# Table of contents

1 Foreword ................................................................................................................................................. 5
   1.1 Notes on the documentation ............................................................................................................... 5
   1.2 Safety instructions ............................................................................................................................... 6

2 Introduction .................................................................................................................................................. 7

3 MDP element access options .................................................................................................................... 8

4 Function blocks .......................................................................................................................................... 9
   4.1 Introduction ........................................................................................................................................ 9
   4.2 Generic ................................................................................................................................................ 10
      4.2.1 Advanced ..................................................................................................................................... 10
      4.2.2 FB_MDP_ReadElement ................................................................................................................ 13
      4.2.3 FB_MDP_ReadModule .................................................................................................................. 14
      4.2.4 FB_MDP_ReadModuleContent ..................................................................................................... 16
      4.2.5 FB_MDP_ReadModuleHeader ....................................................................................................... 17
      4.2.6 FB_MDP_ScanModules ................................................................................................................ 18
      4.2.7 FB_MDP_SplitErrorId .................................................................................................................. 19
   4.3 Specific ............................................................................................................................................... 19
      4.3.1 FB_MDP_CPU_Read .................................................................................................................... 19
      4.3.2 FB_MDP_Device_Read_DevName ................................................................................................ 20
      4.3.3 FB_MDP_IdentityObj_Read ......................................................................................................... 21
      4.3.4 FB_MDP_NIC_Read ...................................................................................................................... 22
      4.3.5 FB_MDP_NIC_Write_IP ................................................................................................................ 23
      4.3.6 FB_MDP_SiliconDrive_Read ......................................................................................................... 24
      4.3.7 FB_MDP_SW_Read_MdpVersion .................................................................................................. 25
      4.3.8 FB_MDP_TwinCAT_Read ............................................................................................................. 26

5 Data types .................................................................................................................................................. 28
   5.1 General data types ............................................................................................................................... 28
      5.1.1 E_MDP_AddrArea ....................................................................................................................... 28
      5.1.2 E_MDP_ModuleType ................................................................................................................... 28
      5.1.3 ST_MDP_Addr ............................................................................................................................. 28
      5.1.4 ST_MDP_ModuleHeader .............................................................................................................. 29
   5.2 Structures specific MDP modules ......................................................................................................... 29
      5.2.1 ST_MDP_CPU ............................................................................................................................. 29
      5.2.2 ST_MDP_IdentityObject .............................................................................................................. 30
      5.2.3 ST_MDP_NIC_Properties ............................................................................................................ 30
      5.2.4 ST_MDP_SiliconDrive ................................................................................................................. 30
      5.2.5 ST_MDP_TwinCAT ..................................................................................................................... 31

6 Global Constants ........................................................................................................................................ 32
   6.1 Global_Version .................................................................................................................................... 32

7 Error Codes ............................................................................................................................................... 33
   7.1 Error codes overview .......................................................................................................................... 33
   7.2 E_MDP_ErrGroup ............................................................................................................................... 33
   7.3 E_MDP_ErrCodesPLC ......................................................................................................................... 34
## 8 Samples

- **8.1 Querying CPU data (generic)**                                                                                      36
- **8.2 Querying CPU data (specific)**                                                                                   45
- **8.3 Querying the fan state (generic)**                                                                             49
- **8.4 Reading IPC serial numbers**                                                                                     51
- **8.5 Setting the IP address**                                                                                         53
1 Foreword

1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with applicable national standards.
It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.
It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the requirements for safety, including all the relevant laws, regulations, guidelines and standards.

Disclaimer

The documentation has been prepared with care. The products described are, however, constantly under development.
We reserve the right to revise and change the documentation at any time and without prior announcement.
No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.
Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:
with corresponding applications or registrations in various other countries.

EtherCAT® is a registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany.

Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.
The reproduction, distribution and utilization of this document as well as the communication of its contents to others without express authorization are prohibited.
Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.
1.2 Safety instructions

Safety regulations

Please note the following safety instructions and explanations! Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

Exclusion of liability

All the components are supplied in particular hardware and software configurations appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

Personnel qualification

This description is only intended for trained specialists in control, automation and drive engineering who are familiar with the applicable national standards.

Description of symbols

In this documentation the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

⚠️ **DANGER**

Serious risk of injury!
Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.

⚠️ **WARNING**

Risk of injury!
Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.

⚠️ **CAUTION**

Personal injuries!
Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.

⚠️ **NOTE**

Damage to the environment or devices
Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.

⚠️ **Tip or pointer**

This symbol indicates information that contributes to better understanding.
Introduction

Update: Tc3_IPCDiag library

The TwinCAT 3 PLC library Tc2_MDP is the predecessor to Tc3_IPCDiag. With the new Tc3_IPCDiag library the number of readable parameters has been increased and the user interface has been optimized. It is recommended to use the Tc3_IPCDiag library.

Future extensions will no longer be performed in the Tc2_MDP library. It is not recommended to use the Tc2_MDP library for new projects. All functionalities of the Tc2_MDP library can also be found in the new Tc3_IPCDiag library.

The TwinCAT 3 PLC library Tc2_MDP is used by the Beckhoff IPC diagnostics. Details can be found in the documentation: Beckhoff Device Manager.

The available IPC diagnosis data are organized in the configuration area in so-called "Modules". A module contains all the data for a particular topic. The example applies to the IPC CPU.

A module can in turn contain subcategories, so-called "Tables". A table organizes the detailed information that it contains in so-called "subindices". Since the contents of the list depend on the components existing in the current IPC, the list is generated dynamically – depending on what components the current PC contains, or what types of information it supports.

Example: Access to data on the mainboard requires a BIOS (and the corresponding hardware in the PC) that can supply these data.

A module therefore cannot be addressed via a fixed address. You first have to determine where exactly the module is located.

Restricted access at the time of system start

MDP forms an interface to the hardware. This is independent of TwinCAT. MDP can be accessed from TwinCAT with the PLC library. This is done internally by means of ADS communication. The versatility of the hardware configuration justifies a different initialization phase of the MDP service. It is possible that first PLC cycles are executed while the MDP initialization is not yet completed. Either the possible error outputs as well as timeouts of the function blocks from the library can be reacted and a new query can be triggered or the queries are executed deliberately delayed after the system start.

It is recommended in the PLC program not to query values from the MDP immediately after the system start, but to consider a small waiting time. How large this should be depends on various parameters (such as the performance of your control computer), and can therefore not be given as a general rule. Typically it is in the range of 10-60 seconds.
### 3 MDP element access options

The TwinCAT 3 PLC Tc2_MDP library offers a wide range of function blocks to facilitate comprehensive access to MDP data.

There are two basic types of function blocks in the library. Firstly the **generic function blocks**. They can be used to query and set arbitrary parameters in the MDP themselves by means of discrete access. Furthermore, **specific function blocks** offer the possibility of accessing certain data as well as groupings of several data with one call.

The function blocks have a uniform appearance. All function blocks are called by a positive edge on the `bExecute` input. Afterwards, cyclic calling of the function block (`bExecute = FALSE`) returns the result of the query at the output as soon as the processing of the query has been completed (`bBusy = FALSE`). Each function block must be called (`bExecute = FALSE`) for as long as it takes for the internal processing (`bBusy = FALSE`) to be completed. During that time, all inputs of the function block must remain unchanged.

In general, the MDP is a model that describes hardware and software components in the form of modules. Information about these modules as well as about the device itself can be queried and changed. A module consists of one or more tables. Each table consists of a fixed number of subindices. A subindex corresponds to a concrete element that can be accessed.

You can find more information on the setup of MDP in the **MDP Information Model** (Device Manager documentation). Further options for accessing the MDP are also described there.

**Generic function blocks**

In order to be able to query or set an IPC diagnostics parameter, the dynamic module ID of the module in which the parameter is located must be known. This is determined with the aid of the function block `FB_MDP_ScanModules`.

Individual parameters can now be read or written by means of `FB_MDP_Read` and `FB_MDP_Write`. In addition to the dynamic Module ID, the number of the selected table (Table ID), the selected subindex within the table as well as further information is thereby specified for the query.

Likewise, the complete header of a module (`ST_MDP_ModuleHeader`) can be queried with the function block `FB_MDP_ReadModuleHeader`. The complete contents of a selected table within a module can be queried with the function block `FB_MDP_ReadModuleContent`.

The function block `FB_MDP_ReadModule` bundles the above queries. The function block implicitly determines the dynamic Module ID and queries both header and table. The function block `FB_MDP_ReadElement` also determines the dynamic module ID implicitly. It can be used to query any individual IPC diagnostics parameter. Accordingly, with these two function blocks it is not necessary to call `FB_MDP_ScanModules` beforehand.

**Specific function blocks**

The function blocks available here offer fast access to the most important IPC diagnostic information.

For example, it is sufficient to call the function block `FB_MDP_NIC_Read` in order to query all important information about a Network Interface Card (see Device Manager documentation `Module NIC`). The module header is also queried and output in each case. The specific function blocks likewise implicitly determine the dynamic Module ID, so that a prior call of `FB_MDP_ScanModules` is superfluous.
4 Function blocks

4.1 Introduction

Generic function blocks
A generic function block can be used to access any IPC diagnostics modules.

The application is more complex and the user requires a certain understanding of the MDP (Module Device Profile) information model.

Sample: Accessing network card information via FB_MDP_ReadElement:

The generic function block FB_MDP_ReadElement determines the dynamic module address internally. The user only needs to specify the module instance that he wishes to access. The system has several network cards and each network card is represented by its own module instance.

Specific function blocks
A specific function block only accesses information from a specific module. It is simple to use and requires no knowledge of the MDP information model used.

The specific function blocks available determine the dynamic module address internally.

However, specific function blocks are only available for a selection of the IPC diagnostics data.

