ➢ *C/C++ Tab*, choose category *Code Generation* and select the desired runtime library (e. g. *Multithreaded DLL*).

A 8.3 Visual Basic Specifics

VB applications must use the MDIS API-DLLs (*men_*.dll*) to access MDIS4 drivers.

A 8.3.1 VB Declaration Files

In general, MEN only provides include files (*.h*) for C/C++ programs, which contain prototypes of functions and other defines. However, the MDIS4 for Windows System Package contains VB declaration files (*.bas*) for the following MDIS4 software modules:

Table A10. *VB Declaration Files included in MDIS4 Windows NT System Package*

VB Declaration File	Corresponding Software Module
mdis_api.bas	MDIS-API DLL (*men_mdis_api.dll*)
usr_oss.bas	USR-OSS DLL (*men_usr_oss.dll*)
usr_utl.bas	USR-UTL DLL (*men_usr_utl.dll*)
mt_drv.bas	MT Test Device Driver (*men_mt.sys*)

Note: A few functions of the MDIS API libraries cannot be used under VB. Refer to the corresponding *.bas* files for further details.

All VB declaration files are located in folder *%WORK%\NT\INCLUDE\NATIVE\MEN*.

If you need other declarations, e. g. for a special MDIS4 device driver, it is up to you to write the needed VB declaration file. The supplied *xxx.h* and corresponding *xxx.bas* files give you some guidance on how to convert a C/C++ include file into a VB declaration file.

On the web (www.programmersheaven.com), we have found a public domain program called *VB Declaration Converter* that reads WIN32 SDK header files and converts C prototypes to VB declarations. Although it does not work well with our include files, it may help you to build your VB declaration files.

A 8.3.2 Multithreading

Currently, VB does not support the free threading model. Therefore it is not safe to use MDIS API functions that use a different thread to call functions which reside in your VB application (callback functions). Currently, only the *UOS_SigInit* function of the USR-OSS library uses these techniques.

Note on Function *UOS_SigInit*

Instead of installing a signal handler (callback function) using the *UOS_SigInit* function, use the *UOS_SigWait* function to wait for signals. However, you can write your own signal handler thread that uses *UOS_SigWait* to notify your application about signals.

A 8.3.3 *MAPIVB* – VB Example MDIS4 Application

MAPIVB is a VB test and example program for MDIS4. It demonstrates the usage of MDIS API DLLs and MDIS drivers with Visual Basic. *MAPIVB* comes with sources to give you an extensive example of how MDIS drivers can be accessed from VB applications. *MAPIVB* is located under *%WORK%\NT\VB\MAPIVB*. There is also a *redme.txt* file with further details.

A 8.4 Delphi Specifics

Delphi applications must use the MDIS API DLLs (*men_*.dll*) to access MDIS4 drivers.

A 8.4.1 Delphi Import Units

In general, MEN only provides include files (*.h*) for C/C++ programs which contain prototypes of functions and other defines. You can use these to create Delphi Import Units (*.pas*) necessary for Delphi.

To convert our include files into Delphi Import Units, we recommend to use Dr.Bob's DLL Header Converter Program, *HeadConv v4.00*, a command-line utility. *HeadConv* is freeware (Delphi Jedi Project) and can be found on the web together with a user guide ("Using C DLLs with Delphi (and HeadConv)") at www.drbob42.com/tools/index.htm.

A 8.5 Measurement Studio

A 8.5.1 General

MDIS4 System Package for Windows NT/2000/XP/Embedded (article no. 13M000-06) is required, and the recommended device drivers for the board (e.g. M-Module carrier) and device (e.g. M-Module) must be installed in your system.

A 8.5.2 Customizing Your Project

A 8.5.2.5 MDIS4 API Libraries

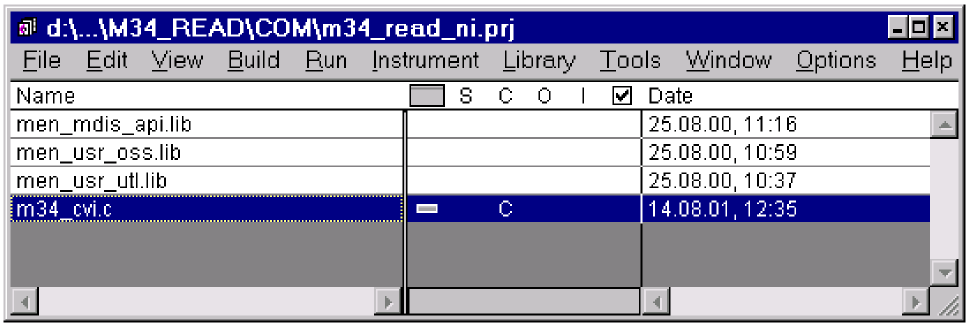
Don't use MDIS4 static libraries, use the MDIS4 dynamic link libraries (DLLs) only.

For each dynamic linc library *men_*.dll*, there is a corresponding *men_*.lib* import library which can be used. Don't confuse the *men_*.lib* import libraries with the **.lib* static libraries.

You can link your application with the import libraries (e.g. *men_mdis_api.lib*), which belong to the MDIS API DLLs (e.g. *men_mdis_api.dll*) that you want to use at runtime.

The following figure shows an example of a project configured to use the MDIS4 DLLs.

Figure A2. *Editing the Project*

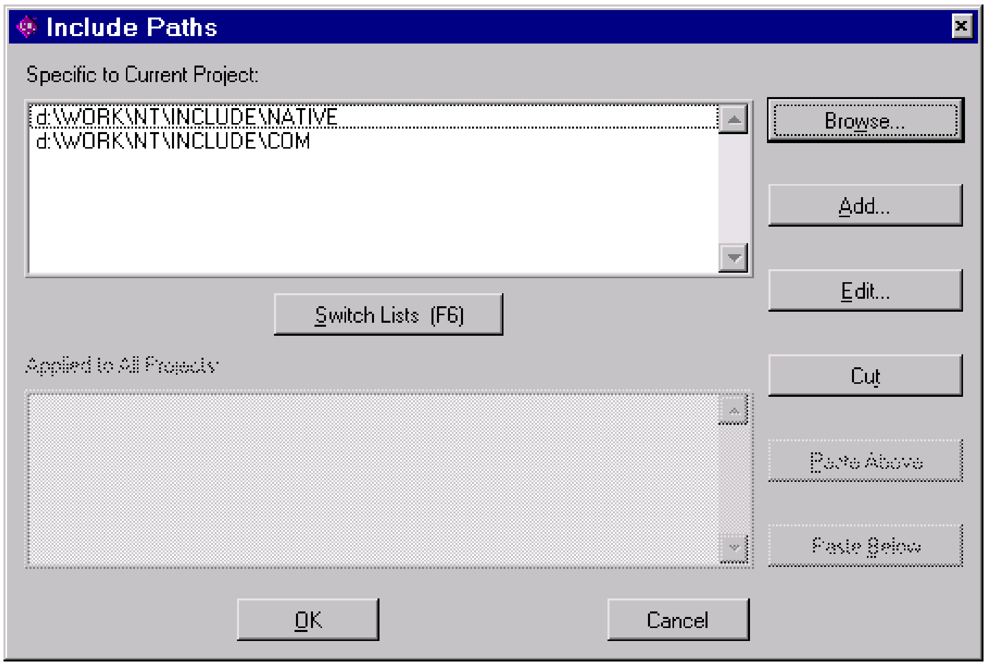

d:\...\M34_READ\COM\m34_read_ni.prj
File Edit View Build Run Instrument Library Tools Window Options Help

Name	S C O I ✓	Date
men_mdis_api.lib		25.08.00, 11:16
men_usr_oss.lib		25.08.00, 10:59
men_usr_utl.lib		25.08.00, 10:37
m34_cvi.c	C	14.08.01, 12:35

A 8.5.2.6 Include Paths for MDIS4 Header Files

Set the include paths to MEN's common and native header files:
(e.g. *D:\WORK \NT\INCLUDE\COM* and *D:\WORK \NT\INCLUDE\NATIVE*)

Figure A3. *Include Paths*

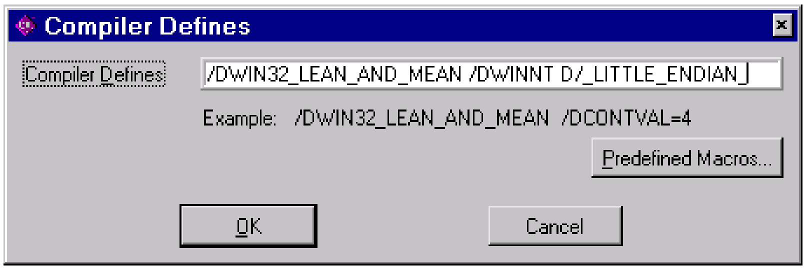


d:\WORK\NT\INCLUDE\NATIVE
d:\WORK\NT\INCLUDE\COM

A 8.5.2.7 Defines

Add defines *WINNT* and *_LITTLE_ENDIAN_*.

Figure A4. *Setting the Defines*



Compiler Defines: /DWIN32_LEAN_AND_MEAN /DWINNT D/_LITTLE_ENDIAN_

Example: /DWIN32_LEAN_AND_MEAN /DCONTVAL=4

A 8.5.3 Writing Code with MDIS4

A 8.5.3.8 Starting Code Generation

After customizing the user interface editor window (**.uir* file) and generating the code frame, you must add the MDIS4 code for accessing the device hardware.

A 8.5.3.9 Main Function

In the main function of your source code you open the path to the device and make the device's (e.g. M-Module's) default configurations.

Note: Additional MDIS code is printed in bold type: `/* MDIS code */`
path is a global variable.

```
int main (int argc, char *argv[])
{
    if (InitCVIRTE (0, argv, 0) == 0)
        return -1;    /* out of memory */
    if ((panelHandle = LoadPanel (0, "m68.uir", PANEL)) < 0)
        return -1;
    /*--------------------+
    |  open path          |
    +--------------------*/
    /* open the device */
    if ((path = M_open("/m68_1")) < 0) {
         /* error handling */
         ...
    }
    /*--------------------+
    |  config             |
    +--------------------*/
    /* set current channel */
    if ((M_setstat(path, M_MK_CH_CURRENT, chan)) < 0) {
         /* error handling */
         ...
    }
    ...
    DisplayPanel (panelHandle);
    RunUserInterface ();
    DiscardPanel (panelHandle);
    if (M_close(path) < 0)  {
         /* error handling */
         ...
    }
    return 0;
}
```

A 8.5.3.10Callback Functions

In a callback function the application can do additional hardware access, depending on what is recommended to the event.

Note: Additional MDIS code is printed in bold type: **`/* MDIS code */`**
path is a global variable.

```
int CVICALLBACK refM (int panel, int control, int event,
        void *callbackData, int eventData1, int eventData2)
{
    switch (event)
        {
        case EVENT_xxx:
            /* Additional MDIS code to communicate with the M-Module */
            /* (e. g. M_read, M_write, M_getstat, ...) */
           ...
            break;
        }
    return 0;
}
```

A 8.6 *MDISNT* Test and Configuration Utility

The *MDISNT* program is a test and configuration utility for Windows that was original written by MEN for internal usage in the MDIS development and test process. Therefore, some *MDISNT* commands deal with MDIS internals, which are not fully documented. However, MEN supplies *MDISNT* because it is also useful for your own application development and test purposes.

You can use the utility to quickly test any driver function, determine the capability of a driver, get revision information of all software modules used by the driver, and much more.

MDISNT highlights:

- Test all MDIS-API library functions (*M_xxx*)
- Test the signal functions (*UOS_SigXxx*) of the USR-OSS library
- Test some MK, low-level driver and BBIS functions
- Multithread support (up to 10 threads can be used)
- Up to 10 device paths can be opened from each thread created
- Block buffer with byte, word or dword alignment used by *M_getblock*, *M_setblock* and block Get/SetStat calls.

To benefit your development process, *MDISNT* ships with C sources (*%WORK%\NT\TOOLS\MDISNT*). However, you cannot build *MDISNT* from the delivered sources on your own, since some required MDIS-internal header files are not included.

A 8.6.1 Using *MDISNT*

The MDISNT executables are located in the executable path (*%WORK%\NT\OBJ\EXE\MEN\I386\FREE*) of your host system.

There are two different versions of the tool:

- *mdisnt.exe* (linked statically with the MDIS API libraries)
- *mdisntdll.exe* (uses the DLL versions of the MDIS API libraries)

Copy the executables to an arbitrary path of the target system (e. g. *C:\MEN\*).

Note: If you want to use *mdisntdll.exe* the following DLLs must be present in the *\WINNT\system32* folder of your target system: *men_mdis_api.dll*, *men_usr_oss.dll*, *men_usr_utl.dll*. However, the *men_winspec.dll* is always required (for *mdisnt.exe* and *mdisntdll.exe*).

If you run the *MDISNT* tool (*mdisnt.exe* or *mdisntdll.exe*) from the command prompt, you will see an overview of the available commands. This command table can always be displayed via the *-h* (HELP) command.

MDISNT Output on Start-up

```
 mdisnt : Test and Configuration Utility for MDIS/NT

 (c) 1999 by MEN mikro elektronik GmbH, V 1.3 2000/11/07
==========================================================

Commands
========
        -h      : HELP                  -e      : EXIT thread/proc
        -tc     : Create thread         -ts     : Suspend thread
        -ps     : Select path
_____MDIS-API_____
        -mo     : M_open                -mc     : M_close
        -mg     : M_getstat blk:(*)     -ms     : M_setstat blk:(*)
        -mr     : M_read                -mw     : M_write
        -mgb    : M_getblock (*)        -msb    : M_setblock (*)
        -merr   : M_errstring           (*) uses blk-buffer
_____USR_OSS_____
        -si     : UOS_SigInit           -se     : UOS_SigExit
        -sin    : UOS_SigInstall        -sre    : UOS_SigRemove
        -sm     : UOS_SigMask           -sum    : UOS_SigUnMask
        -sw     : UOS_SigWait           -uerr   : UOS_ErrString
_____MK_____
        -rev    : get rev strings
        -addr   : get address info      -irq    : get irq info
        -dev    : get device info       -brd    : get board info
        -path   : get path info         -misc   : get misc info
        -dsemw  : wait for dev-sem      -dsemr  : release dev-sem
        -iod    : IOCTL for delay
_____LL-DRV_____
        -lldrv  : ll-drv info           -devid  : device id-prom
_____BBIS_____
        -bbis   : bbis info             -brdid  : board id-prom
_____SERVICES_____
        -dbg    : set/get debug level
        -bbalgn : align blk-buf         -bbshow : show blk-buffer
        -bbset  : set blk-buffer        -bbfill : fill blk-buffer
_____NT_____
        -pid    : get process id        -sev    : set event


Thr:1/1 (MAIN tid:0x5d) / Dev:<none> (path:-) ==> cmd: -
```

A 9 Solving Problems

This chapter gives you some information on how to solve problems with MDIS4 software modules. The first part shows you how to gather important information for error debugging. The second part lists typical problems and the proposed solutions.

