CAN

Worldwide Technical Support and Product Information
ni.com

Worldwide Offices
Visit ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

National Instruments Corporate Headquarters
11500 North Mopac Expressway  Austin, Texas 78759-3504  USA  Tel: 512 683 0100

For further support information, refer to the Technical Support and Professional Services appendix. To comment on National Instruments documentation, refer to the National Instruments Web site at ni.com/info and enter the Info Code feedback.

© 2005–2013 National Instruments. All rights reserved.
Important Information

Warranty
The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this document is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER’S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THEREFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner’s failure to follow the National Instruments installation, operation, or maintenance instructions; owner’s modification of the product; owner’s abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright
Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

National Instruments respects the intellectual property of others, and we ask our users to do the same. NI software is protected by copyright and other intellectual property laws. Where NI software may be used to reproduce software or other materials belonging to others, you may use NI software only to reproduce materials that you may reproduce in accordance with the terms of any applicable license or other legal restriction.

End-User License Agreements and Third-Party Legal Notices
You can find end-user license agreements (EULAs) and third-party legal notices in the following locations:

- Notices are located in the <National Instruments>\_Legal Information and <National Instruments> directories.
- EULAs are located in the <National Instruments>\Shared\MDF\Legal\license directory.
- Review <National Instruments>\_Legal Information.txt for more information on including legal information in installers built with NI products.

Trademarks
Refer to the NI Trademarks and Logo Guidelines at ni.com/trademarks for more information on National Instruments trademarks.

ARM, Keil, and µVision are trademarks or registered of ARM Ltd or its subsidiaries.

LEGO, the LEGO logo, WEDO, and MINDSTORMS are trademarks of the LEGO Group. ©2013 The LEGO Group.

TETRIX by Pitsco is a trademark of Pitsco, Inc. ©2013

FIELDBUS FOUNDATION™ and FOUNDATION™ are trademarks of the Fieldbus Foundation.

EtherCAT® is a registered trademark of and licensed by Beckhoff Automation GmbH.

CANopen® is a registered Community Trademark of CAN in Automation e.V.

DeviceNet™ and EtherNet/IP™ are trademarks of ODVA.

Go!, SensorDAQ, and Vernier are registered trademarks of Vernier Software & Technology. Vernier Software & Technology and vernier.com are trademarks or trade dress.

Xilinx is the registered trademark of Xilinx, Inc.

Taptite and Trilobular are registered trademarks of Research Engineering & Manufacturing Inc.

FireWire® is the registered trademark of Apple Inc.

Linix® is the registered trademark of Linus Torvalds in the U.S. and other countries.

Handle Graphics®, MATLAB®, Real-Time Workshop®, Simulink®, Stateflow®, and xPC TargetBox® are registered trademarks, and TargetBox™ and Target Language Compiler® are trademarks of The MathWorks, Inc.

Tektronix®, Tek, and Tektronix, Enabling Technology are registered trademarks of Tektronix, Inc.

The Bluetooth® word mark is a registered trademark owned by the Bluetooth SIG, Inc.

The ExpressCard™ word mark and logos are owned by PCMCIA and any use of such marks by National Instruments is under license.

The mark LabWindows is used under a license from Microsoft Corporation. Windows is a registered trademark of Microsoft Corporation in the United States and other countries.

Other product and company names mentioned herein are trademarks or trade names of their respective companies.
Members of the National Instruments Alliance Partner Program are business entities independent from National Instruments and have no agency, partnership, or joint-venture relationship with National Instruments.

**Patents**
For patents covering National Instruments products/technology, refer to the appropriate location: Help»Patents in your software, the patents.txt file on your media, or the National Instruments Patent Notice at ni.com/patents.

**Export Compliance Information**
Refer to the Export Compliance Information at ni.com/legal/export-compliance for the National Instruments global trade compliance policy and how to obtain relevant HTS codes, ECCNs, and other import/export data.

**WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS**

1. NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

2. IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED “SYSTEM FAILURES”). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS’ TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.
Contents

About This Manual
Related Documentation ......................................................... xiii

Activating Your Software
How Do I Activate My Software? ...................................................... xv
What is Activation? ........................................................................ xv
What is the NI Activation Wizard? .................................................. xv
What Information Do I Need to Activate? ...................................... xv
How Do I Find My Product Serial Number? ................................. xvi
What is a Computer ID? ............................................................... xvi
How Can I Evaluate NI Software? ................................................... xvi
Moving Software After Activation .............................................. xvii
Deactivating a Product ............................................................... xvii
Using Windows Guest Accounts .................................................. xvii

Chapter 1
Introduction
CAN Calibration Protocol (CCP) Overview ............................................. 1-2
CCP Protocol Version ................................................................. 1-2
Universal Measurement and Calibration Protocol (XCP) Overview ........... 1-3
XCP Protocol Version ................................................................. 1-3
Measurement and Calibration Databases .......................................... 1-4
ECU Measurements ................................................................. 1-4
ECU Characteristics ................................................................. 1-4

Chapter 2
Installation and Configuration
Installation .................................................................................. 2-1
License Management Overview .................................................. 2-1
Activate ECU M&C Toolkit ....................................................... 2-2
Terms ......................................................................................... 2-3
Moving Software After Activation ............................................... 2-4
Volume License Program ............................................................ 2-4
Online Activation ........................................................................ 2-4
Home Computer Use .................................................................... 2-4
Privacy Policy ................................................................................ 2-4
Contents

LabVIEW Real-Time (RT) Configuration ................................................................. 2-5
PXI System ........................................................................................................ 2-5
NI-CAN on PXI RT System .............................................................................. 2-5
NI-XNET on PXI RT System ............................................................................ 2-5
CompactRIO System ...................................................................................... 2-5
DOS Command Prompt ................................................................................... 2-6
Web Browsers .................................................................................................... 2-7
LabVIEW Real-Time Graphical File Transfer Utility ...................................... 2-7
LabVIEW ......................................................................................................... 2-9
Hardware and Software Requirements .............................................................. 2-10

Chapter 3
Application Development

Choose the Programming Language .................................................................. 3-1
  LabVIEW ........................................................................................................ 3-1
  LabWindows/CVI .......................................................................................... 3-1
  Visual C++ 6 ................................................................................................... 3-2
  Other Programming Languages ....................................................................... 3-3

Application Development on CompactRIO or R Series Using an NI 985x or NI 986x C Series Module ............................................................................................................. 3-4
Debugging An Application ................................................................................ 3-6
  NI I/O Trace .................................................................................................. 3-6
  CCP/XCP-Spy .............................................................................................. 3-6
    Saving Captured Communication Data .................................................... 3-8
    Capture Options ....................................................................................... 3-8
    Call History Depth ................................................................................... 3-8
    Capturing Data ........................................................................................... 3-8
    Selecting Which CCP and XCP Commands to View .................................. 3-8

Chapter 4
Using the ECU M&C API

Structure of the ECU M&C API .......................................................................... 4-1
  ECU M&C Channel Functions ....................................................................... 4-2
    What is an ECU Measurement? ................................................................... 4-2
    What is an ECU Characteristic? ................................................................. 4-2
  ECU M&C CCP and XCP Functions .............................................................. 4-3
Contents