Sample: FB_MDP_CPU_Read
4.2  Generic

4.2.1  Advanced

4.2.1.1  FB_MDP_Read

<table>
<thead>
<tr>
<th>Function block</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VAR_INPUT</strong></td>
<td></td>
</tr>
<tr>
<td>bExecute</td>
<td>The function block is called by a rising edge on the input <em>bExecute</em>, if the block is not already active.</td>
</tr>
<tr>
<td>stMDP_DynAddr</td>
<td>The MDP addressing belonging to the selected network module is specified at this input. The structure is of the type <code>ST_MDP_Addr</code> [28]. The dynamic Module ID must already be specified with it.</td>
</tr>
<tr>
<td>pDstBuf</td>
<td>The memory address of the data buffer is specified at this input. The received data are stored there if the query is successful.</td>
</tr>
<tr>
<td>cbDstBufLen</td>
<td>The length of the data buffer in bytes is specified at this input.</td>
</tr>
<tr>
<td>tTimeout</td>
<td>Specifies a maximum length of time for the execution of the function block.</td>
</tr>
<tr>
<td>sAmsNetId</td>
<td>To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type <code>T_AmsNetId</code>) can be specified here.</td>
</tr>
<tr>
<td><strong>VAR_OUTPUT</strong></td>
<td></td>
</tr>
<tr>
<td>bBusy</td>
<td>This output is TRUE as long as the function block is active.</td>
</tr>
<tr>
<td>bError</td>
<td>Becomes TRUE as soon as an error situation occurs.</td>
</tr>
<tr>
<td>nErrId</td>
<td>Returns an error code [33] if the bError output is set.</td>
</tr>
<tr>
<td>nCount</td>
<td>This output indicates the number of bytes read.</td>
</tr>
</tbody>
</table>

The function block enables querying of an IPC diagnostics module element.
Function blocks

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

4.2.1.2 **FB_MDP_Write**

The function block enables setting of an IPC diagnostics module element.

**VAR_INPUT**

<table>
<thead>
<tr>
<th>bExecute</th>
<th>stMDP_DynAddr</th>
<th>pSrcBuf</th>
<th>cbSrcBufLen</th>
<th>tTimeout</th>
<th>sAmsNetId</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_INPUT: BOOL; (* Function block execution is triggered by a rising edge at this input.*)</td>
<td>VAR_INPUT: ST_MDP_Addr;</td>
<td>VAR_INPUT: DWORD; (* Contains the address of the buffer for the sent data.*)</td>
<td>VAR_INPUT: UDINT; (* Contains the max. number of bytes to be sent.*)</td>
<td>VAR_INPUT: TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled.*)</td>
<td>VAR_INPUT: T_AmsNetId; (* keep empty '' for the local device *)</td>
</tr>
</tbody>
</table>

**bExecute**: The function block is called by a rising edge on the input `bExecute`, if the block is not already active.

**stMDP_DynAddr**: The MDP addressing belonging to the selected network module is specified at this input. The structure is of the type `ST_MDP_Addr`. The dynamic Module ID must already be specified with it.

**pSrcBuf**: The memory address of the data buffer is specified at this input. The data to be transmitted must be stored there.

**cbSrcBufLen**: The length of the data buffer in bytes is specified at this input.

**tTimeout**: Specifies a maximum length of time for the execution of the function block.

**sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type `T_AmsNetId`) can be specified here.

**VAR_OUTPUT**

<table>
<thead>
<tr>
<th>bBusy</th>
<th>bError</th>
<th>nErrId</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_OUTPUT: BOOL;</td>
<td>VAR_OUTPUT: BOOL;</td>
<td>VAR_OUTPUT: UDINT;</td>
</tr>
</tbody>
</table>

**bBusy**: This output is TRUE as long as the function block is active.

**bError**: Becomes TRUE as soon as an error situation occurs.

**nErrId**: Returns an error code [33] if the `bError` output is set.

Requirements
4.2.1.3   FB_MDP_ReadIndex

The function block enables querying of any IPC diagnostics element. In addition to the configuration area, data from the device area are also accessible.

VAR_INPUT

<table>
<thead>
<tr>
<th>VAR_INPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>bExecute  : BOOL; (* Function block execution is triggered by a rising edge at this input.*)</td>
</tr>
<tr>
<td>nIndex    : WORD;</td>
</tr>
<tr>
<td>nSubIndex : BYTE;</td>
</tr>
<tr>
<td>pDstBuf   : DWORD; (* Contains the address of the buffer for the received data.*)</td>
</tr>
<tr>
<td>cbDstBufLen : UDINT; (* Contains the max. number of bytes to be received.*)</td>
</tr>
<tr>
<td>tTimeout  : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled.*)</td>
</tr>
<tr>
<td>sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)</td>
</tr>
</tbody>
</table>

END_VAR

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

nIndex: At this input the first part of the addressing for the required IPC diagnostic data is specified.

nSubIndex: At this input the second part of the addressing for the required IPC diagnostic data is specified.

pDstBuf: The memory address of the data buffer is specified at this input. The received data are stored there if the query is successful.

cbDstBufLen: The length of the data buffer in bytes is specified at this input.

tTimeout: Specifies a maximum length of time for the execution of the function block.

sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

VAR_OUTPUT

<table>
<thead>
<tr>
<th>VAR_OUTPUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>bBusy     : BOOL;</td>
</tr>
<tr>
<td>bError    : BOOL;</td>
</tr>
<tr>
<td>nErrId    : UDINT;</td>
</tr>
<tr>
<td>nCount    : UDINT;</td>
</tr>
</tbody>
</table>

END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrId: Returns an error code [33] if the bError output is set.

nCount: This output indicates the number of bytes read.

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.4013</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>
4.2.2 FB_MDP_ReadElement

The function block enables querying of an individual MDP element. In this way, each element from each module of the configuration area can be read!

Internally, the device is scanned for the selected module, and the element information is queried with the dynamic module ID.

**VAR_INPUT**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bExecute</td>
<td>BOOL</td>
<td>The function block is called by a rising edge on the input bExecute, if the block is not already active.</td>
</tr>
<tr>
<td>stMDP_Addr</td>
<td>ST_MDP_Addr</td>
<td>The MDP addressing belonging to the selected module is specified at this input. The structure is of the type ST_MDP_Addr. The area must be specified as configuration area.</td>
</tr>
<tr>
<td>eModuleType</td>
<td>E_MDP_ModuleType</td>
<td>(* chosen module type out of the module type list *)</td>
</tr>
<tr>
<td>iModIdx</td>
<td>USINT</td>
<td>(* chosen index(0..n) of the demanded module type. E.g. second NIC(idx 1) of three found NICs. *)</td>
</tr>
<tr>
<td>pDstBuf</td>
<td>DWORD</td>
<td>(* Contains the address of the buffer for the received data. *)</td>
</tr>
<tr>
<td>cbDstBufLen</td>
<td>UDINT</td>
<td>(* Contains the max. number of bytes to be received. *)</td>
</tr>
<tr>
<td>sAmsNetId</td>
<td>T_AmsNetId</td>
<td>(* keep empty '' for the local device *)</td>
</tr>
</tbody>
</table>

**VAR_OUTPUT**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bBusy</td>
<td>BOOL</td>
<td>(* indicates if Read was successfull or not *)</td>
</tr>
<tr>
<td>bError</td>
<td>BOOL</td>
<td></td>
</tr>
<tr>
<td>nErrID</td>
<td>UDINT</td>
<td></td>
</tr>
<tr>
<td>nCount</td>
<td>UDINT</td>
<td></td>
</tr>
<tr>
<td>stMDP_DynAddr</td>
<td>ST_MDP_Addr</td>
<td>(* includes the new dynamic module type id. *)</td>
</tr>
<tr>
<td>iModuleTypeCount</td>
<td>USINT</td>
<td>(* returns the number of found modules equal the demanded module type. *)</td>
</tr>
<tr>
<td>iModuleCount</td>
<td>USINT</td>
<td>(* returns the number of all detected MDP modules. *)</td>
</tr>
</tbody>
</table>
bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code \[33\] if the bError output is set.

nCount: This output indicates the number of bytes read.

stMDP_DynAddr: At this output the MDP addressing relating to the selected MDP module is specified. The structure is of the type ST_MDP_Addr \[28\]. The dynamic Module ID was added by the function block.

iModuleTypeCount: The output iModuleTypeCount indicates the number of modules that correspond to the specified type.

iModuleCount: The output iModuleCount indicates the entire number of modules on the device.

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### 4.2.3 FB_MDP_ReadModule

```plaintext
VAR_INPUT
bExecute : BOOL;  
stMDP_Addr : ST_MDP_Addr;  (* includes all address parameters without the Dynamic Module Id *)  
eModuleType : E_MDP_ModuleType;  (* chosen module type out of the module type list *)  
iModIdx : USINT;  (* chosen index(0..n) of the demanded module type. E.g. second NIC(idx 1) of three found NICs. *)  
iSubIdxCount : USINT;  
pDstBuf : DWORD;  (* Contains the address of the buffer for the received data. *)  
cbDstBufLen : UDINT;  (* Contains the max. number of bytes to be received. *)  
tTimeout : TIME := DEFAULT_ADS_TIMEOUT;  (* States the time before the function is cancelled. *)  
sAmsNetId : T_AmsNetId;  (* keep empty '' for the local device *)
END_VAR
```

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

stMDP_Addr: The MDP addressing belonging to the selected module is specified at this input. The structure is of the type ST_MDP_Addr \[28\]. The dynamic Module ID is only added internally.

eModuleType: The MDP module type is specified at this input. The possible types are listed in the enumeration E_MDP_ModuleType \[28\]. (General information: module types list)
iModIdx: If several instances of an MDP module exist, a selection can be made by means of the input iModIdx (0,...,n).

iSubIdxCount: The input iSubIdxCount is used to specify how many subindices of the selected Table ID are to be queried.

pDstBuf: The memory address of the data buffer is specified at this input. The received data are stored there if the query is successful.

cbDstBufLen: The length of the data buffer in bytes is specified at this input.

tTimeout: Specifies a maximum length of time for the execution of the function block.

sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

VAR_OUTPUT

VAR_OUTPUT

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>VAR_OUTPUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bBusy</td>
<td>BOOL; (* indicates if Read was successful or not *)</td>
<td></td>
</tr>
<tr>
<td>bError</td>
<td>BOOL;</td>
<td></td>
</tr>
<tr>
<td>nErrID</td>
<td>UDINT;</td>
<td></td>
</tr>
<tr>
<td>iErrPos</td>
<td>USINT;</td>
<td></td>
</tr>
<tr>
<td>iModuleTypeCount</td>
<td>USINT;</td>
<td></td>
</tr>
<tr>
<td>iModuleCount</td>
<td>USINT;</td>
<td></td>
</tr>
<tr>
<td>stMDP_DynAddr</td>
<td>ST_MDP_Addr;</td>
<td></td>
</tr>
<tr>
<td>stMDP_ModuleHeader</td>
<td>ST_MDP_ModuleHeader;</td>
<td></td>
</tr>
<tr>
<td>arrStartIdx</td>
<td>ARRAY[0..255] OF UINT;</td>
<td></td>
</tr>
</tbody>
</table>

END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code [33] if the bError output is set.

iErrPos: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

stMDP_DynAddr: At this output the MDP addressing relating to the selected MDP module is specified. The structure is of the type ST_MDP_Addr [28]. The dynamic Module ID was added by the function block.

iModuleTypeCount: The output iModuleTypeCount indicates the number of modules that correspond to the specified type.

iModuleCount: The output iModuleCount indicates the entire number of modules on the device.

stMDP_ModuleHeader: The header information from the read MDP module is displayed at this output in the form of the structure ST_MDP_ModuleHeader. [29]

arrStartIdx: This array describes how the individually queried subindices have been stored in the buffer. The array index zero indicates the position in bytes at which the data of subindex zero begins in the buffer. Subsequent subindices are handled analogously.