A 9.1 Gathering Information

Under Windows 2000/XP, the *Computer Management* is the central console with a collection of administrative tools. To open *Computer Management*:

☑ Right-click on the *My Computer* desktop icon and select *Manage.*

Or

☑ W2k:
Go to *Start* ➢ *Settings* ➢ *Control Panel* ➢ *Administrative Tools* ➢ *Computer Management.*

☑ XP:
Go to *Start* ➢ *All Programs* ➢ *Administrative Tools* ➢ *Computer Management.*

A 9.1.1 Viewing Event-Log Entries

MEN's drivers use the Windows event-logging service to enter detailed error messages and other information into the Windows event log. The messages can be viewed using the Windows Event Viewer, accessible from *Start* ➢ *Programs* ➢ *Administrative Tools* ➢ *EventViewer* (NT4) or from *Computer Management* ➢ *Event Viewer* ➢ *System* (W2k/XP).

Note: To get the MEN-specific descriptions for the event entries logged by the MDIS drivers, the *men_evlg.dll* file must be present in the *%System-Root%\System32* system path of the target system.

A 9.1.2 Displaying the Used Driver Parameters (NT4)

You can examine and modify the driver parameters under NT4 using one of the registry editors (*regedit* or *regedt32*). You will find the driver parameters under the key

```
HKEY_LOCAL_MACHINE\HARDWARE\SYSTEM\CurrentControlSet\Services\XXX
```

where *XXX* is the driver name (e. g. *men_d201*).

A 9.1.3 Displaying the Used Resources

To display the resources used by MEN drivers (e. g. interrupts, memory), proceed as follows:

Windows NT 4.0 Drivers under NT4

☑ Open the Windows NT Diagnostics and select the register resources. (*Start* ➢ *Programs* ➢ *AdministrativeTools* ➢ *Windows NTDiagnostics*)

☑ You can see the devices created from the drivers by selecting Devices. To view the resources assigned to a certain device, just double-click on the device.

☑ To view all resources of one category, select this category (e. g. IRQ).

Windows 2000 PnP Drivers under W2k/XP

Use the *Resource* tab of the device's property page to view the assigned resources. For further details, refer to Chapter A 6.1 W2k Device Parameters on page 53.

Windows NT 4.0 Drivers under W2k/XP

If you use Windows NT 4.0 drivers under W2k/XP, you can see the NT4 drivers in the Device Manager's "Non-Plug and Play Drivers" tree if the option "Show hidden devices" is selected.

Windows may report some resource conflicts in the device manager. This is because the Windows NT 4.0 drivers notify the required resources for a device to Windows 2000/XP but the OS itself assigns the resources to PnP devices. Now, if an NT4 driver serves a PnP device, Windows will report resource conflicts for the NT4 driver and the PnP device. However, in this case, just ignore the reported resource conflicts.

A 9.1.4 Getting Revision Information on MDIS Modules

You can use *MDISNT* on the target system to get revision information of all software modules used by the driver:

☑ Start *mdisnt.exe* in a DOS box of the target system.

☑ Open a handle to a device of the device driver (command: *-mo*, device name: *<dev>*).

☑ Get the revision strings of the software modules used by the board driver and device driver (command *-rev*).

☑ Exit *MDISNT* (command: *-e*).

For further information on MDISNT refer to Chapter A 8.6 MDISNT Test and Configuration Utility on page 74.

A 9.1.5 Getting *.sys/.dll* File Information

You can get file information for drivers (*.sys*) or DLLs (*.dll*) via context menu item *Properties* of the desired file in your Windows Explorer.

The file information may include:

- File version, e.g. *1.2.0.0*
- File description, e.g. *Windows 2000 PnP Driver (checked build)*
- File comments, e.g. *MDIS4 Device Driver*
- Original file name, e.g. *men_p13.sys*
- Product name, e.g. *13P013-70*
- Product version, e.g. *1.3.0.0*

A 9.1.6 Examining Dependencies of Executables

Use Steve P. Miller's Dependency Walker to scan any exe, dll, or sys module and get a hierarchical tree diagram of all dependent modules. For each module found, Dependency Walker lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. It is also very useful for troubleshooting system errors related to loading and executing modules.

Dependency Walker is a free utility that is included in some MS products (e.g. Visual Studio) and can be downloaded from some web servers (e.g. www.dependencywalker.com).

A 9.1.7 Viewing the PCI Configuration Space

Use a PCI exploration tool to list all PCI devices in your system and to get the PCI Configuration Space parameters of each device. We recommend PCIScope from APSoft, a powerful tool designed to explore, examine and debug PCI subsystems of your computer. The option to save PCI subsystem information to a file is useful for exploring, comparing and debugging remote machines (e.g. for MEN support). For further information and to get a trial version refer to www.tssc.de.

A 9.1.8 Displaying Debug Output from Checked Modules

The checked builds of MDIS4 software modules provide debug outputs, which give you information about the currently working function, errors, warnings and some function-specific information. See Chapter B 4.5 Driver Debugging on page 124 for a detailed description.

Under Windows, you can display the debug output using a debugger (e. g. *SoftICE*, *WinDbg*) or with one of the following free debug monitoring programs:

- *Debug Monitor* from www.osr.com
- *DebugView* from www.sysinternals.com

A 9.2 Problems and Solutions

A 9.2.1 NT4 Driver Does Not Start

If a device driver or board driver cannot be started, please check the following items:

- ☑ Is the driver descriptor successfully inserted into the registry?
  (See Chapter A 5.4 NT4 Driver Descriptor Files on page 41.)
- ☑ Are the driver parameters in the registry (*BOARD_NAME, DEVICE_SLOT, PCI_BUS_PATH, etc.*) properly adapted to the system configuration?
  (See Chapter A 5.4 NT4 Driver Descriptor Files on page 41.)
- ☑ Did you reboot the target system after driver installation?
  (See Chapter A 4.3 Installing Windows NT 4.0 Drivers on page 22.)
- ☑ Are the driver *.sys* files located under *%SystemRoot%\System32\Drivers*?
- ☑ Are all other drivers that will be used by the driver started?
  Remember, a device driver needs at least one board driver.
  (See Chapter A 1.5 How MDIS4 Maps into the Windows NT Architecture on page 10.)
- ☑ If the driver's descriptor parameters were cloned from another target system, verify that the *Enum* subkey under the driver's entry in the registry was not copied by mistake. Otherwise, delete the *Enum* key manually using a registry editor. (See Chapter Subkey Enum on page 40).
- ☑ Is the hardware (boards/devices) that will be handled by the driver properly installed (plugged) in the system?
  Some hardware modules need an external voltage supply or further hardware adapters.
- ☑ Does the driver match the installed hardware?
  Some drivers come with various driver versions (swapped version, version for I/O address space).
- ☑ Check the Windows event log to get further details why the driver was not started.
  (See Chapter A 9.1.1 Viewing Event-Log Entries on page 76.)
- ☑ Consult the user manual for the driver and the user manual for the hardware for further driver-specific requirements.

A 9.2.2 NT4 Driver Does Not Stop

If a device driver or board driver cannot be stopped, please check the following:

- ☑ Are all handles from superior drivers as well as paths from applications that refer to the driver closed?
  (See Chapter A 5.1 Starting and Stopping NT4 Drivers on page 36.)

A 9.2.3 Device Driver Does Not Work

If a device driver does not work properly, please check the following items:

- ☑ Make sure that the driver was started successfully. You can get information about the driver start from the Windows event log.
  (See Chapter A 9.1.1 Viewing Event-Log Entries on page 76.)
- ☑ NT4 driver: Are the driver-specific descriptor parameters in the registry properly adapted to the system configuration?
  (See Chapter A 5.4 NT4 Driver Descriptor Files on page 41.)
- ☑ Is the hardware (boards/devices) that will be handled by the driver, properly installed (plugged) in the system?
  Some hardware modules need an external voltage supply or further hardware adapters.
- ☑ Does the driver match the installed hardware?
  Some drivers come with various driver versions (swapped version, version for I/O address space).
- ☑ Consult the user manual for the driver and the user manual for the hardware for further driver-specific requirements.

A 9.2.4 Strings of Event-log Entries are Missing

- ☑ Check if the *men_evlg.dll* file resides in the *%SystemRoot%\System32* folder of your target system.
  (See Chapter A 4 Installing the Target System on page 18.)
- ☑ NT4 driver: Check if all drivers are successfully added under the *EventLog* entry in the registry.
  (See Chapter A 5.4.2.5 Device and Board Driver Keys on page 46.)

A 9.2.5 W2k Device Cannot be Opened

If an MDIS application cannot open a device (e.g. *m66_1*) created by a Windows 2000 PnP driver:

- ☑ Check that the MDIS application was linked with the static or using the dynamic (DLL) MDIS-API library version 3.0 or later. See Chapter A 8 Writing Applications for MDIS on page 64 for further details.

A 9.2.6 Cannot Link C/C++ Application with Static MDIS API Libraries

Make sure that the application calls all MDIS API functions using the *__stdcall* calling convention, because the MDIS API libraries were built using this calling convention (see Chapter A 8.2.1 Using Static MDIS API Libraries on page 66).

A 10 Performance

The programs used for performance measurement are statically linked with the MDIS-API library. If an application uses the DLL version of the MDIS-API library (*men_mdis_api.dll*), the duration time of an MDIS-API call may be about 5 to 15% longer than a call with the static MDIS-API library.

A 10.1 MDIS-API Calls without Hardware Access

The following measurements show the performance of MDIS4 for Windows without any hardware access. The measured times are the duration from the point when an application calls an MDIS-API function until the point when the called function returns.

Note: The measured times are based on the resolution of the Windows high-resolution performance counter (approx. 0.84µs).

A 10.1.1 NT4 Drivers on 200MHz D1 CPU

System Configuration

• CPU board:	D1 200MHz MMX, 96 MB RAM
• Carrier board:	D201
• M-Module:	None
• Operating system:	Windows NT 4.0 US, SP 5
• NT4 board driver:	*men_d201.sys* (rev. 1.7)
• NT4 device driver:	*men_mt.sys* (rev. 1.1)
• Test program:	*mt_bench.exe* (rev. 1.1)
• MDIS-APIlibrary:	*mdis_api.lib* (rev. 1.7)

Test Results

```
mt_bench pathcount=1000 callcount=10000 devicename=mt_1

Initial OPEN       :     1 calls     100036.709 total usec
                                                  100036.709 usec/call
Further OPENs      :  1000 calls     129791.599 total usec
                                                     129.792 usec/call
GETSTAT            : 10000 calls     241047.887 total usec
                                                      24.105 usec/call
SETSTAT            : 10000 calls     264338.550 total usec
                                                      26.434 usec/call
READ               : 10000 calls     233013.069 total usec
                                                      23.301 usec/call
WRITE              : 10000 calls     232022.441 total usec
                                                      23.202 usec/call
GETBLOCK           : 10000 calls     222136.271 total usec
                                                      22.214 usec/call
SETBLOCK           : 10000 calls     193579.856 total usec
                                                      19.358 usec/call
Further CLOSES's   :  1000 calls      21220.568 total usec
                                                      21.221 usec/call
Terminating CLOSE  :     1 calls        173.486 total usec
                                                     173.486 usec/call
```

A 10.1.2 NT4/W2k Drivers on 1.2GHz F7N CPU

System Configuration

• CPU board:	F7N 1.2GHz Celeron, 256MB RAM
• Carrier board:	F201
• M-Module:	None
• Operating system:	Windows 2000 Professional US
• NT4/W2k board driver:	*men_f201.sys* (rev. 2.4)
• NT4/W2k device driver:	*men_mt.sys* (rev. 1.1)
• Test program:	*mt_bench.exe* (rev. 1.2)
• MDIS-APIlibrary:	*mdis_api.lib* (rev. 3.0)

Test Results with NT4 Drivers

```
mt_bench pathcount=1000 callcount=10000 devicename=mt_1

Initial OPEN       :     1 calls       94490.195 total usec
                                                   94490.195 usec/call
Further OPEN's     :  1000 calls      530639.919 total usec
                                                     530.640 usec/call
GETSTAT            : 10000 calls     43092.336 total usec
                                                       4.309 usec/call
SETSTAT            : 10000 calls     43535.689 total usec
                                                       4.354 usec/call
READ               : 10000 calls     39935.232 total usec
                                                       3.994 usec/call
WRITE              : 10000 calls     40439.765 total usec
                                                       4.044 usec/call
GETBLOCK           : 10000 calls     32728.452 total usec
                                                       3.273 usec/call
SETBLOCK           : 10000 calls     32435.957 total usec
                                                       3.244 usec/call
Further CLOSES's   :  1000 calls     4359.771 total usec
                                                       4.360 usec/call
Terminating CLOSE  :     1 calls     50.286 total usec
                                                      50.286 usec/call
```

Test Results with W2k Drivers

```
mt_bench pathcount=1000 callcount=10000 devicename=mt_1

Initial OPEN       :     1 calls     98705.814 total usec
                                                   98705.814 usec/call
Further OPEN's     :  1000 calls     1523133.101 total usec
                                                    1523.133 usec/call
GETSTAT            : 10000 calls     45541.250 total usec
                                                       4.554 usec/call
SETSTAT            : 10000 calls     46420.412 total usec
                                                       4.642 usec/call
READ               : 10000 calls     43275.041 total usec
                                                       4.328 usec/call
WRITE              : 10000 calls     43260.793 total usec
                                                       4.326 usec/call
GETBLOCK           : 10000 calls     36218.280 total usec
                                                       3.622 usec/call
SETBLOCK           : 10000 calls     36494.852 total usec
                                                       3.649 usec/call
Further CLOSES's   :  1000 calls     5147.580 total usec
                                                       5.148 usec/call
Terminating CLOSE  :     1 calls     158.400 total usec
                                                     158.400 usec/call
```

A 10.2 MDIS-API Calls with Hardware Access

This measurements show the performance of MDIS4 for Windows with hardware access to an M66 M-Module. The measured times are the duration from the point when an application calls an MDIS-API function until the point when the called function returns.