Basic Programming Model ................................................................. 4-3
ECU Open ......................................................................................... 4-5
ASAM MCD 2MC Communication Properties for CCP or XCP with CAN 4-5
   CRO ID ....................................................................................... 4-5
   DTO ID ....................................................................................... 4-5
   Station Address ......................................................................... 4-5
   Baudrate ..................................................................................... 4-6
ASAM MCD 2MC Communication Properties for XCP with UDP or TCP 4-6
   IP Address or hostname ............................................................. 4-6
   Port number ................................................................................. 4-6
ECU Connect ...................................................................................... 4-6
ECU Disconnect ................................................................................ 4-7
ECU Close ......................................................................................... 4-7
Characteristic Read and Write ....................................................... 4-7
   Access Characteristics ............................................................... 4-7
   Characteristic Read ................................................................. 4-8
   Characteristic Write ................................................................. 4-8
Measurement Task ........................................................................... 4-9
   DAQ Initialize ............................................................................. 4-10
   DAQ Start Stop .......................................................................... 4-10
   DAQ Read .................................................................................. 4-11
   DAQ Write ................................................................................. 4-12
   DAQ Clear .................................................................................. 4-13
Memory Programming ....................................................................... 4-13
   Program Start ............................................................................. 4-14
   Clear Memory ............................................................................ 4-15
   Program ...................................................................................... 4-15
   Program Reset ............................................................................ 4-15
Optional Steps for the XCP Protocol ................................................ 4-15
Additional Programming Topics ..................................................... 4-16
Get Names ........................................................................................ 4-16
Set/Get Properties .......................................................................... 4-16
Generic CCP Functions .................................................................... 4-17
Generic XCP Functions .................................................................... 4-18
Seed and Key Algorithm ................................................................... 4-19
   Definition for Seed and Key Algorithm .................................... 4-19
   Seed and Key Example ............................................................. 4-20
   Checksum Algorithm .............................................................. 4-21
   Seed and Key and Checksum Algorithms for VxWorks Targets 4-23
Chapter 5
ECU M&C API for LabVIEW

Section Headings ................................................................. 5-1
Purpose ............................................................................ 5-1
Format .............................................................................. 5-1
Input and Output ........................................................... 5-1
Description ....................................................................... 5-1

List of VIs ........................................................................ 5-1
MC Build Checksum.vi .................................................... 5-5
MC Calc Checksum.vi ....................................................... 5-8
MC CCP Action Service.vi ............................................. 5-11
MC CCP Diag Service.vi ............................................... 5-13
MC CCP Get Active Cal Page.vi .................................. 5-15
MC CCP Get Result.vi .................................................... 5-17
MC CCP Get Session Status.vi .................................... 5-19
MC CCP Get Version.vi ................................................ 5-21
MC CCP Move Memory.vi ............................................. 5-23
MC CCP Select Cal Page.vi .......................................... 5-25
MC CCP Set Session Status.vi ..................................... 5-27
MC Characteristic Read.vi .......................................... 5-29
MC Characteristic Read Single Value.vi ......................... 5-31
MC Characteristic Write.vi ......................................... 5-33
MC Characteristic Write Single Value.vi ...................... 5-35
MC Clear Memory.vi ...................................................... 5-37
MC Conversion Create.vi ............................................. 5-39
MC DAQ Clear.vi .......................................................... 5-41
MC DAQ Initialize.vi .................................................... 5-43
MC DAQ List Initialize.vi ............................................ 5-46
MC DAQ Read.vi .......................................................... 5-49
MC DAQ Start Stop.vi ................................................... 5-55
MC DAQ Write.vi .......................................................... 5-57
MC Database Close.vi ................................................. 5-60
MC Database Create.vi ................................................. 5-62
MC Database Open.vi .................................................. 5-64
MC Double to Text.vi ..................................................... 5-66
MC Download.vi ........................................................ 5-68
MC ECU Close.vi .......................................................... 5-70
MC ECU Connect.vi ..................................................... 5-72
MC ECU Create.vi ......................................................... 5-74
MC ECU Deselect.vi ...................................................... 5-79
MC ECU Disconnect.vi ................................................. 5-81
MC ECU Open.vi ........................................................... 5-83
MC ECU Select.vi ........................................................... 5-87
Chapter 6

ECU M&C API for C

Section Headings ........................................................................................................... 6-1
Purpose ......................................................................................................................... 6-1
Format ........................................................................................................................... 6-1
Input and Output ........................................................................................................ 6-1
Description .................................................................................................................. 6-1
List of Data Types ........................................................................................................ 6-1
List of Functions ........................................................................................................... 6-2
  mcBuildChecksum ..................................................................................................... 6-6
  mcCalculateChecksum ............................................................................................. 6-10
  mcCCPActionService ............................................................................................... 6-12
  mcCCP diagService .................................................................................................. 6-14
  mcCCP Get Active Cal Page ..................................................................................... 6-16
  mcCCP Get Result .................................................................................................... 6-17
  mcCCP Get Session Status ....................................................................................... 6-18
  mcCCP Get Version ................................................................................................. 6-19
  mcCCP Move Memory .............................................................................................. 6-20
  mcCCP Select Cal Page ............................................................................................ 6-22
  mcCCP Set Session Status ....................................................................................... 6-23
Contents