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>
4.2.4 **FB_MDP_ReadModuleContent**

<table>
<thead>
<tr>
<th>Function Block</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FB_MDP_ReadModuleContent</strong></td>
<td>bExecute</td>
</tr>
<tr>
<td></td>
<td>stMDP_DynAddr</td>
</tr>
<tr>
<td></td>
<td>iSubIdxCount</td>
</tr>
<tr>
<td></td>
<td>pDstBuf</td>
</tr>
<tr>
<td></td>
<td>cbDstBufLen</td>
</tr>
<tr>
<td></td>
<td>tTimeout</td>
</tr>
</tbody>
</table>

The function block enables querying of the content of an IPC diagnostics module.

**VAR_INPUT**

```
VAR_INPUT
  bExecute : BOOL;
  stMDP_DynAddr : ST_MDP_Addr; (* includes the dynamic module type for which the module content is requested. All subindexes of the chosen table are requested. *)
  iSubIdxCount : USINT; (* the number of SubIndexes to be requested *)
  pDstBuf : DWORD; (* Contains the address of the buffer for the received data. *)
  cbDstBufLen : DWORD; (* Contains the max. number of bytes to be received. *)
  tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
  sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```

**bExecute**: The function block is called by a rising edge on the input `bExecute`, if the block is not already active.

**stMDP_DynAddr**: The MDP addressing belonging to the selected module is specified at this input. The structure is of the type `ST_MDP_Addr` [28]. The dynamic Module ID must already be transferred with it.

**iSubIdxCount**: The input `iSubIdxCount` is used to specify how many subindices of the selected Table ID are to be queried.

**pDstBuf**: The memory address of the data buffer is specified at this input. The received data are stored there if the query is successful.

**cbDstBufLen**: The length of the data buffer in bytes is specified at this input.

**tTimeout**: Specifies a maximum length of time for the execution of the function block.

**sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type `T_AmsNetId`) can be specified here.

**VAR_OUTPUT**

```
VAR_OUTPUT
  bBusy : BOOL;
  bError : BOOL; (* indicates if Read was successfull or not *)
  nErrID : UDINT;
  iErrMsg : USINT;
  arrStartIdx : ARRAY[0..255] OF UINT; (* startindexes in bytes of each subindex element *)
END_VAR
```

**bBusy**: This output is TRUE as long as the function block is active.

**bError**: Becomes TRUE as soon as an error situation occurs.

**nErrID**: Returns an error code [33] if the `bError` output is set.

**iErrMsg**: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

**arrStartIdx**: This array describes how the individually queried subindices have been stored in the buffer. The array index zero indicates the position in bytes at which the data of subindex zero begins in the buffer. Subsequent subindices are handled analogously.
The function block enables querying of the header of an IPC diagnostics module.

**VAR_INPUT**

```
VAR_INPUT
  bExecute : BOOL;
  nDynModuleId : BYTE; (* the dynamic module id for which the module header is requested *)
  tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
  sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```

- **bExecute**: The function block is called by a rising edge on the input `bExecute`, if the block is not already active.
- **stMDP_DynAddr**: The MDP addressing belonging to the selected network module is specified at this input. The dynamic Module ID must already be specified with it.
- **tTimeout**: Specifies a maximum length of time for the execution of the function block.
- **sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

**VAR_OUTPUT**

```
VAR_OUTPUT
  bBusy : BOOL;
  bError : BOOL; (* indicates if Read was successful or not *)
  nErrID : UDINT;
  stMDP_ModHeader : ST_MDP_ModuleHeader;
END_VAR
```

- **bBusy**: This output is TRUE as long as the function block is active.
- **bError**: Becomes TRUE as soon as an error situation occurs.
- **nErrID**: Returns an error code [33] if the bError output is set.
- **stMDP_ModuleHeader**: At this output the header information for the read IPC diagnostics modules is displayed in the form of the structure `ST_MDP_ModuleHeader [29]`.

### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

---

**4.2.5 FB_MDP_ReadModuleHeader**

```
FUNCTION_BLOCK FB_MDP_ReadModuleHeader
  bExecute : BOOL;
  bBusy:
  nDynModuleId : BYTE;
  bError:
  tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
  nErrID:
  sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
  stMDP_ModHeader:
END_FUNCTION_BLOCK
```

---

**Requirements**

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>
4.2.6 FB_MDP_ScanModules

The function block enables searching of a device for a particular IPC diagnostics module. Selection can be made if several instances of the module type are present. The dynamic Module ID for the selected module type is determined by the function block. This is an important component of the MDP addressing, which is represented in the structure ST_MDP_Addr.

VAR_INPUT

VAR_INPUT

bExecute : BOOL; (* chosen module type out of the module type list *)
nModuleType : WORD; (* chosen index(0..n) of the demanded module type. E.g. second NIC(idx 1) of three found NICs.*)
iModIdx : USINT; (* selected module type out of the module type list *)
tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled.*)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)

END_VAR

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

nModuleType: At this input is the IPC diagnostics module type is specified. The possible types are listed in the enumeration E_MDP_ModuleType. (general information on IPC diagnostics module types.)

iModIdx: If several instances of an IPC diagnostics module exist, a selection can be made by means of the input iModIdx (0,...,n).

In the case of uncertainty concerning the selection: information about which module is explicitly concerned can be queried via the function block FB_MDP_ReadModuleHeader after scanning.

tTimeout: Specifies a maximum length of time for the execution of the function block.

sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

VAR_OUTPUT

VAR_OUTPUT

bBusy : BOOL; (* indicates if Scan was successfull or not *)
bError : BOOL; (* indicates if Scan was successfull or not *)
nErrID : UDINT; (* Dynamic Module Id *)
nDynModuleId : BYTE; (* Dynamic Module Id *)
iModuleTypeCount : USINT; (* returns the number of found modules equal the demanded module type. *)
iModuleCount : USINT; (* returns the number of all detected MDP modules. *)

END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code if the bError output is set.

nDynModuleId: This output indicates the dynamic Module ID determined for the selected module.

iModuleTypeCount: The output iModuleTypeCount indicates the number of modules that correspond to the specified type.

iModuleCount: The output iModuleCount indicates the entire number of modules on the device.
### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### 4.2.7 FB_MDP_SplitErrorId

The function block enables the `nErrID` to be split into an error group [33] and a specific error code. Accordingly, this function block can be referred to for the simplified evaluation of `nErrID`.

#### VAR_INPUT

```plaintext
VAR_INPUT
  nErrID : UDINT;
END_VAR
```

`nErrID`: `nErrID` is specified as an input on the function block. This 4-byte variable corresponds to the output `nErrID` on an MDP function block.

#### VAR_OUTPUT

```plaintext
VAR_OUTPUT
  eErrGroup : E_MDP_ErrGroup; (* type of transmitted error code *)
  nErrCode : UINT; (* error code [see specific error type table] *)
END_VAR
```

`eErrGroup`: The output `eErrGroup` corresponds to a value of the enumeration `E_MDP_ErrGroup` [33]. It is possible with the aid of the error group to distinguish the type of error or the source of error concerned.

`nErrCode`: The error code is specific for each error group.

### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### 4.3 Specific

#### 4.3.1 FB_MDP_CPU_Read

The function block enables querying of the IPC diagnostics module CPU.

#### VAR_INPUT

```plaintext
VAR_INPUT
  bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input *)
  tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
```
### Function blocks

```plaintext
iModIdx : USINT := 0; (* Index number of chosen MDP module *)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR

**bExecute**: The function block is called by a rising edge on the input `bExecute`, if the block is not already active.

**tTimeout**: Specifies a maximum length of time for the execution of the function block.

**iModIdx**: If several instances of an IPC diagnostics module exist, a selection can be made by means of the input `iModIdx (0,...,n)`.

**sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type `T_AmsNetId`) can be specified here.

#### VAR_OUTPUT

```plaintext
VAR_OUTPUT

bBusy : BOOL;
bError : BOOL;
nErrID : USINT;
iErrPos : USINT;
stMDP_ModuleHeader : ST_MDP_ModuleHeader;
stMDP_ModuleContent : ST_MDP_CPU;
END_VAR

**bBusy**: This output is TRUE as long as the function block is active.

**bError**: Becomes TRUE as soon as an error situation occurs.

**nErrID**: Returns an error code [33] if the `bError` output is set.

**iErrPos**: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

**stMDP_ModuleHeader**: At this output the header information for the read IPC diagnostics modules is displayed in the form of the structure `ST_MDP_ModuleHeader[29].`

**stMDP_ModuleContent**: The information from TableId 1 of the read IPC diagnostics module is displayed at this output in the form of the structure `ST_MDP_CPU[29].`

#### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

#### 4.3.2 FB_MDP_Device_Read_DevName

```plaintext
FB_MDP_Device_Read_DevName

VAR_INPUT

bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input.*)
tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```

The function block enables the device name to be queried. This information can be found in the general area of the IPC diagnostics.

#### VAR_INPUT

```plaintext
VAR_INPUT

bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input.*)
tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```
**Function blocks**

**bExecute:** The function block is called by a rising edge on the input `bExecute`, if the block is not already active.

**tTimeout:** Specifies a maximum length of time for the execution of the function block.

**sAmsNetId:** To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

### VAR_OUTPUT

- `bBusy`: This output is TRUE as long as the function block is active.
- `bError`: Becomes TRUE as soon as an error situation occurs.
- `nErrID`: Returns an error code [33] if the `bError` output is set.
- `sDevName`: The queried name is output as a string at this output.

### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

#### 4.3.3 FB_MDP_IdentityObj_Read

The function block allows the query of the `IdentityObject` table of the General Area of the IPC-Diagnose.