Note: The measured times are based on the resolution of the Windows system timer (10ms).

A 10.2.1 NT4 Drivers on 200MHz D1 CPU

System Configuration

• CPU board:	D1 200MHz MMX, 96 MB RAM
• Carrier board:	D201
• M-Module:	M66
• Operating system:	Windows NT 4.0 US, SP 5
• NT4 board driver:	*men_d201.sys* (rev. 1.7)
• NT4 device driver:	*men_m66.sys* (rev. 1.3)
• test program:	*m66_perf.exe* (rev. 1.1)
• MDIS-APIlibrary:	*mdis_api.lib* (rev. 1.7)

Test Results

```
m66_perf callcount=10000 verbose=no
M_write:
 10000 calls:  370msec =>  37.000usec / call
M_read:
 10000 calls:  361msec =>  36.100usec / call
M_setblock:
 10000 calls:  851msec =>  85.100usec / call
M_getblock
 10000 calls: 1302msec => 130.200usec / call
```

A 10.2.2 NT4/W2k Drivers on 1.2GHz F7N CPU

System Configuration

• CPU board:	F7N 1.2GHz Celeron, 256MB RAM
• Carrier board:	F201
• M-Module:	M66
• Operating system:	Windows 2000 Professional US
• NT4/W2k board driver:	*men_f201.sys* (rev. 2.4)
• NT4/W2k device driver:	*men_m66.sys* (rev. 1.5)
• Test program:	*m66_perf.exe* (rev. 1.2)
• MDIS-API library:	*mdis_api.lib* (rev. 3.0)

Test Results with NT4 Drivers

```
m66_perf callcount=10000 verbose=no

M_write:
 10000 calls: 50msec => 5.000usec / call
M_read:
 10000 calls: 50msec => 5.000usec / call
M_setblock:
 10000 calls: 721msec => 72.100usec / call
M_getblock
 10000 calls: 1172msec => 117.200usec / call
```

Test Results with W2k Drivers

```
M_write:
 10000 calls: 61msec => 6.100usec / call
M_read:
 10000 calls: 60msec => 6.000usec / call
M_setblock:
 10000 calls: 711msec => 71.100usec / call
M_getblock
 10000 calls: 1201msec => 120.100usec / call
```

A 11 Development Tools and Resources

A 11.1 Development Tools

Visual C++ Professional Edition

High-performance development environment for 32-bit Windows C/C++ applications and drivers. Contains C++ Compiler/Linker for IX86 platforms and further tools.

Manufacturer: Microsoft, www.microsoft.com

MSDN Professional Subscription

MSDN Subscriptions is a membership service which delivers essential programming information, the latest Microsoft software and tools, each month, on CD-ROM or DVD-ROM, by choice. MSDN subscribers also receive updates, service packs, selected betas, and new releases shipped throughout the year.

The Professional Subscription contains among others:

- The MSDN Library, an essential reference for developers, with more than a gigabyte of technical programming information, including sample code, documentation, technical articles and the Microsoft Developer Knowledge Base.
  The MSDN Library is also accessible from the web (see Chapter A 11.3 Resources on the Web on page 87).
- The Visual C++ Professional Edition
- Platform Software Development Kit (SDK)
- Device Driver Kit (DDK)

Manufacturer: Microsoft, www.microsoft.com

A 11.2 Literature

- Art Baker; The Windows NT Device Driver Book; Prentice Hall; ISBN 0-13-184474-1
- Peter G. Viscarola & W. Anthony Mason; Windows NT Device Driver Development; MTP; ISBN 1-57870-058-2
- Edward N. Decker & Joseph M. Newcomer; Developing Windows NT Device Drivers; Addison-Wesley; ISBN 0-201-69590-1
- David A. Solomon, Mark E. Russinovich; Inside Windows 2000 Third Edition; Microsoft Press; ISBN 0-7356-1021-5

A 11.3 Resources on the Web

- www.asktheoracle.com/driver
  Device Driver Resource Page — A reference page for programmers developing device drivers for Microsoft Windows NT/2000/XP.
  It contains links to specifications, documentation and software that are useful in developing device drivers.
- www.osr.com
  The website of Open Systems Resources, Inc. (OSR), a company specialized for the Windows NT/2000/XP system internals.
  You will find various resources (technical articles as well as tools) related to the Windows NT/2000/XP internals on this site.
- www.sysinternals.com
  The Sysinternals website provides advanced utilities, technical information and source code related to Windows internals.
- www.microsoft.com/hwdev
  The Windows Driver and Hardware Development site provides tools, information, and services for driver developers and hardware designers who create products that work with one of the Microsoft Windows operating systems.
- www.msdn.microsoft.com
  The Microsoft MSDN online website with essential resources for developers. From there, you have access to the MSDN Library that contains a bounty of technical programming information, including sample code, documentation, technical articles, and reference guides.

Part B Common MDIS Reference

B 1 MBUF Device I/O

B 1.1 Channels

Each device is logically divided into several channels. Every channel I/O access via *M_read()* and *M_write()* and some of the status calls refer to the **current channel**.

You can obtain the total number of device channels using GetStat call *M_MK_LL_CH_NUMBER*.

B 1.2 Channel I/O

The functions *M_read()* and *M_write()* can be used to read from the current channel of a device or to write a value to it.

B 1.2.1 Channel I/O Modes

The *M_MK_IO_MODE* status code is used to define/query the mode in which all channel I/O to the device is executed. This only affects functions *M_read()* and *M_write()*.

Table B1. *Channel I/O Modes*

I/O Mode	Description
M_IO_EXEC	I/O without increment[1]
M_IO_EXEC_INC	I/O with auto-increment

[1] Default mode

In ***M_IO_EXEC*** mode, I/O is directly done to the current channel of the device.

M_IO_EXEC_INC mode is the same as *M_IO_EXEC* but with subsequent incrementation of the current channel.

B 1.2.2 Channel Direction

Each I/O channel has a specific I/O direction as

- input channel
- output channel
- input/output channel.

The I/O direction may be fixed or changeable depending on the hardware and the device driver implementation. If the direction is changeable, you can use the *M_LL_CH_DIR* SetStat call to change it. If it is not, an error is returned. The current channels direction can always be queried with the *M_LL_CH_DIR* GetStat call.

Each access using *M_read()* or *M_write()* is checked for I/O direction and an error is returned if the direction is illegal.

B 1.3 Block I/O

To read or write blocks of data to the device or to the I/O buffers, you must use functions *M_getblock()* and *M_setblock()*. Depending on the device driver implementation, block I/O may be used to

- read/write a block of data directly from/to the hardware
- read/write a block of data from/to a device driver's buffer.

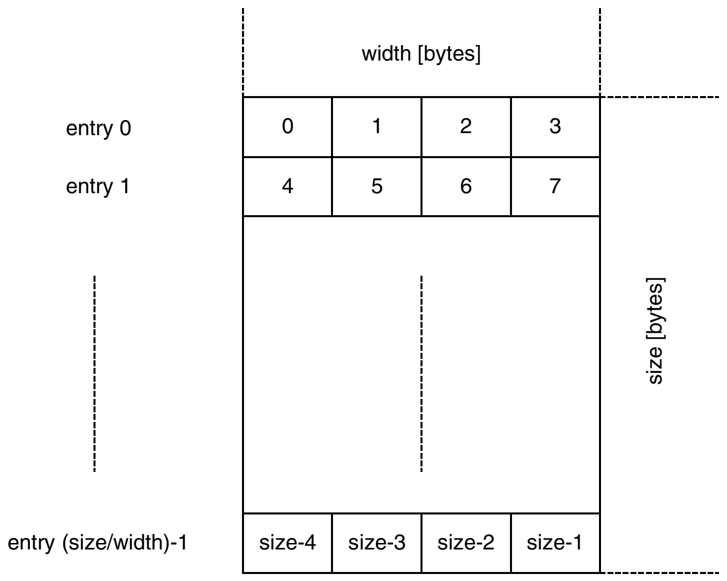
Both block I/O functions return the count of bytes transferred. If the block size requested by the application is too small, the function returns an error.

The following chapter describes the handling of device driver buffers. If the driver supports only hardware access, the described functionality is not available.

B 1.3.1 Driver Buffers

Driver buffers are allocated and controlled by the device driver. Depending on the device driver implementation a buffer may contain data for a single channel or for multiple channels. At most each channel may have its own input and/or output buffer.

Figure B1. *Buffer Structure*



The **buffer size** may be specified in the device descriptor. Otherwise a default size is used. The driver rounds-down the specified size if needed.

Each buffer has a specific **buffer width** which describes the minimum amount of data bytes that can be read from or written into the buffer. This can be seen as the size of one "entry" in the buffer.

The **buffer counter** reports the amount of available data bytes in input buffers and the free space in output buffers.

The buffer counter is updated each time an application or the interrupt service routine read data from or write data into the buffer. Handling of the counter depends on the selected block I/O mode and is described with the corresponding buffer mode.

Buffer **size**, **width** and **counter** can be queried through GetStat calls.

The buffer can be reset (logically cleared) and for debug purposes also filled with zero (physically cleared) using SetStat calls.

For ring buffers further parameters can be set or queried as described with the corresponding buffer mode:

- Overflow/underrun error handling
- Read/write timeout
- High/lowwater marks
- Input buffer high/lowwater signal
- Output buffer high/lowwater signal

The structure of a buffer and the available block I/O modes always depend on the device driver implementation. It is described in the respective device driver user manual.

B 1.3.2 Block I/O Modes

For all block I/O done with the *M_getblock()* and *M_setblock()* functions the selected block I/O mode defines the action to be performed and the type of buffer to be used.

The block I/O mode can be selected

- via the *M_BUF_RD_MODE* SetStat call for each input buffer and
- via the *M_BUF_WR_MODE* SetStat call for each output buffer.

If the driver does not support the specified block mode, it will return an error.

If separate buffers are available for more than one channel, the above status calls refer to the buffer of the current channel.

The number of buffers and the supported buffer modes are device-dependent and described in the respective device driver user manual.

Table B2. *Block I/O Modes*

Block I/O Mode	Description	Buffer Location	Blocked
M_BUF_USRCTRL	User-controlled buffer[1]	Application	No
M_BUF_CURRBUF	MDIS-controlled current buffer	Driver	No
M_BUF_RINGBUF	MDIS-controlled ring buffer	Driver	Yes
M_BUF_RINGBUF_ OVERWR	MDIS-controlled ring buffer (self-overwriting)	Driver	No

[1] Default mode

M_BUF_USRCTRL mode allows direct I/O to the device, i.e. you can bypass the I/ O buffers.

Note: In all other buffer modes, driver-internal buffers are used for I/O. The block I/ O functions copy data from the driver's buffer into the application's buffer or vice versa.

M_BUF_CURRBUF mode uses only the first entry of a buffer. This buffer can be continuously overwritten and therefore always contains the currently valid, i.e. the last, recently read/written I/O values.

In ***M_BUF_RINGBUF*** mode the input or output buffer behaves as an endless ring buffer. Input/output is **blocked**, i.e. if the requested block is not available a sleep/ wake-up mechanism takes effect.

M_BUF_RINGBUF_OVERWR mode is similar to the ring buffer; but if the buffer is full, the oldest entries will be overwritten.

Changing the block I/O mode will always reset the buffers.

B 1.3.2.1 User Control Mode

In **User Control Mode**, the MDIS-controlled buffers are not used. The buffer is provided by the application and reading/writing to/from the device is directly done into these user buffer.

- Use *M_getblock()* to read directly from the device into the application buffer.
- Use *M_setblock()* to write from the application buffer directly to the device.

The structure of the buffer is device-dependent and is described in the respective device driver user manual.

B 1.3.2.2 Current Buffer Mode

In **Current Buffer Mode**, MDIS allocates and manages the buffer. The buffer is provided by the driver and reading/writing to/from the device is done asynchronously via the buffer.

The **Current Buffer** is a self-overwriting buffer providing space for one buffer entry. The remaining buffer space is not used in this mode:

- An **input buffer** is filled with data by the driver's interrupt service. This process is interrupt-triggered. Asynchronously the application can read data from the buffer using *M_getblock()*. The buffer is not blocked, i.e. if there is no (new) data in the buffer, the getblock call doesn't wait, but returns with the old data.
- An **output buffer** is filled with data using *M_setblock()*. Asynchronously the data is written to the device by the driver's interrupt service. This process is interrupt-triggered. The buffer is not blocked, i.e. if the last written value has not been written to the device, the setblock call doesn't wait, but overwrites the old data.

Figure B2. *Current Input Buffer*

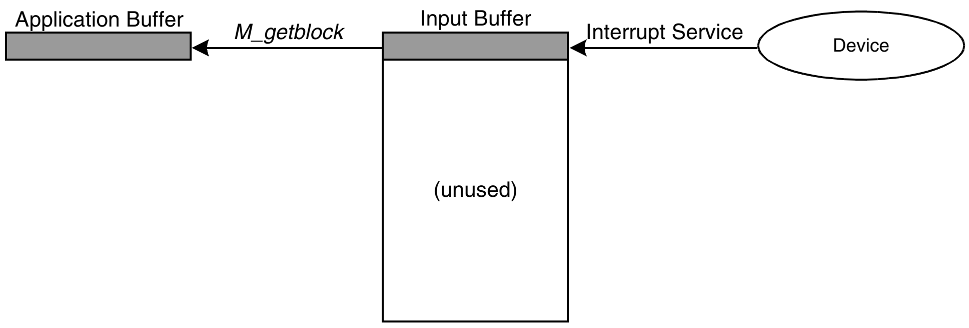
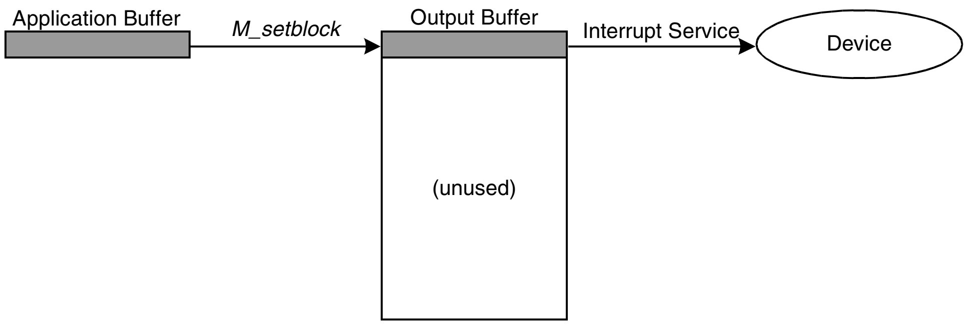


Figure B3. *Current Output Buffer*



The buffer counter is zero after initialization or after reset of the buffer. As soon as the first value is written to the buffer the buffer counter switches to ’1’ and remains at this value until the buffer is reset.

B 1.3.2.3 Ring Buffer Mode

In **Ring Buffer Mode**, MDIS allocates and manages the buffer. The buffer is provided by the driver and reading/writing to/from the device is done asynchronously via the buffer.

The **Ring Buffer** is a blocking, quasi-endless buffer providing space for several buffer entries:

- An **input buffer** is filled with data by the driver’s interrupt service. This process is interrupt-triggered. Asynchronously the application can read data from it using *M_getblock()*. The buffer is blocked, i.e. if there is no more data in the buffer, the getblock call is put to sleep until the requested data is available or a timeout has occurred.
- An **output buffer** is filled with data using *M_setblock()*. Asynchronously the data is written to the device by the driver’s interrupt service. This process is interrupt-triggered. The buffer is blocked, i.e. if there is no more space in the buffer for new data, the setblock call is put to sleep until the required space is available or a timeout has occurred.

Figure B4. *Ring Input Buffer*

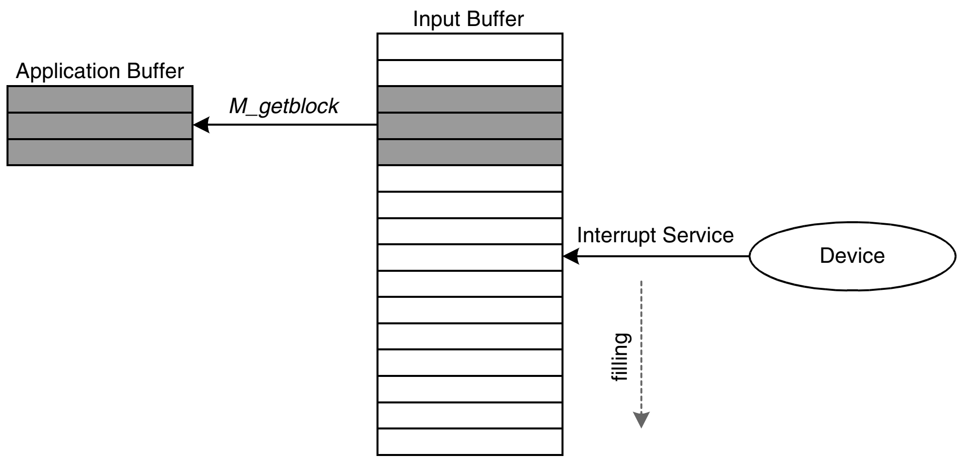
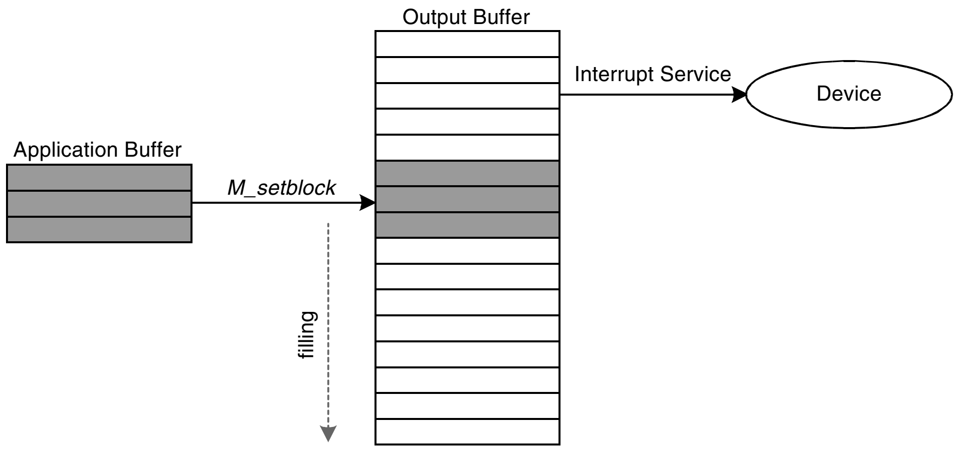


Figure B5. *Ring Output Buffer*



The buffer counter is zero after initialization or after reset of the buffer. Each time data is written to the buffer the counter is incremented. Each time data is read from the buffer, the counter is decremented.

Buffer overflow and underrun conditions cause errors if error handling was enabled via the *M_BUF_RD/WR_ERR* SetStat calls. By default, error handling is disabled:

- If the driver's interrupt service cannot write new data to an input buffer because the buffer is full, a **buffer overflow** has occurred. In this case this data is lost and the overflow error counter is incremented. If overflow error handling is enabled, the next or currently executed getblock call returns an *ERR_MBUF_OVERFLOW* error. The overflow error counter can be queried using the *M_BUF_RD_ERR_COUNT* GetStat call. It can be reset to zero via the appropriate SetStat call.

- If the drivers interrupt service cannot read new data from the output buffer because the buffer is empty, a **buffer underrun** has occurred. In this case no data is written to the device and the underrun error counter is incremented. If underrun error handling is enabled, the next or currently executed setblock call returns an *ERR_MBUF_UNDERRUN* error. The underrun error counter can be queried using the *M_BUF_WR_ERR_COUNT* GetStat call. It can be reset to zero via the appropriate SetStat call.

It is possible to read or write blocks exceeding the size of the input or output buffer. This applies to both *M_getblock()* and *M_setblock()*. In this case the application process is put to sleep several times within the function and the application buffer is filled with the requested data one part at a time.

You can limit the maximum time during which a process may stay asleep by defining a **timeout period** in the device descriptor or via the *M_BUF_RD/ WR_TIMEOUT* SetStat calls.

If this time is exceeded the process returns from the sleeping state with an *ERR_OSS_TIMEOUT* error. Setting the timeout period to zero disables the timeout.

B 1.3.2.4 Ring Buffer Mode (Self-Overwriting)

The **Self-Overwriting Ring Buffer Mode** is identical with the normal **Ring Buffer Mode**, with the exception that this buffer is never blocked on "full" or "empty" conditions.

MDIS allocates and manages the buffer. The buffer is provided by the driver and reading/writing to/from the device is done asynchronously via the buffer.

The **Self-Overwriting Ring Buffer** is a non-blocking, quasi-endless buffer providing space for several buffer entries:

- An **input buffer** is filled with data by the driver's interrupt service. This process is interrupt-triggered. Asynchronously the application can read data from it using *M_getblock()*. The buffer is not blocked, i.e. if there is no more new data in the buffer, the getblock call returns the oldest data again.
- An **output buffer** is filled with data using *M_setblock()*. Asynchronously the data is written to the device by the driver's interrupt service. This process is interrupt-triggered. The buffer is not blocked, i.e. if there is no more space in the buffer for new data, the setblock call overwrites old data that has currently not been written to the device. This data is lost.

The buffer counter is zero after initialization or after reset of the buffer. Each time data is written to the buffer the counter is incremented until the buffer is full. Each time data is read from the buffer, the counter is decremented until the buffer is empty.

Buffer overflow and underrun conditions never occur in this buffer mode. The buffer timeout is not used since the getblock/setblock calls never sleep.

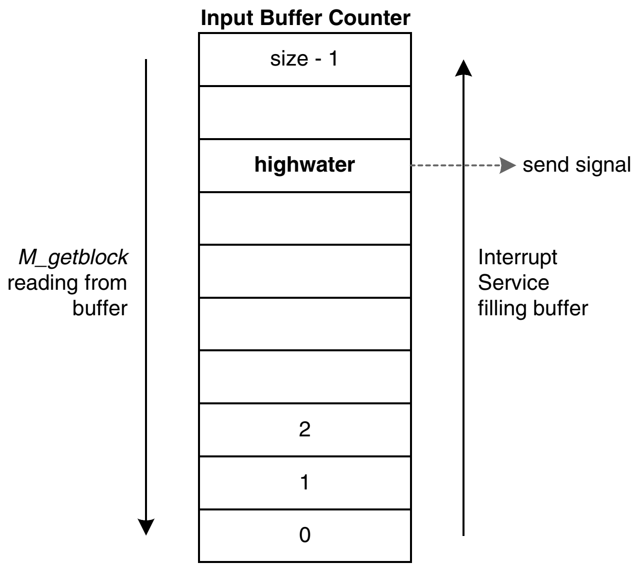
B 1.4 Buffer Events

MDIS allows a user-definable signal to be sent to the application process when certain buffer levels have been reached:

- The input buffer counter has reached the **highwater mark** when up-counting
- The output buffer counter has reached the **lowwater mark** when down-counting

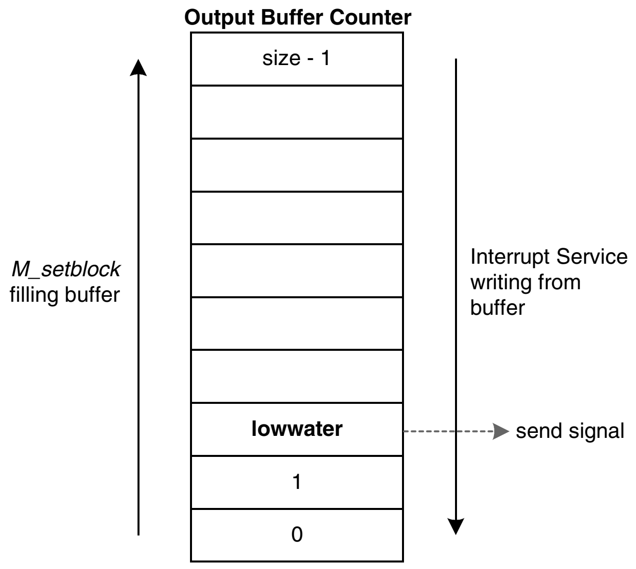
The application can activate the **highwater** signal condition via the *M_BUF_RD_SIGSET_HIGH* SetStat call and deactivate it using the *M_BUF_RD_SIGCLR_HIGH* SetStat call:

Figure B6. *Signal Input Buffer*


Input Buffer Counter
size - 1
highwater
send signal
2
1
0
M_getblock reading from buffer
Interrupt Service filling buffer

The application can activate the **lowwater** signal condition via the *M_BUF_WR_SIGSET_LOW* SetStat call and deactivate it using the *M_BUF_WR_SIGCLR_LOW* SetStat call:

Figure B7. *Signal Output Buffer*


Output Buffer Counter
size - 1
lowwater
1
0
M_setblock filling buffer
Interrupt Service writing from buffer
send signal

If the signal condition is already true when being activated, the signal is sent immediately. An activated signal condition can only be deactivated by the process that activated it.

B 2 Status Codes

Driver parameters can be changed or queried using the *M_setstat()* or *M_getstat()* functions, passing the appropriate status code. Each device driver supports a set of **common status codes** and optionally a set of **device-specific status codes**.

The common codes are for all standard parameteters supported by most device drivers, whereas the device-specific codes are for handling special parameters of the device's hardware. All device-specific codes supported by a driver are described in detail in the respective device driver user manual.

B 2.1 Status Code Types

You can make status calls using the *M_setstat()* or *M_getstat()* functions with two types of status codes:

- Standard status codes
- Block status codes

With **standard status codes** the passed value is always a 32-bit value.

With **block status codes** the passed value is always a pointer to structure *M_SG_BLOCK*, containing the application buffer's pointer and size.

Structure *M_SG_BLOCK*

```
typedef struct {
     int32 size;   /* application buffer size */
     void *data;   /* application buffer location */
} M_SG_BLOCK;
```

You must prepare this structure before calling the GetStat/SetStat function, and the pointer to this structure is passed to function parameter *value*.

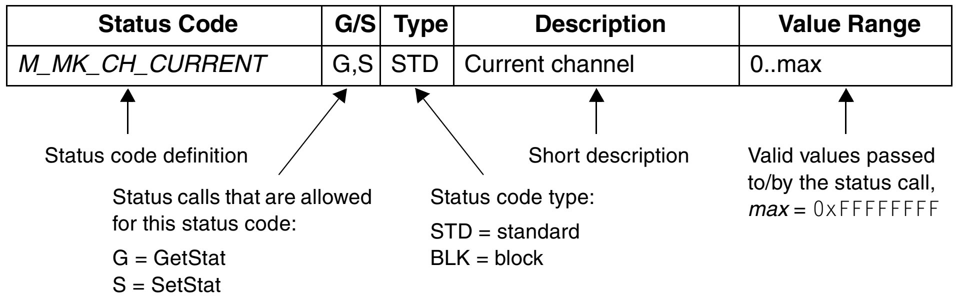
See also the example programs of the *M_setstat()* / *M_getstat()* calls.