**VAR_INPUT**

- `bExecute`: The function block is called by a rising edge on the input `bExecute`, if the block is not already active.
- `tTimeout`: Specifies a maximum length of time for the execution of the function block.
- `sAmsNetId`: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

**VAR_OUTPUT**

- `bBusy`: This output is TRUE as long as the function block is active.
- `bError`: Becomes TRUE as soon as an error situation occurs.
- `nErrID`: Returns an error code [33] if the `bError` output is set.
- `sDevName`: The queried name is output as a string at this output.
Function blocks

```plaintext
iErrPos : USINT;
stMDP_ModuleContent : ST_MDP_IdentityObject;
END_VAR
```

**bBusy:** This output is TRUE as long as the function block is active.

**bError:** Becomes TRUE as soon as an error situation occurs.

**nErrID:** Returns an error code [33] if the bError output is set.

**iErrPos:** If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

**stMDP_ModuleContent:** The information from the table is displayed at this output in the form of the structure ST_MDP_IdentityObject [30].

Serial number is no longer supported

---

**Outdated parameter leads to error situation**

In the Identity Object the serial number of the IPC is read from the MDP General Area. This parameter is obsolete. The parameter is no longer supported for newer Beckhoff IPC devices. This causes the function block to return an error and name the error position (iErrPos = 4), which corresponds to the iSerialNumber parameter.

Alternatively, the serial number can be read from the MDP Device Area. See corresponding example [51]: It is recommended to use the PLC library Tc3_IPCDialog, which is the successor to the PLC library Tc2_MDP.

---

**Requirements**

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

**4.3.4 FB_MDP_NIC_Read**

```plaintext
VAR_INPUT
VAR_INPUT
  bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input.*)
tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled.*)
iModIdx : USINT := 0; (* Index number of chosen MDP module *)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)

FB_MDP_NIC_Read
```

The function block enables querying of the IPC diagnostics Moduls NIC (Network Interface Card).

---

**VAR_INPUT**

**bExecute:** The function block is called by a rising edge on the input bExecute, if the block is not already active.

**tTimeout:** Specifies a maximum length of time for the execution of the function block.

**iModIdx:** If several instances of an IPC diagnostics module exist, a selection can be made by means of the input iModIdx (0,...,n).
sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

VAR OUTPUT

VAR_OUTPUT
  bBusy : BOOL;
  bError : BOOL;
  nErrID : UDINT;
  iErrPos : USINT;
  stMDP_ModuleHeader : ST_MDP_ModuleHeader;
  stMDP_ModuleContent : ST_MDP_NIC_Properties;
END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code [33] if the bError output is set.

iErrPos: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

stMDP_ModuleHeader: At this output the header information for the read IPC diagnostics modules is displayed in the form of the structure ST_MDP_ModuleHeader [29].

stMDP_ModuleContent: The information from TableID 1 of the read IPC diagnostics module is displayed at this output in the form of the structure ST_MDP_NIC [30].

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

4.3.5 FB_MDP_NIC_Write_IP

The function block enables a new IP address to be set. This element is part of the IPC diagnostics Moduls NIC.

Please note that changes of this kind affect an existing network connection to the computer.

VAR_INPUT

VAR_INPUT
  bExecute : BOOL;
  nDynModuleId : BYTE; (* the dynamic module id *)
  sIPAddress : T_MaxString; (* IP Address *)
  tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
  sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

nDynModuleId: The dynamic Module ID belonging to the selected network module is specified at this input.
**sIPAddress**: The IP address specified at this input in the form of a string is transmitted.

**tTimeout**: Specifies a maximum length of time for the execution of the function block.

**sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

### VAR_OUTPUT

```
VAR_OUTPUT

bBusy : BOOL;
bError : BOOL;
nErrID : UDINT;
END_VAR
```

**bBusy**: This output is TRUE as long as the function block is active.

**bError**: Becomes TRUE as soon as an error situation occurs.

**nErrID**: Returns an error code \[33\] if the bError output is set.

### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### 4.3.6 FB_MDP_SiliconDrive_Read

The function block enables querying of the IPC diagnostics module SiliconDrive.

**Obsolete functionality**

The SiliconDrive hardware was replaced by newer memory card types. The functionality is therefore obsolete. We recommend using the query of the IPC diagnostic module Physical Drive SMARTParameters instead.

### VAR_INPUT

```
VAR_INPUT

bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input *)
tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
iModIdx : USINT := 0; (* Index number of chosen MDP module *)
sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```

**bExecute**: The function block is called by a rising edge on the input bExecute, if the block is not already active.

**tTimeout**: Specifies a maximum length of time for the execution of the function block.

**iModIdx**: If several instances of an IPC diagnostics module exist, a selection can be made by means of the input iModIdx (0,...,n).

**sAmsNetId**: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.
VAR_OUTPUT

VAR_OUTPUT

| bBusy   : BOOL;      |
| bError  : BOOL;      |
| nErrID  : UDINT;     |
| iErrPos : USINT;     |
| stMDP_ModuleHeader : ST_MDP_ModuleHeader; |
| stMDP_ModuleContent : ST_MDP_SiliconDrive; |

END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code \[33\] if the bError output is set.

iErrPos: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

stMDP_ModuleHeader: At this output the header information for the read IPC diagnostics modules is displayed in the form of the structure ST_MDP_ModuleHeader \[29\].

stMDP_ModuleContent: The information from TableID 1 of the read IPC diagnostics module is displayed at this output in the form of the structure ST_MDP_SiliconDrive \[30\].

Notice

The querying of the IPC-diagnostics Drive module is one of the more time consuming processes. Hence, the Standard ADS Timeout can by all means be exceeded. This can be remedied by increasing the time period tTimeout applied to the input of the function block.

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### 4.3.7 FB_MDP_SW_Read_MdpVersion

```
 FB_MDP_SW_READ_MDPVERSION

 bExecute : BOOL;
 tTimeout : TIME :=DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
 sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)

```

The function block enables querying of the MDP version. This information can be found in the software module in the configuration area of the IPC diagnostics.

The MDP version is independent of the PLC library version. The constant stLibVersion_Tc2_MDP \[32\] is used to query the PLC library version.

VAR_INPUT

VAR_INPUT

| bExecute : BOOL; | (* Function block execution is triggered by a rising edge at this input.*) |
| tTimeout : TIME :=DEFAULT_ADS_TIMEOUT; | (* States the time before the function is cancelled. *) |
| sAmsNetId : T_AmsNetId; | (* keep empty '' for the local device *) |

END_VAR

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.
tTimeout: Specifies a maximum length of time for the execution of the function block.

sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

**VAR_OUTPUT**

```plaintext
VAR_OUTPUT
    bBusy : BOOL;
    bError : BOOL;
    nErrID : UDINT;
    sMdpVersion : STRING(23); (* complete MDP version as string [e.g.: '1, 0, 4, 47'] *)
    iMajorNbr : UINT; (* major number [e.g.: 1] *)
    iMinorNbr : UINT; (* minor number [e.g.: 4] *)
    iRevNbr : UINT; (* revision number [e.g.: 47] *)
END_VAR
```

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code [33] if the bError output is set.

sMdpVersion: At this output the queried MDP version is output as a string.

iMajorNbr: The first part of the version number output is iMajorNbr.

iMinorNbr: The second part of the version number output is iMinorNbr.

iRevNbr: The third part of the version number output is iRevNbr.

**Requirements**

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

**4.3.8 FB_MDP_TwinCAT_Read**

The function block enables querying of the IPC diagnostics module TwinCAT.

**VAR_INPUT**