Note: As a name convention all block status codes are named *<PREFIX>_BLK_xxxx*.

B 2.2 Common Status Codes

The following chapters describe all standard status codes. The table fields have the following meanings:

Figure B8. *Example of Status Code Table*



Status Code	G/S	Type	Description	Value Range
M_MK_CH_CURRENT	G,S	STD	Current channel	0..max

Status code definition
Status calls that are allowed for this status code:
G = GetStat
S = SetStat
Status code type:
STD = standard
BLK = block
Short description
Valid values passed to/by the status call, *max* = 0xFFFFFFFF

All definitions and typedefs (except *max*) are defined in the *mdis_api.h* include file.

B 2.2.1 MDIS Kernel Status Codes

Some of the following codes are optional. They are not supported by all MDIS implementations. Optional codes are shown with a grey background ▨.

Table B3. *MDIS Kernel Status Codes*

Status Code	G/S	Type	Description	Value Range
M_MK_CH_CURRENT	G,S	STD	Current channel	0..max
M_MK_CH_CURRENT_OLD	G,S	STD	Current channel (MDIS 3.x compatible)	1..max
M_MK_IO_MODE	G,S	STD	Channel I/O mode	*M_IO_EXEC*, *M_IO_EXEC_INC*
M_MK_IRQ_ENABLE	G,S	STD	Device interrupt enable	0 = disable, 1 = enable
M_MK_IRQ_COUNT	G,S	STD	Global interrupt counter	0..max
M_MK_IRQ_INFO	G	STD	Board interrupt capabilities	*BBIS_IRQ_DEVIRQ, BBIS_IRQ_EXPIRQ*
M_MK_IRQ_MODE	G	STD	Board interrupt mode flag(s)	*BBIS_IRQ_NONE, BBIS_IRQ_EXCEPTION, BBIS_IRQ_EXCLUSIVE, BBIS_IRQ_SHARED*
M_MK_IRQ_INSTALLED	G	STD	Interrupt service installed	0..1
M_MK_TICKRATE	G	STD	System tick rate [tics/s]	
M_MK_NBR_ADDR_SPACE	G	STD	Number of device address spaces	0..max
M_MK_BLK_PHYSADDR	G	BLK	Physical address and size (of given space)	See below
M_MK_BLK_VIRTADDR	G	BLK	Virtual address and size (of given space)	See below
M_MK_BLK_BB_HANDLER	G	BLK	Board handler name	See below
M_MK_BLK_BRD_NAME	G	BLK	Board name	See below
M_MK_BLK_LL_HANDLER	G	BLK	Device driver name	See below

Status Code	G/S	Type	Description	Value Range
M_MK_BLK_DEV_NAME	G	BLK	Device name	See below
M_MK_BLK_HW_NAME	G	BLK	Hardware name	
M_MK_BLK_REV_ID	G	BLK	Software revision ID string	See below
M_MK_REV_SIZE	G	STD	Software revision ID string size [bytes]	See below
M_MK_DEBUG_LEVEL	G,S	STD	Debug level of MDIS kernel	See *dbg.h*
M_MK_API_DEBUG_LEVEL	G,S	STD	Debug level of MDIS API	See *dbg.h*
M_MK_OSS_DEBUG_LEVEL	G,S	STD	Debug level of operating system services	See *dbg.h*
M_MK_LOCKMODE	G	STD	Process lock mode	*LL_LOCK_NONE, LL_LOCK_CALL, LL_LOCK_CHAN*
M_MK_PATHCNT	G	STD	Opened paths on device	1..max
M_MK_DEV_SLOT	G	STD	Device slot of board	0..max
M_MK_DEV_ADDRMODE	G	STD	Device addr mode flag(s)	*MDIS_MA08, MDIS_MA24*
M_MK_DEV_DATAMODE	G	STD	Device data mode flag(s)	*MDIS_MD08, MDIS_MD16, MDIS_MD32*
M_MK_BUSTYPE	G	STD	Board bus system	*OSS_BUSTYPE_NONE, OSS_BUSTYPE_VME, OSS_BUSTYPE_PCI, OSS_BUSTYPE_ISA*

M_MK_CH_CURRENT sets and queries the current channel. Channel numbers start with 0. *M_MK_CH_CURRENT_OLD* exists only for compatibility with MDIS V3.x, where channel numbers start with 1.

M_MK_IO_MODE sets and queries the I/O mode for the read/write calls.

M_MK_IRQ_ENABLE enables or disables the device interrupt at runtime.

M_MK_IRQ_COUNT reads (G) or clears (S) the global interrupt counter.

M_MK_IRQ_INFO queries the board's interrupt capabilities. The returned flag(s) show if the board supports a device interrupt (*BBIS_IRQ_DEVIRQ*) and/or a board exception interrupt (*BBIS_IRQ_EXPIRQ*). See board's interrupt definitions in *bb_defs.h*.

M_MK_IRQ_MODE queries the board's interrupt mode. The returned flag(s) show, if the device interrupt is exclusive for the device slot (*BBIS_IRQ_EXCLUSIVE*), or if it is shared with device interrupts of other board slots (*BBIS_IRQ_SHARED*). Also, it shows if the device interrupt is shared with a board exception interrupt. See board's interrupt definitions in *bb_defs.h*.

M_MK_IRQ_INSTALLED shows if the interrupt service routine for the device is installed (1) or not (0).

M_MK_TICKRATE returns system's ticker rate in ticks per second.

M_MK_NBR_ADDR_SPACE returns the number of address spaces required by the device. For all of the returned address spaces, subsequent calls of the *M_MK_BLK_PHYS/VIRTADDR* GetStats are possible.

M_MK_BLK_PHYSADDR queries the physical address and requested size of a given address space (index). The structure *M_ADDRSPACE* is passed via parameter *data* of the *M_SG_BLOCK* structure. Before the GetStat call can be made, parameter *space* of the *M_ADDRSPACE* structure must be set to the address space which should be queried. After the GetStat call the *addr* and *size* parameters of the *M_ADDRSPACE* structure are initialized with the resulting values:

```
typedef struct {
      native_int space;   /* in:  address space (index) */
      native_int addr;    /* out: start address of address space */
      native_int size;    /* out: size of address space [bytes] */
} M_ADDRSPACE;
```

Note: All parameters of the *M_ADDRSPACE* structure represent the target system's native integer format, being declared as *native_int*.

M_MK_BLK_VIRTADDR queries the virtual address and available (mapped) size of a given address space (index) in the same way as *M_MK_BLK_PHYSADDR* (see above). For systems without virtual address management, the code will return the same values as ***M_MK_BLK_PHYSADDR***.

Getstats ***M_MK_BLK_BB_HANDLER***, ***M_MK_BLK_BRD_NAME***, ***M_MK_BLK_LL_HANDLER***, ***M_MK_BLK_DEV_NAME*** and ***M_MK_BLK_HW_NAME*** can be used to query the name of boards/devices and the corresponding board handler/drivers. A character array of size *M_MAX_NAME* is passed via parameter *data* of the *M_SG_BLOCK* structure. After the GetStat call the array contains a zero-terminated name string:

```
char namestr[M_MAX_NAME];
```

M_MK_BLK_REV_ID queries the revision ID strings of all driver modules which are used by the driver. A character array is passed via parameter *data* of the *M_SG_BLOCK* structure. After the GetStat call the array contains multiple lines terminated with the system's native carriage return character. The last line is zero-terminated. The required size of the character array can be queried by the *M_MK_REV_SIZE* GetStat call.

The resulting lines consist of the driver module's name and description and the ID string of the revision control system in the following format:

```
<mod_name> - <mod_description>: <rcs_id>
```

M_MK_DEBUG_LEVEL, ***M_MK_API_DEBUG_LEVEL*** and ***M_MK_OSS_DEBUG_LEVEL*** define the debug level of a debug driver. This influences the number of debug messages which are produced. See debug level definitions in *dbh.h* and Chapter B 4.5 Driver Debugging on page 124.

M_MK_LOCKMODE queries the process lock mode (none, call or channel locking) of the driver. See lock mode definitions in *ll_defs.h*.

M_MK_PATHCNT queries how many processes have an opened path to a device.

M_MK_DEV_SLOT queries which slot on the board is used by the device.

M_MK_DEV_ADDRMODE and ***M_MK_DEV_DATAMODE*** query the hardware characteristics of the device. The returned flag(s) show which address and data modes are supported from the device. For local devices (directly connected to the CPU), no flags are returned. See address/data mode definitions in *mdis_com.h*.

M_MK_BUSTYPE queries the board's bus system type: *OSS_BUSTYPE_NONE* specifies a local device (directly connected to the CPU), *OSS_BUSTYPE_VME* specifies VMEbus and *OSS_BUSTYPE_ISA* PC ISA bus. See bus system definitions in *oss.h*.

B 2.2.2 Input Buffer Management Status Codes

If more than one input buffer is supported by the driver, the following status calls refer to the input buffer of the current channel.

Table B4. *Input Buffer Management Status Codes*

Status Code	G/S	Type	Description	Value Range
M_BUF_RD_MODE	G,S	STD	Input buffer block I/O mode	*M_BUF_USRCTRL, M_BUF_CURRBUF, M_BUF_RINGBUF, M_BUF_RINGBUF_OVERWR*
M_BUF_RD_ERR	G,S	STD	Overflow error enable	0 = disable, 1 = enable
M_BUF_RD_SIGSET_HIGH	G,S	STD	Highwater signal activate	See below
M_BUF_RD_SIGCLR_HIGH	S	STD	Highwater signal deactivte	0
M_BUF_RD_HIGHWATER	G,S	STD	Highwater mark [bytes]	0..max
M_BUF_RD_TIMEOUT	G,S	STD	Read timeout [ms]	0 = none, 1..max
M_BUF_RD_BUFSIZE	G	STD	Input buffer size [bytes]	0..max
M_BUF_RD_WIDTH	G	STD	Input buffer width [bytes]	0..max
M_BUF_RD_COUNT	G	STD	Input buffer counter [bytes]	0..max
M_BUF_RD_ERR_COUNT	G,S	STD	Overflow error counter	0..max
M_BUF_RD_RESET	S	STD	Logically reset input buffer	0
M_BUF_RD_CLEAR	S	STD	Physically clear input buffer	0
M_BUF_BLK_RD_DATA	G	BLK	Read input buffer data	See below
M_BUF_RD_DEBUG_LEVEL	G,S	STD	Debug level of input buffer	

M_BUF_RD_MODE sets and queries the input buffer block I/O mode.

M_BUF_RD_ERR enables/disables the input buffer overflow error handling.

M_BUF_RD_SIGSET_HIGH activates and queries the input buffer's highwater signal. The signal to be sent is passed to the SetStat call.

M_BUF_RD_SIGCLR_HIGH deactivates an activated highwater signal.

M_BUF_RD_HIGHWATER sets and queries the input buffer's highwater mark.

M_BUF_RD_TIMEOUT sets and queries the input buffer's read timeout. The given timeout is internally rounded up to system ticks.

M_BUF_RD_BUFSIZE queries the input buffer's total size.

M_BUF_RD_WIDTH queries the minimum block size that can be read from the input buffer.

M_BUF_RD_COUNT queries the input buffer counter, i.e. the number of available bytes.

M_BUF_RD_ERR_COUNT reads (G) or clears (S) the input buffer error counter, i.e. the number of buffer overflows that have occurred.

M_BUF_RD_RESET logically resets the input buffer, i.e. internal buffer pointers and counters are reset.

M_BUF_RD_CLEAR physically clears the input buffer for debug purposes, i.e. the entire buffer space is filled with zero.

M_BUF_BLK_RD_DATA copies the entire input buffer into a user buffer passed via parameter *data* of the *M_SG_BLOCK* structure. The user buffer size must be equal to the input buffers size.

M_BUF_RD_DEBUG_LEVEL defines the debug level of the input buffer functions of a debug driver. This influences the number of debug messages which are produced. See debug level definitions in *dbh.h* and Chapter B 4.5 Driver Debugging on page 124.

B 2.2.3 Output Buffer Management Status Codes

If more than one output buffer is supported by the driver, the following status calls refer to the output buffer of the current channel.

Table B5. *Output Buffer Management Status Codes*

Status Code	G/S	Type	Description	Value Range
M_BUF_WR_MODE	G,S	STD	Output buffer block I/O mode	*M_BUF_USRCTRL, M_BUF_CURRBUF, M_BUF_RINGBUF, M_BUF_RINGBUF_OVERWR*
M_BUF_WR_ERR	G,S	STD	Underrun error enable	0 = disable, 1 = enable
M_BUF_WR_SIGSET_LOW	G,S	STD	Lowwater signal activate	See below
M_BUF_WR_SIGCLR_LOW	S	STD	Lowwater signal deactivte	0
M_BUF_WR_LOWWATER	G,S	STD	Lowwater mark [bytes]	0..max
M_BUF_WR_TIMEOUT	G,S	STD	Write timeout [ms]	0 = none, 1..max
M_BUF_WR_BUFSIZE	G	STD	Output buffer size [bytes]	0..max
M_BUF_WR_WIDTH	G	STD	Output buffer width [bytes]	0..max
M_BUF_WR_COUNT	G	STD	Output buffer counter [bytes]	0..max
M_BUF_WR_ERR_COUNT	G,S	STD	Underrun error counter	0..max
M_BUF_WR_RESET	S	STD	Logically reset output buffer	0
M_BUF_WR_CLEAR	S	STD	Physically clear output buffer	0
M_BUF_BLK_WR_DATA	G	BLK	Read output buffer data	See below
M_BUF_WR_DEBUG_LEVEL	G,S	STD	Debug level of write buffer	

M_BUF_WR_MODE sets and queries the output buffer block I/O mode.

M_BUF_WR_ERR enables/disables the output buffer underrun error handling.

M_BUF_WR_SIGSET_LOW activates and queries the output buffers lowwater signal. The signal to be sent is passed to the SetStat call.

M_BUF_WR_SIGCLR_LOW deactivates an activated lowwater signal.

M_BUF_WR_LOWWATER sets and queries the output buffer's lowwater mark.

M_BUF_WR_TIMEOUT sets and queries the output buffer's read timeout. The given timeout is internally rounded up to system ticks.

M_BUF_WR_BUFSIZE queries the output buffer's total size.

M_BUF_WR_WIDTH queries the minimum block size that can be written to the output buffer.

M_BUF_WR_COUNT queries the output buffer counter, i.e. the number of free bytes.

M_BUF_WR_ERR_COUNT reads (G) or clears (S) the output buffer error counter, i.e. the number of buffer underruns that have occurred.

M_BUF_WR_RESET logically resets the output buffer, i.e. internal buffer pointers and counters are reset.

M_BUF_WR_CLEAR physically clears the output buffer for debug purposes, i.e. the entire buffer space is filled with zero.

M_BUF_BLK_WR_DATA copies the entire output buffer into a user buffer passed via parameter *data* of the *M_SG_BLOCK* structure. The user buffer size must be equal to the output buffer's size.

M_BUF_WR_DEBUG_LEVEL defines the debug level of the output buffer functions of a debug driver. This influences the number of debug messages which are produced. See debug level definitions in *dbh.h* and Chapter B 4.5 Driver Debugging on page 124.

B 2.2.4 Device Driver Status Codes

Table B6. *Device Driver Status Codes*

Status Code	G/S	Type	Description	Value Range
M_LL_CH_NUMBER	G	STD	Number of device channels	1..max
M_LL_CH_DIR	G,S	STD	Device channel direction	*M_CH_IN, M_CH_OUT, M_CH_INOUT*
M_LL_CH_LEN	G	STD	Device channel length [bits]	0..max
M_LL_CH_TYP	G	STD	Device channel type	*M_CH_UNKNOWN, M_CH_BINARY, M_CH_ANALOG, M_CH_SERIAL*
M_LL_IRQ_COUNT	G,S	STD	Device interrupt counter	0..max
M_LL_ID_CHECK	G	STD	Device ID check enabled	0 = disabled, 1 = enabled
M_LL_DEBUG_LEVEL	G,S	STD	Debug level of device driver	See *dbg.h*
M_LL_ID_SIZE	G	STD	Device ID PROM size [bytes]	See below
M_LL_BLK_ID_DATA	G	BLK	Read device ID PROM	See below

M_LL_CH_NUMBER queries the total number of device channels.

M_LL_CH_DIR queries the current channel's direction. If supported by the driver, the channel direction can also be changed at runtime using this call.

M_LL_CH_LEN is for channel information purposes and returns the physical bit width of the current channel.

M_LL_CH_TYP is for channel information purposes and returns the type of hardware of the current channel.

M_LL_IRQ_COUNT reads (G) or clears (S) the device interrupt counter.

M_LL_ID_CHECK returns if the device's ID PROM was checked at device initialization.

M_LL_DEBUG_LEVEL allows to define the debug level for the driver's functions in debug drivers. See definitions in *dbg.h* and Chapter B 4.5 Driver Debugging on page 124.

M_LL_BLK_ID_DATA reads the raw device ID PROM data (*M_LL_ID_SIZE* bytes) from a device into a buffer passed via parameter *data* of the *M_SG_BLOCK* structure. The ID PROM data size can be queried by the *M_LL_ID_SIZE* GetStat call. If no ID PROM exists an *ERR_LL_UNK_CODE* error is returned.

B 2.2.5 Board Handler Status Codes

Table B7. *Board Handler Status Codes*

Status Code	G/S	Type	Description	Value Range
M_BB_IRQ_VECT	G	STD	Device interrupt vector	0 = none, 1..max
M_BB_IRQ_LEVEL	G	STD	Device interrupt level	0..max
M_BB_IRQ_PRIORITY	G	STD	Device interrupt priority	0..max
M_BB_IRQ_EXP_COUNT	G,S	STD	Exception interrupt counter	0..max
M_BB_ID_CHECK	G	STD	Board ID check enabled	0 = disabled, 1 = enabled
M_BB_DEBUG_LEVEL	G,S	STD	Debug level of board handler	See *dbg.h*
M_BB_ID_SIZE	G	STD	Board ID PROM size [bytes]	See below
M_BB_BLK_ID_DATA	G	BLK	Read board ID PROM	See below

M_BB_IRQ_VECT reads the installed interrupt vector. If no interrupt is installed, zero is returned as value.

M_BB_IRQ_LEVEL queries the used interrupt level.

M_BB_IRQ_PRIORITY queries the used interrupt priority (for shared interrupts).

M_BB_IRQ_EXP_COUNT reads (G) or clears (S) the exception interrupt counter.

M_BB_ID_CHECK returns if the board ID PROM was checked at device initialization.

M_BB_DEBUG_LEVEL allows to define the debug level for the board handler's functions in debug drivers. See definitions in *dbg.h* and Chapter B 4.5 Driver Debugging on page 124.

M_BB_BLK_ID_DATA reads the raw board ID PROM data from a base board into a buffer passed via parameter *data* of the *M_SG_BLOCK* structure. The ID PROM data size can be queried by the *M_BB_ID_SIZE* GetStat call. If no ID PROM exists an *ERR_BB_UNK_CODE* error is returned.

B 3 Error Codes

Note: Please see *mdis_err.h* for error numbers. The file's location depends on the operating system.

B 3.1 Operating System Specific Errors

Table B8. *Operating System Specific Errors*

Error Code	Error Message	Error Description
ERR_BAD_PATH	bad path number	The given path number is unknown to the system's path table.
ERR_PATH_FULL	path table full	The path table is full (too many open paths).
ERR_BUSERR	bus error occurred	An exception error occurred when the hardware was accessed.

B 3.2 MDIS Kernel Errors

Table B9. *MDIS Kernel Errors*

Error Code	Error Message	Error Description
ERR_MK	general error	A general error occurred.
ERR_MK_USERBUF	user buffer too small	The user buffer for a block status call is too small.
ERR_MK_UNK_CODE	unknown status code	The status code is not known.
ERR_MK_ILL_PARAM	illegal parameter	General error for illegal parameters.
ERR_MK_ILL_DESC	illegal descriptor type	An illegal descriptor type was detected in the device descriptor.
ERR_MK_ILL_MSIZE	address space size conflict	The board slot's address space is not large enough for the device's requirements.
ERR_MK_NO_LLDESC	device descriptor not found	The device descriptor does not exist.
ERR_MK_NO_BBISDESC	board descriptor not found	The board descriptor does not exist.
ERR_MK_NO_LLDRV	device driver not found	The device driver does not exist or cannot be linked.
ERR_MK_NO_BBISDRV	board handler not found	The board handler does not exist or cannot be linked.
ERR_MK_NO_IRQ	board does not support interrupts	The board has no interrupt capabilities.
ERR_MK_NO_IRQ	board doesn't support interrupt	The device needs an interrupt, but the board slot is not able to handle device interrupts.
ERR_MK_IRQ_INSTALL	can't install interrupt	The interrupt cannot be installed in the system.
ERR_MK_IRQ_REMOVE	can't remove interrupt	An installed interrupt cannot be removed from the system.
ERR_MK_IRQ_ENABLE	can't enable/disable interrupt	An installed interrupt cannot be enabled/disabled.
ERR_MK_DESC_PARAM	descriptor values out of range	An illegal descriptor value was detected in the device descriptor.

B 3.3 Device Driver Errors

Table B10. *Device Driver Errors*

Error Code	Error Message	Error Description
ERR_LL	general error	General error occurred.
ERR_LL_USERBUF	user buffer too small	The user buffer for a block status call is too small.
ERR_LL_UNK_CODE	unknown status code	The status code is not known.
ERR_LL_ILL_PARAM	illegal parameter	General error for illegal parameters.
ERR_LL_ILL_ID	wrong device id detected	The device ID check has detected a non-matching device.
ERR_LL_ILL_DIR	illegal i/o direction	The current channel direction does not match any read/write call.
ERR_LL_ILL_FUNC	ll-driver fct. not supported	An unsupported driver function was called.
ERR_LL_DEV_BUSY	device is busy	Driver call refused since the device is in a state where no i/o is possible.
ERR_LL_READ	device read error	General read error.
ERR_LL_WRITE	device write error	General write error.
ERR_LL_DESC_PARAM	descriptor values out of range	An illegal descriptor value was detected in the device descriptor.

B 3.4 Board Handler Errors

Table B11. *Board Handler Errors*

Error Code	Error Message	Error Description
ERR_BBIS	general error	General error occurred.
ERR_BBIS_USERBUF	user buffer too small	The user buffer for a block status call is too small.
ERR_BBIS_UNK_CODE	unknown status code	The status code is not known.
ERR_BBIS_ILL_PARAM	illegal parameter	General error for illegal parameters.
ERR_BBIS_ILL_ID	wrong board id detected	The board ID check has detected a non-matching board.
ERR_BBIS_NO_IRQ	can't determine interrupt parameters	An interrupt cannot be installed on the board since the board cannot determine interrupt parameters.
ERR_BBIS_ILL_IRQPARAM	illegal interrupt parameter	The board does not support the required interrupt parameters.
ERR_BBIS_ILL_SLOT	illegal board slot	The requested board slot does not exist or is already in use.
ERR_BBIS_ILL_DATAMODE	illegal address space (data mode)	The requested data access mode is not available on the board slot.
ERR_BBIS_ILL_ADDRMODE	illegal address space (address mode)	The requested address mode is not available on the board slot.
ERR_BBIS_NO_CHECKLOC	can't check board location	The PCI bus backplane is not able to check the board location.

Error Code	Error Message	Error Description
ERR_BBIS_ILL_FUNC	board handler function not supported	A not supported board handler function was called.
ERR_BBIS_DESC_PARAM	descriptor values out of range	An illegal descriptor value was detected in the board descriptor.

B 3.5 Descriptor Errors

Table B12. *Descriptor Errors*

Error Code	Error Message	Error Description
ERR_DESC	general error	General error occurred.
ERR_DESC_CORRUPTED	descriptor data corrupted	The descriptor has a wrong format.
ERR_DESC_KEY_NOTFOUND	descriptor key not found	The requested descriptor key is not defined in the descriptor.
ERR_DESC_BUF_TOOSMALL	descriptor buffer too small	Internal error when reading descriptor.

B 3.6 ID PROM Errors

Table B13. *ID PROM Errors*

Error Code	Error Message	Error Description
ERR_ID	general error	General error occurred.
ERR_ID_NOTFOUND	id prom not found	The ID PROM does not exist.
ERR_ID_CORRUPTED	id prom data corrupted	The ID PROM contains invalid data.
ERR_ID_ILL_PARAM	id prom illegal parameter	Internal error when reading ID PROM.

B 3.7 Operating System Service Errors

Table B14. *Operating System Service Errors*

Error Code	Error Message	Error Description
ERR_OSS	general error	General error occurred.
ERR_OSS_ILL_PARAM	illegal parameter	General error for illegal parameters.
ERR_OSS_UNK_BUSTYPE	unknown bus type	The bus type required for a base board is not supported.
ERR_OSS_TIMEOUT	timeout occured	A timeout has occurred while waiting for data or device response.
ERR_OSS_NO_PERM	no permission accessing memory	There is no permission to access a given user buffer.
ERR_OSS_NO_SYSCLOCK	no system ticker available	There is no systems ticker available for timer functions.
ERR_OSS_ILL_HANDLE	illegal OSS handle	Internal error.
ERR_OSS_SIG_OCCURED	signal occured	A deadly signal has been received while waiting for data or device response.
ERR_OSS_SIG_SEND	can't send signal	Internal error.
ERR_OSS_SIG_SET	can't install signal	A user defined signal cannot be installed.

Error Code	Error Message	Error Description
ERR_OSS_SIG_CLR	can't remove signal	A user defined signal cannot be deinstalled.
ERR_OSS_MEM_ALLOC	can't allocate memory	Not enough memory available.
ERR_OSS_MEM_FREE	can't free memory	Internal error.
ERR_OSS_SEM_CREATE	can't create semaphore	Internal error.
ERR_OSS_SEM_REMOVE	can't remove semaphore	Internal error.
ERR_OSS_UNK_RESOURCE	unknown ressource	An address space or interrupt is not known to the system or illegal.
ERR_OSS_BUSY_RESOURCE	busy ressource	An address space or interrupt is already in use.
ERR_OSS_MAP_FAILED	can't map address space	An address space cannot be mapped.
ERR_OSS_NO_BUSTOPHYS	can’t map bus address	The system supports no function for mapping bus to physical address.
ERR_OSS_NO_MIKRODELAY	mikrodelay not available	The microdelay function is not available.
ERR_OSS_ALARM_CREATE	can't create alarm	Internal driver error.
ERR_OSS_ALARM_REMOVE	can't remove alarm	Internal driver error.
ERR_OSS_ALARM_SET	can't install alarm routine	Internal driver error.
ERR_OSS_ALARM_CLR	can't remove alarm routine	Internal driver error.
ERR_OSS_CALLBACK_CREATE	can't create callback	Callback initialization failed.
ERR_OSS_CALLBACK_REMOVE	can't remove callback	Callback termination failed.
ERR_OSS_CALLBACK_SET	can't install callback routine	The specified callback routine cannot be installed.
ERR_OSS_CALLBACK_CLR	can't remove callback routine	The specified callback routine cannot be removed.
ERR_OSS_CALLBACK_EMPTY	callback queue empty	Internal driver error.
ERR_OSS_CALLBACK_OVER	callback queue overflow	The callback queue has overflown, callbacks may be lost.

B 3.7.1 PCI System Specific Error Codes

The following codes are only returned on PCI bus systems.

Table B15. *PCI System Specific Error Codes*

Error Code	Error Message	Error Description
ERR_OSS_PCI	general error	General error occurred.
ERR_OSS_PCI_ILL_DEV	illegal PCI device	Problem with PCI slot mapping.
ERR_OSS_PCI_ILL_DEVNBR	illegal PCI device number	Problem with PCI slot mapping.
ERR_OSS_PCI_ILL_ADDRNBR	illegal PCI address number	Unsupported PCI address.
ERR_OSS_PCI_NO_DEVINSLOT	no PCI device found in slot	No PCI device was found in the specified PCI slot.
ERR_OSS_PCI_UNK_REG	unknown PCI register	The board handler tried to read an unknown PCI register.
ERR_OSS_PCI_SLOT_TO_DEV	can’t map PCI slot to device	Internal error.