```plaintext
VAR_INPUT
    bExecute : BOOL; (* Function block execution is triggered by a rising edge at this input *)
    tTimeout : TIME := DEFAULT_ADS_TIMEOUT; (* States the time before the function is cancelled. *)
    iModIdx : USINT := 0; (* Index number of chosen MDP module *)
    sAmsNetId : T_AmsNetId; (* keep empty '' for the local device *)
END_VAR
```

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

tTimeout: Specifies a maximum length of time for the execution of the function block.

iModIdx: If several instances of an IPC diagnostics module exist, a selection can be made by means of the input iModIdx (0,...,n).
sAmsNetId: To execute the query on the local device, it is not necessary to specify this input variable. Alternatively, an empty string can be specified. To direct the query to another computer, its AMS Net Id (of type T_AmsNetId) can be specified here.

VAR_OUTPUT

VAR_OUTPUT
bBusy : BOOL;
bError : BOOL;
nErrID : UDINT;
iErrPos : USINT;
stMDP_ModuleHeader : ST_MDP_ModuleHeader;
stMDP_ModuleContent : ST_MDP_TwinCAT;
END_VAR

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an error code \[33\] if the bError output is set.

iErrPos: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

stMDP_ModuleHeader: At this output the header information for the read IPC diagnostics modules is displayed in the form of the structure ST_MDP_ModuleHeader \[29\].

stMDP_ModuleContent: The information from TableID 1 of the read IPC diagnostics module is displayed at this output in the form of the structure ST_MDP_TwinCAT \[31\].

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>
5 Data types

5.1 General data types

5.1.1 E_MDP_AddrArea

TYPE E_MDP_AddrArea :
   eMDP_Area_ConfigArea := 16#8,
   eMDP_Area_ServiceArea := 16#B,
   eMDP_Area_DeviceArea := 16#F
);
END_TYPE

The enumeration E_MDP_AddrArea defines constant values for the different site the IPC diagnostics.

5.1.2 E_MDP_ModuleType

TYPE E_MDP_ModuleType :
   eMDP_ModT_NIC             := 16#0002,
   eMDP_ModT_Time             := 16#0003,
   eMDP_ModT_UserManagement   := 16#0004,
   eMDP_ModT_RAS             := 16#0005,
   eMDP_ModT_FTP             := 16#0006,
   eMDP_ModT_SMB             := 16#0007,
   eMDP_ModT_TwinCAT         := 16#0008,
   eMDP_ModT_Datastore       := 16#0009,
   eMDP_ModT_Software        := 16#000A,
   eMDP_ModT_CPU             := 16#000B,
   eMDP_ModT_Memory          := 16#000C,
   eMDP_ModT_Firewall        := 16#000E,
   eMDP_ModT_FileSystemObject := 16#0010,
   eMDP_ModT_PLC             := 16#0012,
   eMDP_ModT_DisplayDevice   := 16#0013,
   eMDP_ModT_KW              := 16#0014,
   eMDP_ModT_FBWF            := 16#0015,
   eMDP_ModT_SiliconDrive    := 16#0017,
   eMDP_ModT_OS              := 16#0018,
   eMDP_ModT_Raid            := 16#0019,
   eMDP_ModT_Fan             := 16#001B,
   eMDP_ModT_Mainboard       := 16#001C,
   eMDP_ModT_DiskManagement  := 16#001D,
   eMDP_ModT_UPS             := 16#001E,
   eMDP_ModT_Misc            := 16#0100
);  
END_TYPE

The enumeration E_MDP_ModuleType defines constant values for the different module types in the MDP. A module type can occur several times per device. Hence, a device with two Ethernet interfaces also has two MDP NIC modules.

Detailed information on the modules can be found in the documentation for the IPC diagnostics under module types.

Notice: This module type is not the same as the dynamic module ID!

5.1.3 ST_MDP_Addr

TYPE ST_MDP_Addr :
  STRUCT
  nArea : BYTE; (* Area [range: 0x0-0xF] *)
  nModuleId : BYTE; (* Dynamic Module Id [range: 0x00-0xFF] *)
  nTableId : BYTE; (* Table Id [range: 0x00-0xFF] *)
  nFlag : BYTE; (* Flags [range: 0x00-0xFF] *)
  nSubIdx : BYTE; (* SubIndex [range: 0x00-0xFF] *)
END_STRUCT

The enumeration ST_MDP_Addr defines constant values for the different site the IPC diagnostics.
The structure contains information that is required for the MDP addressing.

nArea: Possible MDP areas are listed in E_MDP_AddrArea [\ref{28}].

nModuleId: The Module ID is assigned dynamically. It does not correspond to the module types listed in E_MDP_ModuleType. The function block FB_MDP_ScanModules [\ref{18}] can be used in order to determine a dynamic Module ID for a special type of module.

nTableId: This value specifies the number of the selected table of the selected module.

nFlag: This parameter is used internally only. It remains at the default value of 0x00.

nSubIdx: The Subindex parameter corresponds to the subindex in a table in an MDP module.

More detailed information on MDP addressing can be found in the Drive Manager documentation.

5.1.4 ST_MDP_ModuleHeader

The structure contains device information. This information always corresponds to the Table ID 0 of an MDP module. Each module possesses this module header.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>iLen</td>
<td>Specifies the number of parameters in the Table ID, in this case the module</td>
</tr>
<tr>
<td>nAddr</td>
<td>Specifies the address of the module.</td>
</tr>
<tr>
<td>sType</td>
<td>Specifies the type of module. Possible types are listed in the MDP module</td>
</tr>
<tr>
<td>sName</td>
<td>Specifies the name of this MDP module.</td>
</tr>
<tr>
<td>nDevType</td>
<td>Specifies the type of MDP module as code.</td>
</tr>
</tbody>
</table>

5.2 Structures specific MDP modules

5.2.1 ST_MDP_CPU

The structure contains information on the IPC diagnostics module CPU.

This complete information can be queried by means of the function block FB_MDP_CPU_Read [\ref{19}].

The parameters existing in this structure correspond to the subindices of the first table (Table ID 1) within the IPC-diagnostics CPU module.
5.2.2 ST_MDP_IdentityObject

TYPE ST_MDP_IdentityObject:
  STRUCT
    iLen : UINT; (* Length *)
    iVendor : UDINT; (* Vendor *)
    iProductCode : UDINT; (* Product Code *) (* not yet supported *)
    iRevNumber : UDINT; (* Revision Number *) (* not yet supported *)
    iSerialNumber : UDINT; (* Serial Number *)
  END_STRUCT
  END_TYPE

The structure contains information on the 'IdentityObject' table, which can be found in the IPC diagnostics general area.

This complete information can be queried by means of the function block FB_MDP_IdentityObj_Read [21].

The parameters in this structure correspond to the subindices of the 'identity object' table in the IPC diagnostics general area.

5.2.3 ST_MDP_NIC_Properties

TYPE ST_MDP_NIC_Properties:
  STRUCT
    iLen : UINT; (* Length *)
    sMACAddress : T_MaxString; (* MAC Address *)
    sIPAddress : T_MaxString; (* IP Address *)
    sSubnetMask : T_MaxString; (* Subnet Mask *)
    bDHCP : BOOL; (* DHCP *)
  END_STRUCT
  END_TYPE

The structure contains information on the IPC diagnostics module NIC (Network Interface Card).

This complete information can be queried by means of the function block FB_MDP_NIC_Read [22].

The parameters existing in this structure correspond to the subindices of the first table (Table ID 1) within the IPC-diagnostics NIC module.

5.2.4 ST_MDP_SiliconDrive

TYPE ST_MDP_SiliconDrive:
  STRUCT
    iLen : UINT; (* Length *)
    iTOTALEraseCounts : UDINT; (* Total EraseCounts (lower 4 bytes) *)
    iDriveUsage : UINT; (* Drive Usage (%) *)
    iNbrSpares : UINT; (* Number of Spares *)
    iNbrUsedSpares : UINT; (* Spares Used *)
    iTOTALEraseCountsHigh : UDINT; (* Total EraseCounts (higher 4 bytes) *)
  END_STRUCT
  END_TYPE

The structure contains information on the MDP SiliconDrive module.

This complete information can be queried by means of the function block FB_MDP_SiliconDrive_Read [24].

iLen: iLen indicates the number the MDP elements in the table in the MDP module.

iTOTALEraseCounts: This value indicates the total number of write and delete cycles of all memory blocks of a Silicon Drive. This number is a 64-bit value. iTOTALEraseCounts contains the lower 32 bits.

iTOTALEraseCountsHigh: This value indicates the total number of write and delete cycles of all memory blocks of a Silicon Drive. This number is a 64-bit value. iTOTALEraseCountsHigh contains the higher 32 bits.

iDriveUsage: This parameter indicates the calculated wear of the Silicon Drive. The value is based on two million write cycles per block as maximum value.

iNbrSpares: Reserved blocks are used to replace worn memory blocks. iNbrSpares indicates the number of spare blocks available on the Silicon Drive.
iNbrUsedSpares: The value indicates the number of spare blocks already in use.

The parameters existing in this structure correspond to the subindices of the first table (Table ID 1) within the IPC diagnostics SiliconDrive module.

**Obsolete functionality**
The SiliconDrive hardware was replaced by newer memory card types. The functionality is therefore obsolete. We recommend using the query of the IPC diagnostic module Physical Drive SMART Parameters instead.

### 5.2.5  ST_MDP_TwinCAT

```plaintext
TYPE ST_MDP_TwinCAT :
STRUCT
    iLen           : UINT;    (* Length *)
    iMajorVersion  : UINT;    (* Major Version *)
    iMinorVersion  : UINT;    (* Minor Version *)
    iBuild         : UINT;    (* Build *)
    sAmsNETid      : T_MaxString;  (* Ams NET ID *)
    iRegLevel      : UDINT;   (* TwinCAT registration level *)
    iStatus        : UINT;    (* TwinCAT status *)
    iRunAsDev      : UINT;    (* Run As Device *)  (* available for WindowsCE *)
    iShowTargetVisu: UINT;    (* show target visualization *)  (* available for WindowsCE *)
    iLogFileSize   : UDINT;   (* log file size *)  (* available for WindowsCE *)
    sLogFilePath   : T_MaxString;  (* log file path *)  (* available for WindowsCE *)
END_STRUCT
END_TYPE
```

The structure contains information on the MDP TwinCAT module.

This complete information can be queried by means of the function block FB_MDP_TwinCAT_Read.[26]

The parameters existing in this structure correspond to the subindices of the first table (Table ID 1) within the MDP TwinCAT module.
6 Global Constants

6.1 Global_Version

All libraries have a certain version. The version is indicated in the PLC library repository, for example. A global constant contains the information about the library version:

Global_Version

VAR_GLOBAL CONSTANT
    stLibVersion_Tc2_MDP : ST_LibVersion;
END_VAR

stLibVersion_Tc2_MDP: Version number of the Tc2_MDP library (type: ST_LibVersion).

To check whether the version you have is the version you need, use the function F_CmpLibVersion (defined in Tc2_System library).

All other options for comparing library versions, which you may know from TwinCAT 2, are outdated!
7 Error Codes

7.1 Error codes overview

The function blocks of the Tc2_MDP library have an nErrID output. This value is 4 bytes in size and returns the error code in the event of an error. nErrID is comprised of two parts:

<table>
<thead>
<tr>
<th>Error Group (MSB) 2 Byte</th>
<th>Error Code 2 Byte (LSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x EC80</td>
<td>E_MDP_ErrCodesPLC</td>
</tr>
<tr>
<td>0x ECA6</td>
<td>MDP general error</td>
</tr>
<tr>
<td>0x ECA7</td>
<td>MDP API error</td>
</tr>
<tr>
<td>0x ECA8</td>
<td>ADS error</td>
</tr>
<tr>
<td>0x ECAF</td>
<td>MDP module specific error</td>
</tr>
</tbody>
</table>

The function block FB_MDP_SplitErrorID enables the automatic separation of the variable nErrID into error group and error code.

Error Group

The fault group describes the type of error that has occurred. The different groups are listed in the enumeration E_MDP_ErrGroup.

All errors generated within the PLC library have the error group 0xEC80.

Error Code

The error code describes the specific error.

For errors within the PLC library with the error group 0xEC80, the identifiers are listed in the enumeration E_MDP_ErrCodesPLC. A description of the other error codes can be found in the documentation of the IPC diagnostics in the section on error messages.

General MDP-dependent errors are output in the error group 16#ECA6 “General error codes”. These errors sometimes indicate that an element from the module element list is not available. Example: 16#ECA60105 “No data available” If, in the case of a general or specific function block (see access options), several elements are queried at the same time and one of these elements is not available or exhibits an error, then the output variable iErrPos indicates the index position (0..n) at which the error occurred for the first time. All elements below this index were queried successfully and are indicated despite the generation of an error at the output.

Example:

nErrID = 0x ECA8 0745

The error group is 0x ECA8, therefore it is an Ads error. The error code is 0x 0745, therefore it is a timeout error.

Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

7.2 E_MDP_ErrGroup

TYPE E_MDP_ErrGroup :
    eMDP_Err_NoError := 16#0000, (* Success - No Error *)
    eMDP_Err_PLL := 16#EC80, (* PLC library internal error codes *)
    eMDP_Err_GenErr := 16#ECA6, (* General error codes *)
The enumeration `E_MDP_ErrGroup` defines constant values for the different error groups in the MDP. These indicate the type of error. The values appear in the error codes ▶ 16#ECA6 - 16#ECAF are described there. The error codes from group 16#EC80 are generated by the PLC MDP library and are described in chapter `E_MDP_ErrCodesPLC` ▶ 16#EC80.

### Requirements

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>

### E_MDP_ErrCodesPLC

The enumeration `E_MDP_ErrCodesPLC` defines constant values for the different errors that can be generated internally in the library. These values appear in the error codes ▶ 16#ECA6 - 16#ECAF are described there. The error codes from group 16#EC80 are generated by the PLC MDP library and are described in chapter `E_MDP_ErrCodesPLC` ▶ 16#EC80.

- **eMDP_ErrPLC_TimeOut**
  The error `eMDP_ErrPLC_TimeOut` is generated if the time period `tTimeout` applied to the input of the function block has expired. The length of the processing time can vary depending on the MDP query. Due to the internal processes, the processing time can sometimes exceed the Standard ADS Timeout. This can be remedied by increasing the time period `tTimeout` applied to the input of the function block.

- **eMDP_ErrPLC_ModuleNotFound**
  A list of active modules exists in the MDP. The function blocks in the PLC MDP library search this list for the queried module. If the list does not contain the module, then the error `eMDP_ErrPLC_ModuleNotFound` is output. This is the case when the particular module/device is not installed on the system or does not even exist.

- **eMDP_ErrPLC_BufferTooSmall**
  If a buffer has been specified at the input of the function block by means of pointers, then it is possible that this is not large enough for the existing data. In this case the error `eMDP_ErrPLC_BufferTooSmall` is output.

- **eMDP_ErrPLC_ElementNotFound**
  The query of a certain element was not successful. The element was not found. The respective module or element may not even be present on the system.
A general description can be found in the MDP Information Model (IPC Diagnosis).

**Requirements**

<table>
<thead>
<tr>
<th>Development environment</th>
<th>Target platform</th>
<th>PLC libraries to include</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwinCAT v3.1.0</td>
<td>PC or CX (x86, x64, ARM)</td>
<td>Tc2_MDP</td>
</tr>
</tbody>
</table>
8 Samples

Update: Tc3_IPCDiag library

The TwinCAT 3 PLC library Tc2_MDP is the predecessor to Tc3_IPCDiag. With the new Tc3_IPC-Diag library the number of readable parameters has been increased and the user interface has been optimized. It is recommended to use the Tc3_IPCDiag library. Future extensions will no longer be performed in the Tc2_MDP library. It is not recommended to use the Tc2_MDP library for new projects. All functionalities of the Tc2_MDP library can also be found in the new Tc3_IPCDiag library.

The section **Querying CPU data (generic)** describes a sample program for reading the CPU data of the IPC diagnostics via the generic function block FB_MDP_ReadElement. It is designed in such a way that it can be easily extended with the basic knowledge of the MDP information model for access to other IPC diagnosis modules. The sample for **Querying the fan status (generic)** is based on the sample program for querying the CPU data.

The section **Querying CPU data (specific)** describes a sample program for reading the CPU data of the IPC diagnostics via the specific function block FB_MDP_CPU_Read. The sample cannot be extended for access to other IPC diagnostic modules, e.g. fan data.

Further sample programs are described in the sections **Reading IPC serial numbers** and **Setting the IP address**.

8.1 Querying CPU data (generic)

This sample demonstrates access to CPU data by the IPC diagnostics via the generic function block FB_MDP_ReadElement.

The program is structured in such a way that it can easily be adapted for access to other IPC diagnostics modules. Program lines, which have to be modified to adapt the program to other IPC diagnostics modules, are identified in the comment with the string "/**."

Following the program code of the sample you will find the textual Description of the sample program.

Sample: access via the generic function block FB_MDP_ReadElement

Individual CPU data can be read via a subindex in module CPU in the Configuration Area of the IPC diagnostics. The generic function block FB_MDP_ReadElement is used for this purpose.

**Enumeration definition**