Note: The PCI specific errors do not exist on all systems!

B 3.7.2 VMEbus Specific Error Codes

The following codes are only returned on VMEbus systems.

Table B16. *VMEbus Specific Error Codes*

Error Code	Error Message	Error Description
ERR_OSS_VME	general error	General error occurred.
ERR_OSS_VME_ILL_SPACE	illegal address space	The VMEbus address space required for the board is not available on the CPU board.
ERR_OSS_VME_ILL_SIZE	illegal address space size	The CPU board's VMEbus address space is not large enough for the board's requirements.

Note: The VMEbus specific errors do not exist on all systems!

B 3.8 Buffer Management Errors

Table B17. *Buffer Management Errors*

Error Code	Error Message	Error Description
ERR_MBUF	general error	General error occurred.
ERR_MBUF_USERBUF	user buffer too small	The user buffer for a block I/O call is smaller than buffer width.
ERR_MBUF_UNK_CODE	unknown status code	The status code is not known.
ERR_MBUF_ILL_PARAM	illegal parameter	General error for illegal parameters.
ERR_MBUF_OVERFLOW	buffer overflow occured	A block read function has been aborted after an input buffer overflow occurred.
ERR_MBUF_UNDERRUN	buffer underrun occured	A block write function has been aborted after an output buffer underrun occurred.
ERR_MBUF_NO_BUF	no buffer installed	There is no buffer available.
ERR_MBUF_ILL_SIZE	illegal buffer size	The buffer size specified in the device descriptor is not possible.
ERR_MBUF_ILL_DIR	illegal buffer direction	Internal error.

B 3.9 PLD Loader Errors

Table B18. *PLD Loader Errors*

Error Code	Error Message	Error Description
ERR_PLD	general error	General error occurred.
ERR_PLD_NOTFOUND	no response from PLD	The PLD does not respond.
ERR_PLD_INIT	error initializing PLD	An error has occurred in the PLD initializing sequence.
ERR_PLD_LOAD	error loading PLD	An error has occurred while PLD was loaded.
ERR_PLD_TERM	error terminating PLD	An error has occurred in the PLD terminating sequence.

B 3.10 CPU Handler (Bus Mapper) Errors

Table B19. *CPU Handler (Bus Mapper) Errors*

Error Code	Error Message	Error Description
ERR_CBIS	general error	General error occurred.
ERR_CBIS_UNK_CODE	unknown status code	Internal error.
ERR_CBIS_ILL_PARAM	illegal parameter	Internal error.
ERR_CBIS_ILL_FUNC	function not supported	Internal error.

Note: The CPU handler errors do not exist on all systems!

B 3.11 BBIS Kernel Errors

Table B20. *BBIS Kernel Errors*

Error Code	Error Message	Error Description
ERR_BK	general error	General error occurred.
ERR_BK_ILL_PARAM	illegal parameter	General error for illegal parameters.

Note: The BBIS Kernel errors do not exist on all systems!

B 4 MDIS Device Descriptors

Although it may be useful to know some details about device descriptors, you normally won't need to know such details, especially if you use the MDIS installation wizard. We recommend to use the comfortable tools provided in the MDIS distribution. If you cannot make use of such tools, this chapter gives you any information you need.

B 4.1 General

B 4.1.1 Devices and Device Descriptors

A **device** is a piece of hardware located on a "base board" (see below). This may be

- a controller, located on a CPU board
- a mezzanine I/O device, plugged on a carrier board, e.g. M-Module or PC•MIP.

Each device is described by a so-called **device descriptor**. The name of the device descriptor is usually called **device name**.

The device descriptor describes all logical parameters of the device:

- Device driver name
- Board name (name of the board descriptor)
- Device location (slot of the board on which the device is mounted, or logical slot for on-board devices)
- Device/driver specific parameters

All physical characteristics of a device are hard-coded in the (low-level) driver and can be queried by GetStat calls:

- Required address spaces
- Info about ID PROM
- Number of channels
- Characteristics of channels

B 4.1.2 Boards and Board Descriptors

A **board**, or base board, is a piece of hardware that controls access to a device and the interrupts from this device. The base board can be

- the CPU board that contains or implements the controlled device
- a carrier board for mezzanine I/O devices, e.g. M-Module or PC•MIP carrier.

Each base board is described by a so-called **board descriptor**. The name of the board descriptor is usually called **board name**. This name is defined as a string in the device descriptor.

The board descriptor describes all logical characteristics of the board:

- Board handler name
- Board location (geographical location or physical address)
- Interrupt configuration
- Board/handler specific parameters

All physical characteristics of a base board are hard-coded in the board handler and can be queried by GetStat calls:

- Info about ID PROM
- Number of board slots
- Available address spaces of a board slot
- Interrupt parameters of a board slot

B 4.2 Descriptor Format

Descriptors are generated from a text file, the so-called **meta descriptor**, with the following format:

```
Objname1 {
      Key1 = DataType Value,...
      Key2 = DataType Value,...
      SubKey {
         Key3 = DataType Value,...
      }
}
```

The following items are used:

Table B21. *Descriptor Items*

Item	Description	Value Range
Objname	Name of the following descriptor (enclosed in {} braces)	[A..Z, 0..9, _]
Key	Name of the following descriptor field	[A..Z, 0..9, _]
DataType	Specifies the data type of the field's values	*BINARY* (byte array), *U_INT32* (single 32-bit), *STRING* (zero-terminated)
Value	Single value or comma separated list of values	depends on *DataType*
SubKey	Creates a new "sub-directory", which can contain its own keys and subkeys (enclosed in {} braces)	[A..Z, 0..9, _]

For types *BINARY* and *U_INT32* the following notations of *Value* are recognized:

`123` ◊	Decimal notation
`0x123` ◊	Hexadecimal notation
`%1011` ◊	Binary notation

Comments are allowed with trailing '#' or '//'. The rest of the line will be ignored then. Long lines may be clipped using '\'. The next line will then be interpreted as part of this line. Opening braces must be in the same line as the object or key name. Closing braces must be in a separate line.

The descriptor-generating process, i.e. the conversion from the text file to a target system specific format is described in detail in the operating sytem specific chapters of the MDIS user guide.

Note: A descriptor definition file can contain multiple descriptor objects.

B 4.3 Device Descriptor Keys

The device descriptor defines the following parameters:

- Name of the hardware
- Name of the base board
- Geographical location on the board
- Further device-specific parameters

For these standard parameters the following descriptor keys are defined:

Table B22. *Device Descriptor Keys*

Key	DataType	Description	Value Range	Req	Fix
DESC_TYPE	*U_INT32*	Descriptor type 1 = device, 2 = board	1	•	•
HW_TYPE	*STRING*	Hardware type (name of the device)	[A..Z, 0..9, _]	•	•
DEVICE_SLOT	*U_INT32*	Base board slot where device is mounted	0..0xFFFFFFFF	•	
BOARD_NAME	*STRING*	Base board name (descriptor name)	[A..Z, 0..9, _]	•	
ID_CHECK	*U_INT32*	Read and check the device ID 0 = disable, 1 = enable	0, 1		
IRQ_ENABLE	*U_INT32*	Enable interrupt immediately (after init) 0 = no, 1 = yes	0, 1		
DEBUG_LEVEL	*U_INT32*	Debug level of device driver	see *dbg.h*		
DEBUG_LEVEL_MK	*U_INT32*	Debug level of MDIS kernel			
DEBUG_LEVEL_DESC	*U_INT32*	Debug level of descriptor decoder			
DEBUG_LEVEL_MBUF	*U_INT32*	Debug level of buffer manager			
DEBUG_LEVEL_OSS	*U_INT32*	Debug level of system services			
SUBDEVICE_OFFSET_x	*U_INT32*	Subdevice offset	0..0xFFFFFFFF		

Legend

Req:	required configuration keys, **must** be defined
Fix:	fixed keys, should not be changed by users

See also Chapter B 4.5 Driver Debugging on page 124.

DESC_TYPE defines the type of the descriptor object. For device descriptors the value 1 must be used.

HW_TYPE specifies the name of the device. From this name the matching device driver name is built internally.

DEVICE_SLOT defines at which slot on the board the device is mounted. Slot numbers begin with 0.

BOARD_NAME defines the name of the board descriptor.

ID_CHECK decides if the device ID PROM is read and checked at initialization.

IRQ_ENABLE specifies if the interrupt should implicitly be enabled during initialization.

DEBUG_LEVEL_xxx enable debug output from the driver functions. This only takes effect on debug drivers (see Chapter B 4.5 Driver Debugging on page 124).

SUBDEVICE_OFFSET_x can be used to create multiple identical sub-devices within an MDIS device. For example, the M51 M-Module (CAN bus) has four identical CAN controllers. You can now create MDIS devices for each CAN controller by specifying the offsets to the controller base. On M51 the controllers have an offset of 0x40 bytes to each other, so for the second CAN chip, one would specify *SUBDEVICE_OFFSET_x = 0x40*.

Further device-specific keys are described in the respective device driver user manual.

Example (M-Module)

```
M31_1  {
   DESC_TYPE     = U_INT32   1           # descriptor type (1=device)
   HW_TYPE       = STRING    M031        # hardware name of device

   BOARD_NAME    = STRING    A201_1      # device name of base board
   DEVICE_SLOT   = U_INT32   0           # used slot on base board (0..n)

   DEBUG_LEVEL   = U_INT32   0xc0008007  # LL driver debug level

   ID_CHECK      = U_INT32   1           # check module ID PROM
}
```

B 4.3.1 Additional Descriptor Keys for PCI Devices (PC•MIP Modules)

All PCI devices are supported by the generic PCI BBIS board driver in MDIS. The device descriptor of a PCI device must contain additional parameters in order to allow MDIS to check if you are accessing the right device.

Table B23. *Additional PCI Device Descriptor Keys*

Key	DataType	Description	Value Range	Req	Fix
PCI_VENDOR_ID	*U_INT32*	Vendor ID of the device in PCI configuration header	0x0000..0xFFFF	•	
PCI_DEVICE_ID	*U_INT32*	Device ID of the device in PCI configuration header	0x0000..0xFFFF	•	
PCI_SUBSYS_VENDOR_ID	*U_INT32*	Subsystem vendor ID in PCI configuration header	0x0000..0xFFFF		
PCI_SUBSYS_ID	*U_INT32*	Subsystem ID in PCI configuration header	0x0000..0xFFFF		
PCI_FUNCTION	*U_INT32*	PCI function number to use on device	0..7		
PCI_BASEREG_ASSIGN_x	*U_INT32*	Mapping between low-level driver's address spaces and PCI base address registers	0..5		

Legend
Req: required configuration keys, **must** be defined
Fix: fixed keys, should not be changed by users

PCI_VENDOR_ID and ***PCI_DEVICE_ID*** must match the values in the device's PCI configuration header.

PCI_SUBSYS_VENDOR_ID and ***PCI_SUBSYS_ID*** - if specified - must match the corresponding fields in the device's PCI configuration header.

PCI_FUNCTION defines the PCI function number to use within the devices. A PCI device can have up to 7 subfunctions. If this key is not present, the first function (number 0) is used.

PCI_BASEREG_ASSIGN_x: A low-level driver may request multiple address spaces from the MDIS kernel. This key defines the mapping between the low-level driver's address spaces and the PCI base address register 0..5. For example to use the PCI base address register #3 for the low-level driver's first address space set *PCI_BASEREG_ASSIGN_0 = U_INT32 3*. See the low-level driver documentation for further details.
If this key is not defined for the corresponding low-level driver's address space, a 1:1 mapping is used (i.e. PCI base address register #0 is used for the low-level driver's first address space).

B 4.4 Board Descriptor Keys

The board descriptor defines the following parameters:

- Name of the hardware
- Address or location on the bus system
- Further board-specific parameters
- Further bus-specific parameters

For these standard parameters the following descriptor keys are defined:

Table B24. *Board Descriptor Keys*

Key	DataType	Description	Value Range	Req	Fix
DESC_TYPE	*U_INT32*	Descriptor type 1 = device, 2 = board	2	•	•
HW_TYPE	*STRING*	Hardware type (name of the board)	[A..Z, 0..9, _]	•	•
DEBUG_LEVEL	*U_INT32*	Debug level for the board handler functions	0..0xFFFFFFFF		
DEBUG_LEVEL_BK	*U_INT32*	Debug level of BBIS kernel			
DEBUG_LEVEL_DESC	*U_INT32*	Debug level of descriptor decoder			
DEBUG_LEVEL_OSS	*U_INT32*	Debug level of system services			

Legend

Req:	required configuration keys, **must** be defined
Fix:	fixed keys, should not be changed by users

See also Chapter B 4.5 Driver Debugging on page 124.

DESC_TYPE defines the type of the descriptor object. For board descriptors the value 2 must be used.

HW_TYPE specifies the name of the base board. From this name the matching board handler name is built internally.

DEBUG_LEVEL enables debug output from the board handler functions. This only takes effect on debug board handlers (see Chapter B 4.5 Driver Debugging on page 124).

B 4.4.1 VMEbus M-Module Carrier Boards

The following additional descriptor keys are defined for VMEbus carrier boards such as the MEN A201S, B201S, B202S:

Table B25. *Special Keys for VMEbus M-Module Carrier Boards*

Key	DataType	Description	Value Range	Req	Fix
VME_A16_ADDR	*U_INT32*	Board base address in VME short (A16) space	`0x0000..0xFFFF`	(•)	
VME_A24_ADDR	*U_INT32*	Board base address in VME standard (A24) space	`0x000000..0xFFFFFF`	(•)	
PHYS_ADDR	*U_INT32*	Physical board address as seen from the CPU (optional), over-rides *VME_Axx_ADDR*	`0x00000000..0xFFFFFFFF`	(•)	
VME_DATA_WIDTH	*U_INT32*	1 = D16, 3 = D32	1 or 3	•	
IRQ_VECTOR	*BINARY*	IRQ vector for each slot	0 = none, 1..255	•	
IRQ_LEVEL	*BINARY*	IRQ level for each slot	1..6	•	
IRQ_PRIORITY	*BINARY*	IRQ priority	system-dependent		

Legend

Req:	required configuration keys, **must** be defined
Fix:	fixed keys, should not be changed by users

VME_Axx_ADDR defines the relative bus address(es) of the board in the specified A16, A24 VMEbus address space. The defined address must be equal to the configured address on the base board, i.e. the DIL or hex switches on the hardware. For the A201S, B201S and B202S boards, you must define either *VME_A16_ADDR* or *VME_A24_ADDR*, or *PHYS_ADDR*.

PHYS_ADDR is an optional key for addressing. When this key is defined instead of a *VME_Axx_ADDR* key, the board handler does not use the CPU board specific bus-to-physical-address function for mapping the given address into the CPU's address space. Instead the defined physical address is used as it is. This key must be used for systems where no such mapping functions exist. If this is the case, an *ERR_OSS_NO_PHYSTOBUS* error is returned at board initialization. The address defines the physical base address of the base board in the CPU's VMEbus address space.

VME_DATA_WIDTH key defines the VMEbus data access type and must not exceed the VME backplane's capabilities. It must be '1' (D16) for A201S, B201S and B202S.

IRQ_VECTOR defines an array of interrupt vectors to be used for each slot. It is recommended that you use an exclusive interrupt for each slot. Please check your system configuration to see which vectors are already occupied by other drivers.

IRQ_LEVEL defines the interrupt level on the VMEbus on which the board interrupts for each slot. The VMEbus has seven interrupt lines, IRQ level 7 has the highest priority, level 1 the lowest. Do not use level 7, because this is a non-maskable interrupt. Also check that your CPU board is able to receive an interrupt on the selected level. For example, MEN A8/A9/A10 boards only allow to receive VME interrupt levels 2, 3 and 5. Depending on your operating system you must also enable the corresponding level in the CPU's setup.

IRQ_PRIORITY is only used for shared interrupts. Some systems (OS-9) maintain a linked list of handlers in a priority-based manner. However, since VME interrupt sharing is not recommended, this key is not required.

B 4.4.2 CompactPCI M-Module Carrier Boards

The following describes how to configure the D201, F201 and F202 CompactPCI M-Module carrier boards.

First you must tell MDIS the PCI bus number of the CompactPCI backplane. Since this depends on the CPU and may even vary depending on the system configuration, it is recommended to configure a *PCI_BUS_PATH* in the descriptor. The PCI bus path allows to address an exact geographical location within a PCI system, indepedently of which PCI devices are present in the system. *PCI_BUS_PATH* is an array of device IDs of PCI-to-PCI bridges starting from PCI bus 0.

For example, to address the Compact PCI backplane on a MEN D2 CPU, you must enter *PCI_BUS_PATH = BINARY 0x08,* because the CompactPCI bridge has device ID `0x08` on PCI bus 0. See *D201/DOC/pcibuspath.txt* for a list of currently known bus paths.

If you don't know the device IDs of your system's bridges, you can alternatively enter the PCI bus number of the CompactPCI backplane directly using *PCI_BUS_NUMBER.*

Note: With VxWorks you can use the *pciscanner* tool to find out the PCI bus path (*sysPciScan()*).

Table B26. *Special Keys for CompactPCI M-Module Carrier Boards - PCI Bus Keys*

Key	DataType	Description	Value Range	Req	Fix
PCI_BUS_PATH	*BINARY*	Device IDs of bridges to CompactPCI backplane (see text)	0x00..0x1F, ...	(•)	
PCI_BUS_NUMBER	*U_INT32*	Alternative to *PCI_BUS_PATH*. Specify PCI bus number directly.	0x00..0xFF	(•)	

Legend
Req: required configuration keys, **must** be defined
Fix: fixed keys, should not be changed by users

Additionally, you must tell MDIS the CompactPCI slot in which your carrier board is plugged. Again you have two alternatives:

If you have a standard CompactPCI backplane you should use *PCI_BUS_SLOT.* This is the geographical slot number within the CompactPCI rack (slot 1 = system slot). MDIS will compute the PCI device number from this value. Some CompactPCI racks support encoding on each slot connector (signals GA[0..4]) that allows a board to check its geographical location. You can advice your board driver to check if *PCI_BUS_SLOT* and the actual geographical location match using descriptor key *PCI_CHECK_LOCATION.*

Alternatively you can specify the PCI device number on the CompactPCI bus directly using *PCI_DEVICE_ID* (note that this key exists also in the device descriptor of PCI devices but has a different meaning there).

Table B27. *Special Keys for CompactPCI M-Module Carrier Boards - PCI Device Keys*

Key	DataType	Description	Value Range	Req	Fix
PCI_BUS_SLOT	*U_INT32*	CompactPCI bus slot (1 = system slot)	2..n	(•)	
PCI_DEVICE_ID	*U_INT32*	PCI device number of board on CompactPCI bus, overrides *PCI_BUS_SLOT*	`0x00..0x1F`	(•)	
PCI_CHECK_LOCATION	*U_INT32*	0 = don't check location 1 = check location If key not present, defaults to 1!	0,1		

Legend
Req: required configuration keys, **must** be defined
Fix: fixed keys, should not be changed by users

B 4.4.3 Standard PCI M-Module Carrier Boards

The M-Module carrier boards for standard PCI bus, C203 and C204, are handled by a variant of the D201 board driver. As opposed to CompactPCI, the PCI bus in desktop PCs usually has always PCI bus number 0, so you do not need to specify a *PCI_BUS_PATH*. On the other hand the PCI slots have no standard numbering of device IDs. So you need to find out the device number of the C203/C204 board using a PCI viewer utility and enter this value in *PCI_DEVICE_ID*.

Table B28. *Special Keys for Standard PCI M-Module Carrier Boards*

Key	DataType	Description	Value Range	Req	Fix
PCI_BUS_NUMBER	*U_INT32*	Specify PCI bus number (always 0 in desktop PCs)	0	•	
PCI_DEVICE_ID	*U_INT32*	PCI device number of board on PCI bus 0	`0x00..0x1F`	•	
PCI_CHECK_LOCATION	*U_INT32*	Must be 0, since GA[0..4] not available on standard PCI bus	0	•	

Legend
Req: required configuration keys, **must** be defined
Fix: fixed keys, should not be changed by users

B 4.4.4 PC•MIP Carrier Boards

For PC•MIP modules, the F203 and D202 boards are available for CompactPCI systems. Additionally PC•MIP modules can reside directly on the on-board PC•MIP slots of the MEN CPUs D2, F1, F2 and A11.

All of these boards are handled by the generic PCI board handler. Again, a *PCI_BUS_PATH* key should be present in the descriptor to allow an exact geographical addressing of a specific PC•MIP module (see also Chapter B 4.4.2 CompactPCI M-Module Carrier Boards on page 120). As opposed to the M-Module carriers, you must include the device number of the CompactPCI backplane in the *PCI_BUS_PATH*. For example, to address a D202 carrier board in CompactPCI slot #3 of a D2 system, enter *PCI_BUS_PATH = BINARY 0x08,0x0e*, because the CompactPCI bridge on the D2 has device ID `0x08` on PCI bus 0, and slot #3 has device ID `0x0E` on the CompactPCI bus.

A carrier board usually has multiple PC•MIP slots. The device numbers of each PC•MIP slot must be entered in the descriptor keys *DEVICE_SLOT_x*.

Table B29. *Special Keys for PC•MIP Carrier Boards*

Key	DataType	Description	Value Range	Req	Fix
PCI_BUS_PATH	*BINARY*	Device IDs of bridges to carrier board	`0x00..0x1F`, ...	(•)	
PCI_BUS_NUMBER	*U_INT32*	Optionally overrides *PCI_BUS_PATH*	`0x00..0xFF`	(•)	
DEVICE_SLOT_x	*U_INT32*	Specify the PCI device number of PC•MIP slot *n* on the carrier board	`0x00..0x1F`	•	

Legend
Req: required configuration keys, **must** be defined
Fix: fixed keys, should not be changed by users

The following table lists the *PCI_BUS_PATH*s on some CompactPCI carriers and CPUs.

Table B30. *PCI_BUS_PATH Values on MEN CompactPCI CPUs and PC•MIP Carrier Boards*

CPU/Carrier Board (Slot 1)	***PCI_BUS_PATH*** **Values for PC•MIP Carrier Board in Compact PCI Slot**					
	2	**3**	**4**	**5**	**6**	**7**
D1	`0x14,0x0F`	`0x14,0x0E`	`0x14,0x0D`	`0x14,0x0C`	`0x14,0x0B`	`0x14,0x0A`
D2	`0x08,0x0F`	`0x08,0x0E`	`0x08,0x0D`	`0x08,0x0C`	`0x08,0x0B`	`0x08,0x0A`
F1, D3	`0x1E,0x0F`	`0x1E,0x0E`	`0x1E,0x0D`	`0x1E,0x0C`	`0x1E,0x0B`	`0x1E,0x0A`
F2	`0x08,0x0F`	`0x08,0x0E`	`0x08,0x0D`	`0x08,0x0C`	`0x08,0x0B`	`0x08,0x0A`
F3	`0x1F`	`0x1E`	`0x1D`	`0x1C`	`0x1B`	`0x1A`
F7	`0x1E,0x06, 0x0F`	`0x1E,0x06, 0x0E`	`0x1E,0x06, 0x0D`	`0x1E,0x06, 0x0C`	`0x1E,0x06, 0x0B`	`0x1E,0x06, 0x0A`

The following table lists the *DEVICE_SLOT_x* assignments for some carrier boards:

Table B31. *DEVICE_SLOT_x for PC•MIP Slots on MEN PC•MIP Carrier Boards*

CPU/Carrier Board	***DEVICE_SLOT_x* for PC•MIP Slot**					
	0 (A)	**1 (B)**	**2 (C)**	**3 (D)**	**4 (E)**	**5 (F)**
A11	`0x00`	`0x01`				
D2	`0x0C`	`0x0B`	-	-	`0x0E`	
D202	`0x0C`	`0x0D`	`0x0E`	`0x0F`	`0x0A`	`0x0B`
F1, B11	`0x1A`	`0x1D`				
F2	`0x0D`					
F203 Note 1)	`0x0F`	`0x0E`	`0x0D`			
A12a, D3a, SC13a	`0x00`	`0x01`	`0x02`			
A12c, D3c, SC13c (PMC slots)	`0x03`	`0x04`				

Note: The labels of slot A and B are swapped on Rev. 00 of the F203, compared to later revisions.

Example of a D202 Base Board Descriptor

The D202 resides in Compact PCI slot #3 of a D2 system.

```
D202_1 {

    DESC_TYPE       = U_INT32  2         # descriptor type (2=board)
    HW_TYPE         = STRING   PCI       # hardware name of device

    PCI_BUS_PATH   = BINARY   0x08,0x0e
    DEVICE_SLOT_0  = U_INT32  0x0C
    DEVICE_SLOT_1  = U_INT32  0x0D
    DEVICE_SLOT_2  = U_INT32  0x0E
    DEVICE_SLOT_3  = U_INT32  0x0F
    DEVICE_SLOT_4  = U_INT32  0x0A
    DEVICE_SLOT_5  = U_INT32  0x0B
}
```

B 4.5 Driver Debugging

All MDIS driver modules (Kernel, low-level driver, board handler etc.) can be compiled as debug drivers for problem fixing purposes.

The debug messages produced by a debug driver can be scaled via descriptor entries or via SetStat calls at runtime by defining a "debug level" for a specific module or functionality:

Note: Refer to the operating system specific part of the MDIS User Guide for details about viewing the debug messages.

B 4.5.1 Debug Level

The debug level is a 32-bit word containing several flags:

31	30	29..16	15	14..3	2	1	0
INTR enable	NORM enable	-	ERROR enable	-	LEV3 enable	LEV2 enable	LEV1 enable

INTR enable	Enables debug output within interrupt service routine
NORM enable	Enables debug output for all other (non-interrupt) routines
ERROR enable	Enables debug output for error messages and warnings
LEV1 enable	Enables debug output level 1 (function names)
LEV2 enable	Enables debug output level 2 (additional infos)
LEV3 enable	Enables debug output level 2 (verbose)

Examples

• `0xC0008000`	Only error messages and warnings
• `0xC0008001`	Only function names
• `0xC0008002`	Only additional infos
• `0xC0008007`	All messages

(See also defintions in *dbg.h*.)

B 4.5.2 Debug Settings

The debug level can be defined for the following modules and functionality:

Table B32. *Debug Level Definitions*

Function(s)	Descriptor Key	Status Call	Recommended
MDIS API Calls	-	*M_MK_API_DEBUG_LEVEL*	`0x00000000`
MDIS Kernel	*DEBUG_LEVEL_MK (D)*	*M_MK_DEBUG_LEVEL*	`0xC0008000`
Device Driver	*DEBUG_LEVEL (D)*	*M_LL_DEBUG_LEVEL*	`0xC0008007`
BBIS Kernel	*DEBUG_LEVEL_BK (B)*	*M_MK_BK_DEBUG_LEVEL*	`0xC0008000`
Board Handler	*DEBUG_LEVEL (B)*	*M_BB_DEBUG_LEVEL*	`0xC0008000`
Descriptor Decoder Calls	*DEBUG_LEVEL_DESC*	-	`0x80008002`
System Calls	*DEBUG_LEVEL_OSS*	*M_MK_OSS_DEBUG_LEVEL*	`0xC0008002`
Buffer Manager Calls (In)	*DEBUG_LEVEL_MBUF*	*M_BUF_RD_DEBUG_LEVEL*	`0xC0008000`
Buffer Manager Calls (Out)	*DEBUG_LEVEL_MBUF*	*M_BUF_WR_DEBUG_LEVEL*	`0xC0008000`

Legend
D = device descriptor
B = board descriptor