```plaintext
/* = simply adjust these lines if modifying code for own purposes

// central definition of state machine states
// (supports easy program modification)
(attribute 'qualified_only')
TYPE E_State :
{  
  Idle, 
  ReadCPUUsageInit, /** initiate reading CPU usage 
  ReadCPUUsageProcess /** process reading CPU usage 
};
END_TYPE
```

**Variable Declaration**

```plaintext
PROGRAM MAIN
VAR
  // internal use
  sAmsNetId : STRING := ''; /** ADS Net ID (local = '')
  eState : E_State; // Enum with index for state machine
  bStart : BOOL := TRUE; // flag to trigger (re)start of statemachine
  nData : UINT; // data storage for unsigned integer
  stMDP_Addr : ST_MDP_Addr; // structure will include all address parameters
```

The section **Querying CPU data (generic)** describes a sample program for reading the CPU data of the IPC diagnostics via the generic function block FB_MDP_ReadElement. It is designed in such a way that it can be easily extended with the basic knowledge of the MDP information model for access to other IPC diagnosis modules. The sample for **Querying the fan status (generic)** is based on the sample program for querying the CPU data.

The section **Querying CPU data (specific)** describes a sample program for reading the CPU data of the IPC diagnostics via the specific function block FB_MDP_CPU_Read. The sample cannot be extended for access to other IPC diagnostic modules, e.g. fan data.

Further sample programs are described in the sections **Reading IPC serial numbers** and **Setting the IP address**.

8.1 Querying CPU data (generic)

This sample demonstrates access to CPU data by the IPC diagnostics via the generic function block FB_MDP_ReadElement.

The program is structured in such a way that it can easily be adapted for access to other IPC diagnostics modules. Program lines, which have to be modified to adapt the program to other IPC diagnostics modules, are identified in the comment with the string "/**."

Following the program code of the sample you will find the textual Description of the sample program.

Sample: access via the generic function block FB_MDP_ReadElement

Individual CPU data can be read via a subindex in module CPU in the Configuration Area of the IPC diagnostics. The generic function block FB_MDP_ReadElement is used for this purpose.

**Enumeration definition**

```plaintext
/* = simply adjust these lines if modifying code for own purposes

// central definition of state machine states
// (supports easy program modification)
(attribute 'qualified_only')
TYPE E_State :
{  
  Idle, 
  ReadCPUUsageInit, /** initiate reading CPU usage 
  ReadCPUUsageProcess /** process reading CPU usage 
};
END_TYPE
```

**Variable Declaration**

```plaintext
PROGRAM MAIN
VAR
  // internal use
  sAmsNetId : STRING := ''; /** ADS Net ID (local = '')
  eState : E_State; // Enum with index for state machine
  bStart : BOOL := TRUE; // flag to trigger (re)start of statemachine
  nData : UINT; // data storage for unsigned integer
  stMDP_Addr : ST_MDP_Addr; // structure will include all address parameters
```
Program code

// For an easy re-use of the following code for own purposes, parts of this sample program use
// "general" data names (and copy the results in specific variables after processing the code).

CASE eState OF
  E_State.Idle:
    IF bStart THEN
      bStart := FALSE;
      eState := E_State.ReadCPUusageInit; //** initiate first state
    END_IF
  E_State.ReadCPUusageInit:
    stMDP_Addr.nArea := INT_TO_BYTE(eMDP_Area_ConfigArea); //** set area address to "Config Area"
    stMDP_Addr.nTableId := 1; //** table ID in module for "cpu properties"
    stMDP_Addr.nSubIdx := 2; //** subindex in table ID for "CPU usage"
    fbReadMDPElement(bExecute := TRUE, // Flag: trigger execution of FB
      eModuleType := eMDP_ModT_CPU, //** desired module type = CPU
      stMDP_Addr := stMDP_Addr, // MDP address structure. Dynamic module ID added internally.
      iModIdx := 0, //** instance of desired module type (0 = first instance)
      pDstBuf := ADR(nData), // buffer for storing data
      cbDstBufLen := SIZEOF(nData), // length of buffer
      sAmsNetId := sAmsNetId, // AMS Net ID
    ); //** Note: fbReadMDPElement.tTimeOut must be > cycle time!
    eState := E_State.ReadCPUusageProcess; //** next state: process FB
  E_State.ReadCPUusageProcess:
    fbReadMDPElement(bExecute := FALSE); // Flag: Get execution state of FB
    eState := E_State.Idle; //** Note: fbReadMDPElement.tTimeOut must be > cycle time!
    IF NOT fbReadMDPElement.bBusy THEN // FB executed?
      IF fbReadMDPElement.bError THEN // Error?
        bError := TRUE; // set error flag
        nErrID := fbReadMDPElement.nErrID; // store error id (16#ECA60105 = BIOS or HW does not support this data (here: mainboard data))
      ELSE // set parameters for next steps
        bError := FALSE;
        nCpuUsage := nData; //** store CPU usage in dedicated variable
      END_IF
    eState := E_State.ReadCPUusageInit; //** next state
  END_CASE

Description of the sample program

The function block FB_MDP_ReadElement requires at least the parameters listed below:
The output values of the function block are described here for completeness:

A very important role is played by the structure for addressing the desired information. It is described by the data type ST_MDP_Addr:
There are two possibilities to determine the parameter "nArea":

- The individual areas are stored as enumerations in the Tc2_MDP library. This entry can be used as easily readable input parameter (dashed red arrow).
- The value for the area can alternatively be taken from the index of a table (left-hand 4 bits - red arrow)
The parameter "nModule" is NOT an input parameter. The module address (=ModuleID) determined by the function block is assigned to it after it has determined the desired module instance.

"nTableId" corresponds to the right-hand 4 bits of the table index (yellow arrow).

"Subindex" is the number of the entry in the table (blue arrow).

The "eModuleType" parameter specifies the type of the module. There is also an enumeration for the module type that can be used for better readability of the program:
Alternatively the value can also be taken from the module description and entered directly.

Structure of the state machine

The states of the state machine are defined via enumeration values in a DUT and can thus be adapted or extended centrally and easily.
CASE eState_InitiateStateMachine:
  eState_ReadStateMachine(bExecute := FALSE); // initiate program parameters (if required)
eState := eState_ReadStateMachine; // initiate first state

CASE eState_ReadCPUUsageInit:
  stMPD.Addr.nArea := INIT_F_BTR(EndMPD.Area_ConfigArea); // set area address to "Config Area"
eNumAddr := 1; // table ID in module for "cpu properties"
  stMPD.Addr.nSubIdx := 2; // subindex in table ID for "CPU usage"
  for i := 0 TO 10 DO // read 10 areas
    eState := eState_ReadCPUUsageProcess; // next state: process FB
  NEXT i

CASE eState_ReadCPUUsageProcess:
  fbReadMDElement(bExecute := FALSE); // Flags: Get execution state of FB
  IF NOT fbReadMDElement.bError THEN // FB executed?
    bError := TRUE;
    eState := eState_IdleState; // finish state machine
  ELSE
    bError := FALSE;
    iCPUUsage := uData; // store CPU usage in dedicated variable
    eState := eState_ReadCPUUsageInit; // next state
  END IF

CASE eState_IdleState:
  // idle state
  IF bRestart THEN // flag = TRUE -> restart state machine
    eState := eState_InitiateStateMachine;
  END IF
ELSE
  eState := eState_IdleState; // capture undefined states
END_CASE
Functional areas of the state machine

```c
CASE eState OF
  eState_InitiateStateMachine: // initiate program parameters (if required) 
    ifReadMDPElement(bExecute := FALSE); // Flag: trigger event 
    eState := eState_ReadCPUUsageInit; // requested state
  eState_ReadCPUUsageInit: 
    stMDP.Area := TB_NMT_User("Area"); // set area address to “Config Area”
    stMDP.Area := TB_NMT_User("CPU properties"); // set area ID
    stMDP.Area := FB_NMT_User("CPU usage"); // set area ID
    eState := eState_ReadCPUProcess; // next state: process FB
  eState_ReadCPUProcess: 
    ifReadMDPElement(bExecute := FALSE); // Flag: trigger event 
    IF NOT ifReadMDPElement.bBusy THEN // FB executed
      IF ifReadMDPElement.bError THEN // Error?
        bError := TRUE; // set error
        nErrID := ifReadMDPElement.nErrID; // store error ID
        eState := eState_IdleState; // finish state machine
      ELSE
        nError := FALSE; // turn off
      END_IF
      eState := eState_IdleState; // idle state
    END_IF
  eState_IdleState: 
    eState := eState_InitiateStateMachine; // flag = TRUE -> restart
  END_CASE
END_TYPE
```

Input and output parameters of the sample program

```c
TYPE E_State :
  
  eState_InitiateStateMachine := 0; // initiate state machine (set parameters)
  eState_ReadCPUUsageInit := 20; // initiate reading serial number of mainboard
  eState_ReadCPUUsageProcess := 21; // process reading serial number of mainboard
  eState_IdleState := 100; // idle state
) UINT;
END_TYPE
```
8.2 Querying CPU data (specific)

This sample demonstrates access to CPU data by the IPC diagnostics via the specific function block FB_MDP_CPU_Read [19].

This sample cannot be modified for access to other IPC diagnostics modules such as fan data.

Following the program code of the sample you will find the textual Description of the sample program [46].

Sample: access via the specific function block FB_MDP_CPU_Read

The specific function block FB_MDP_CPU_Read facilitates access to selected data of the CPU module in the Configuration Area of the IPC diagnostics.

Enumeration definition

```plaintext
/** = simply adjust these lines if modifying code for own purposes

// central definition of state machine states
// (supports easy program modification)
{attribute 'qualified_only'}
TYPE E_State :
(
  Idle,
  ReadCPUDataInit,    //** initiate reading CPU data
  ReadCPUDataProcess  //** process reading CPU data
);
END_TYPE
```

Variable Declaration

```plaintext
PROGRAM MAIN
VAR
  // internal use
  sAmsNetId          : STRING := '';   //** ADS Net ID (local = '')
  eState             : E_State;        // Enum with index for state machine
  bStart             : BOOL := TRUE;   // flag to trigger (re)start of state machine

  // FB instances
  fbReadCPUData      : FB_MDP_CPU_Read; // instance of FB for reading CPU data

  // results of execution
  bError             : BOOL;           // error flag (indicator: error occurred)
  nErrID             : UDINT;          // last error ID
  stHeaderCpuMod     : ST_MDP_ModuleHeader; // buffer for header data of CPU module
  stCPUData          : ST_MDP_CPU;     // structure which will contain CPU data
END_VAR
```

Program code

```plaintext
// For an easy reuse of the following code for own purposes, parts of this sample program use
// "general" data names (and copy the results in specific variables after processing the code).
CASE eState OF
  E_State.Idle:
    IF bStart THEN
      bStart := FALSE;
      eState := E_State.ReadCPUDataInit;    //** initiate first state
    END_IF
  E_State.ReadCPUDataInit:
    /** trigger FB: request CPU data
    fbReadCPUData(
      bExecute := TRUE,        // Flag: trigger execution of FB
      iModIdx := 0,            //** Instance of desired module type (0 = first instance)
      sAmsNetId := sAmsNetId); // AMS Net ID

    eState := E_State.ReadCPUDataProcess; /** next state: process FB
  E_State.ReadCPUDataProcess:
    /** process FB: request CPU data
    fbReadCPUData(bExecute := FALSE); // Flag: Get execution state of FB

    IF NOT fbReadCPUData.bBusy THEN    // FB executed?
      IF fbReadCPUData.bError THEN      // Error?
        bError := TRUE;                 // set error flag
```
Samples

nErrID := fbReadCPUData.nErrID; // store error id
eState := E_State.Idle; // finish state machine
ELSE // set parameters for next steps
bError := FALSE; // turn off error flag
stHeaderCpuMod := fbReadCPUData.stMDP_ModuleHeader; //
** store CPU module header data
stCPUData := fbReadCPUData.stMDP_ModuleContent; //** store CPU data
eState := E_State.ReadCPUDataInit; //** read next set of CPU data
END_IF
END_IF
END_CASE

Description of the sample program

The function block "FB_MDP_CPU_READ" requires a minimum of two parameters:

- The AMS Net ID as input parameter for the address of the IPC (local: "")
- A structure stMDP_ModuleContent that contains the data after calling the function block.

FB_MDP_CPU_Read

VAR_INPUT:

bExecute: The function block is called by a rising edge on the input bExecute, if the block is not already active.

bTimeout: Specifies a maximum length of time for the execution of the function block.

iModId: If several instances of an MDP module exist, a selection can be made by means of the input iModId (0...n)

sAmsNetId: For local access don't specify this input or allocate an empty string. For remote access to another computer specifies AMS Net Id.

VAR_OUTPUT:

bBusy: This output is TRUE as long as the function block is active.

bError: Becomes TRUE as soon as an error situation occurs.

nErrID: Returns an errorcode # if the bError output is set.

eErrPos: If an error occurred and this refers to an individual element, then this output indicates the position (subindex of the element) at which an error first occurred.

stMDP_ModuleHeader: The header information from the read MDP module is displayed at this output in the form of the structure ST_MDP_ModuleHeader.

stMDP_ModuleContent: The information from Table 1 of the read MDP module is displayed at this output in the form of the structure ST_MDP_CPU.

the function block does not supply the CPU temperature; this can only be read via the generic sample program. The value "CPU temperature" is not supported by all IPCs.

The parameters and the function block are used at these points in the program:
The states of the state machine are listed as constants in order to enable simple adaptation of the program. The desired "State" value thus only needs to be centrally changed once. The statuses are defined as enumeration declarations in the PLC project in subfolder "DUTs" as DUT under the name "E-State".

The various areas of the state machine and their functions are explained below:
8.3 Querying the fan state (generic)

This sample illustrates access to the fan speed data via the generic function block FB_MDP_ReadElement [13]. It can be used to diagnose a fan failure (fan speed = 0).

A prerequisite is the presence of a fan. If no fan is available in the IPC, the IPC does not support the addressed module type, and the access attempt results in the error message 16#EC800002.

The contents of the FAN module are described in the Configuration Area of the IPC diagnostics. The generic function block FB_MDP_ReadElement is used for the access.

A description of the structure of this sample program can be found in the sample: Querying CPU data (generic) [36].

Sample: access via the generic function block FB_MDP_ReadElement

Enumeration definition

```plaintext
//** = simply adjust these lines if modifying code for own purposes

// central definition of state machine states
// (supports easy program modification)
[attribute 'qualified_only']
TYPE E_State :
{
  Idle,                  // idle state
  ReadFanSpeedInit,     //** initiate reading fan speed
```
ReadFanSpeedProcess  ///** process reading fan speed

END_TYPE

PROGRAM MAIN
VAR
// internal use
sAmsNetId : STRING := ''; ///** ADS Net ID (local = '')
eState : E_State; /// Enum with index for state machine
bStart : BOOL := TRUE; /// flag to trigger restart of statemachine
nData : UINT; /// data storage for unsigned integer
stMDP_Addr : ST_MDP_Addr; /// structure will include all address parameters
nModuleIndex : USINT := 0; ///** Fan index (no. of fan)

// FB instances
fbReadMDPElement : FB_MDP_ReadElement; /// instance of FB for reading MDP element

// results of execution
bError : BOOL; /// error flag (indicator: error occured)
nErrID : UDINT; /// last error ID
aFanSpeed : ARRAY[0..1] OF UINT; ///** buffer for speed of fans

END_VAR

Program code

// For an easy re-use of the following code for own purposes, parts of this sample program use
// "general" data names (and copy the results in specific variables after processing the code).
// Remark: Error 16#EC800002 means module type not supported (IPC does not provide this type of
// information, e.g. does not have fans)

CASE eState OF
E_State.Idle:
  IF bStart THEN
    bStart := FALSE;
    eState := E_State.ReadFanSpeedInit; ///** initiate first state
  END_IF
E_State.ReadFanSpeedInit:
  ///**
  stMDP_Addr.nArea := INT_TO_BYTE(eMDP_Area_ConfigArea); ///** set area address to "Config
  Area"
  stMDP_Addr.nTableId := 1; ///** table ID in module for "Fan properties"
  stMDP_Addr.nSubIdx := 1; ///** subindex in table ID for "Fan speed"
  fbReadMDPElement(bExecute := TRUE, /// Flag: trigger execution of FB
    eModuleType := eMDP_ModT_Fan, ///** desired module type = Fan
    stMDP_Addr := stMDP_Addr, /// MDP address structure. Dynamic module ID added
    iModIdx := nModuleIndex, ///** instance of desired module type (0 = first
    instance)
    pDstBuf := ADR(nData), /// buffer for storing data
    cbDstBufLen := SIZEOF(nData), /// length of buffer
    sAmsNetId := sAmsNetId, /// AMS Net ID
  ); ///** Note: fbReadMDPElement.tTimeOut must be > cycle

  eState := E_State.ReadFanSpeedProcess; ///** next state: process FB
E_State.ReadFanSpeedProcess:
  ///** process FB: request fan data
  fbReadMDPElement(bExecute := FALSE); /// Flag: Get execution state of FB

  IF NOT fbReadMDPElement.bBusy THEN /// FB executed?
    bError := TRUE;
    nErrID := fbReadMDPElement.nErrID; // store error id (16#ECA60105 = BIOS or HW does
    /// not support this data (here: mainboard data))
    eState := E_State.Idle; /// finish state machine
  ELSE
    bError := FALSE; /// turn off error flag
    aFanSpeed[nModuleIndex] := nData; ///** store fan speed in dedicated array
  END_IF

  IF nModuleIndex = 0 THEN ///** Current fan = fan 1?
    nModuleIndex := 1; ///** Read fan 2 (= second module instance) in next
    loop
  ELSE
    nModuleIndex := 0; ///** Read fan 1 (= first module instance) in next loop
  END_IF

  eState := E_State.ReadFanSpeedInit; ///** next state
END_IF
8.4 Reading IPC serial numbers

This sample illustrates access to the serial number of the IPCs and the serial number of the IPC’s mainboard.

- The serial number of the mainboard can be read via a subindex in module Mainboard in the Configuration Area of the IPC diagnostics. The general function block `FB_MDP_ReadElement [13]` is used for this purpose.
- The serial number of the IPC can be read via index 0xF9F0 in the Device Area of the IPC diagnostics. The general function block `FB_MDP_ReadIndex [12]` is used for this purpose.

Sample: querying the serial number of a Beckhoff IPC

**Enumeration definition**

```plaintext
//** = simply adjust these lines if modifying code for own purposes
/// central definition of state machine states
// (supports easy program modification)
{attribute 'qualified_only'}
TYPE E_State :
{
    Idle, // idle state
    ReadSnoMainboardInit, //** initiate reading serial number of mainboard
    ReadSnoMainboardProcess, //** process reading serial number of mainboard
    ReadSnoIPCInit, //** initiate reading serial number of IPC
    ReadSnoIPCPProcess //** process reading serial number of IPC
};
END_TYPE
```

**Variable Declaration**

```plaintext
PROGRAM MAIN
VAR
// internal use
sAmsNetId : STRING := ''; //** ADS Net ID (local = '')
eState : E_State; // Enum with index for state machine
bStart : BOOL := TRUE; // flag to trigger restart of statemachine
sData : STRING; // data storage for string variable
stMDP_Addr : ST_MDP_Addr; // structure which will include all address parameters

// FB instances
fbReadMDPElement : FB_MDP_ReadElement; // instance of FB for reading MDP element
fbReadMDPIndex : FB_MDP_ReadIndex; // instance of FB for reading MDP index

// results of execution
bError : BOOL; // error flag (indicator: error occured)
nErrID : UDINT; // last error ID
sSerialNoMainboard : STRING; //** buffer for serial number of mainboard
sSerialNoIPC : STRING; //** buffer for serial number of IPC
END_VAR
```

**Program code**

```plaintext
// For an easy re-use of the following code for own purposes, parts of this sample program use
// "general" data names (and copy the results in specific variables after processing the code).
CASE eState OF
    E_State.Idle:
        IF bStart THEN
            bStart := FALSE;
            eState := E_State.ReadSnoMainboardInit; //** initiate first state
        END_IF
    //
    ** read serial number of mainboard ******************************************
        E_State.ReadSnoMainboardInit: //** trigger FB: request mainboard serial number
            sData := ''; // clear data buffer
            sSerialNoMainboard := ''; //** clear buffer for serial number of mainboard
```

END_CASE
stMDP_Addr.nArea := INT_TO_BYTE(eMDP_Area_ConfigArea);  //
** set area address to "Config Area"
stMDP_Addr.nTableId := 1;     /// table ID in index for "mainboard information"
stMDP_Addr.nSubIdx := 2;     //
** subindex in table ID for "serial number"

xfbReadMDPElement ( 
  bExecute := TRUE,     // Flag: trigger execution of FB
  eModuleType := eMDP_ModT_Mainboard,   //
  ** desired module type / index = Mainboard
  stMDP.Addr := stMDP.Addr,   // MDP address structure. Dynamic module ID will be
  // added internally.
  iModIdx := 0,  //
  ** Instance of desired module type (default: 0 = first instance)
  pDstBuf := ADR(sData),     // buffer for storing data
  cbDstBufLen := SIZEOF(sData),  // length of buffer
  sAmsNetId := sAmsNetId,   // AMS Net ID
);  

  eState := E_State.ReadSnoMainboardProcess;  /// next state: process FB
E_State.ReadSnoMainboardProcess:  /// process FB: request mainboard serial number
  fbxReadMDPElement(bExecute := FALSE);  /// Flag: Get execution state of FB
FW NOT fbxReadMDPElement.bBusy THEN  // FB executed?
FW fbxReadMDPElement.bError THEN  // Error?
  bError := TRUE;   // set error flag
  nErrID := fbxReadMDPElement.nErrID;  // store error id (16#ECA60105 = BIOS or HW does
  // not support this data (here: mainboard data))
ELSE  // finish state machine
  eState := E_State.Idle;  // set parameters for next steps
  bError := FALSE;  // turn off error flag
  sSerialNoMainboard := sData;  //
** store serial number of mainboard in dedicated variable
E_State := E_State.ReadSnoIPCInit;  /// next state
END_IF

E_State.ReadSnoIPCInit:  ///
** trigger FB: request single index in MDP Device Area, IPC serial number
FB xReadMDPIndex( 
  bExecute := TRUE,  // Flag: trigger execution of FB
  nIndex := 16#F9F0,  //** index: read serial number IPC (-
  > see docu 'MDP device area')
  nSubIndex := 0,  //
  ** first subindex (there is only one available for index 16#F9F0)
  pDstBuf := ADR(sData),   // buffer for storing serial number
  cbDstBufLen := SIZEOF(sData),   // length of buffer
  sAmsNetId := sAmsNetId,  // AMS Net ID
);  

  eState := E_State.ReadSnoIPCProcess;  /// next state: process FB
E_State.ReadSnoIPCProcess:  ///
** process FB: request single index in MDP Device Area, IPC serial number
  fbxReadMDPIndex(bExecute := FALSE);  /// flag: Get execution state of FB
FW NOT fbxReadMDPIndex.bBusy THEN  // FB executed?
FW fbxReadMDPIndex.bError THEN  // error?
  bError := TRUE;  // set error flag
  nErrID := fbxReadMDPIndex.nErrID;  // store error id (16#ECA60105 = BIOS or HW does
  // not support this data (here: IPC serial number))
ELSE  // finish state machine
  eState := E_State.Idle;  // set parameters for next steps
  bError := FALSE;  // turn off error flag
  sSerialNoIPC := sData;  //** store serial number of mainboard
  eState := E_State.Idle;  //
** set here next state if expanding the state machine
END_IF
END_CASE
Returning of the mainboard serial number instead of the IPC serial number

In older BIOS version (before Q4/2013) the serial number was not stored in the IPC BIOS. In these cases the return value is the serial number of the IPC mainboard. With older Beckhoff Automation Device Driver versions, the return value is also the serial number of the IPC mainboard. The serial number of the IPC mainboard can always be read via the mainboard module.

8.5 Setting the IP address

This sample illustrates write access to IPC diagnostics data via general function blocks. In the same way all other elements of the IPC diagnostics modules can be accessed.

Active network connection

Changing these settings requires an active network connection for the selected Ethernet/EtherCAT adapter. Without an active network connection no parameters can be (pre-)set.

Sample: setting the IP address of a Beckhoff IPC

Variable Declaration

```plaintext
PROGRAM MAIN
// First DHCP is set off and then the given new IP address is set.
// The program code is executed only one time. To restart the program set bStart TRUE again.
VAR
  bStart : BOOL := TRUE;
  nState : INT := 100;
  fbScan : FB_MDP_ScanModules;
  fbWrite : FB_MDP_Write;
  stMDPAddr : ST_MDP_Addr;
  bDHCP : BOOL;
  sIP : T_IPv4Addr := '174.18.3.154'; // the new ip address
  bError : BOOL;
  nErrID : UDINT;
END_VAR

Program code

CASE nState OF
  100: // idle
    IF bStart THEN
      bStart := FALSE;
      nState := 00;
    END_IF
  00: (* scan MDP module list for dyn.module id of NIC module *)
    fbScan(
      bExecute : = TRUE,
      nModuleType := eMDP_ModT_NIC,
      iModIdx := 0, (* index of NIC module / network port *)
      sAmsNetId := '',
    );
    nState := 01;
  01:
    fbScan( bExecute:= FALSE );
    IF NOT fbScan.bBusy THEN
      IF NOT fbScan.bError THEN
        stMDPAddr.nArea := INT_TO_BYTE(eMDP_Area_ConfigArea);
        stMDPAddr.nModuleId := fbScan.nDynModuleId;
        stMDPAddr.nTableId := 1;
        nState := 10;
      ELSE
        bError := TRUE;
        nErrID := fbScan.nErrID;
        nState := 00;
      END_IF
    END_IF
  10: // set DHCP off
    stMDPAddr.nSubIdx := 4; // DHCP
    bDHCP := FALSE;
```

TE1000
Version: 1.6
```
fWrite(
  bExecute := TRUE,
  stMDP_DynAddr := stMDPAddr,
  pSrcBuf := ADR(bDHCP),
  cbSrcBufLen := SIZEOF(bDHCP),
  sAmsNetId := ''
);

nState:= 11;

11:
  fbWrite( bExecute:= FALSE );
  IF NOT fbWrite.bBusy THEN
    IF NOT fbWrite.bError THEN
      nState := 20;
    ELSE
      bError := TRUE;
      nErrID := fbWrite.nErrID;
      nState := 10;
    END_IF
  END_IF
END_IF

20: // set new IP address
  stMDPAddr.nSubIdx := 2; // IP address
  fbWrite(
    bExecute := TRUE,
    stMDP_DynAddr := stMDPAddr,
    pSrcBuf := ADR(sIP),
    cbSrcBufLen := LEN(sIP),
    sAmsNetId := ''
  );
  nState := 21;

21:
  fbWrite( bExecute:= FALSE );
  IF NOT fbWrite.bBusy THEN
    IF NOT fbWrite.bError THEN
      nState := 100; (* NIC settings executed *)
    ELSE
      bError := TRUE;
      nErrID := fbWrite.nErrID;
      nState := 20;
    END_IF
  END_IF
END_IF
END_CASE
```