mcCharacteristicRead ................................................................. 6-25
mcCharacteristicReadSingleValue .............................................. 6-26
mcCharacteristicWrite ............................................................... 6-28
mcCharacteristicWriteSingleValue ............................................ 6-29
mcClearMemory ........................................................................ 6-31
mcConversionCreate ................................................................. 6-32
mcDAQClear ............................................................................... 6-34
mcDAQInitialize ......................................................................... 6-35
mcDAQListInitialize ................................................................. 6-38
mcDAQRead ............................................................................... 6-40
mcDAQReadTimestamped ............................................................ 6-43
mcDAQStartStop ....................................................................... 6-46
mcDAQWrite ............................................................................ 6-48
mcDatabaseClose ...................................................................... 6-50
mcDatabaseOpen ........................................................................ 6-51
mcDatabaseOpenEx ................................................................. 6-52
mcDoubleToText ....................................................................... 6-53
mcDownload ............................................................................. 6-55
mcECUCConnect ......................................................................... 6-57
mcECUCreate ............................................................................ 6-58
mcECUDeselect ......................................................................... 6-62
mcECUDisconnect ....................................................................... 6-63
mcECUSelectEx .......................................................................... 6-64
mcEventCreate .......................................................................... 6-67
mcGeneric .................................................................................. 6-68
mcGetNames ............................................................................. 6-70
mcGetNamesLength .................................................................... 6-73
mcGetProperty ............................................................................ 6-76
mcMeasurementCreate .............................................................. 6-96
mcMeasurementRead ............................................................... 6-98
mcMeasurementWrite ............................................................... 6-99
mcProgram ............................................................................... 6-100
mcProgramReset ......................................................................... 6-102
mcProgramStart ......................................................................... 6-103
mcSetProperty ........................................................................... 6-104
mcStatusToString ....................................................................... 6-116
mcTextToDouble ........................................................................ 6-118
mcUpload ................................................................................... 6-120
mcXCPCopyCalPage ..................................................................... 6-122
mcXCPGetCalPage ....................................................................... 6-124
mcXCPGetId ............................................................................... 6-126
mcXCPGetStatus ........................................................................ 6-128
mcXCPProgramPrepare ............................................................. 6-132
mcXCPProgramVerify .................................................................. 6-134

ECU M&C Toolkit User Manual

ni.com
Appendix A
Summary of the CCP Standard

Appendix B
Technical Support and Professional Services

Glossary

Index
About This Manual

This manual provides instructions for using the ECU Measurement & Calibration (ECU M&C) Toolkit. It contains information about installation, configuration, and troubleshooting, and also contains ECU M&C function references for LabVIEW-based and C-based APIs.

Use the ECU M&C Toolkit Installation Guide in the jewel case of the program CD to install the ECU M&C Toolkit software. Use this manual to learn the basics of ECU Measurement and Calibration, as well as how to develop an application.

Related Documentation

The following documents contain information that you may find helpful as you read this manual:


• **CiA Draft Standard 102, Version 2.0, CAN Physical Layer for Industrial Applications**

• **CAN Calibration Protocol Specification, Version 2.1, ASAP Arbeitskreis zur Standardisierung von Applikationssystemen Standardization of Application/Calibration Systems task force**

• **Interface Specification Interface 2 (ASAM MCD 2MC/ASAP2) Version 1.51 Release 2003-03-11, Applications Systems Standardization Working Group**

• **XCP Version 1.0, The Universal Measurement and Calibration Protocol Family, Association for Standardization of Automation and Measuring Systems:**
  – **Part 1, Overview**
  – **Part 2, Protocol Layer Specification**
  – **Part 3, XCP on CAN - Transport Layer Specification**
  – **Part 3, XCP on Ethernet - Transport Layer Specification**
  – **Part 4, Interface Specification**

• **NI-CAN Hardware and Software Manual**
Activating Your Software

This section describes how to use the NI Activation Wizard to activate your software.

How Do I Activate My Software?

Use the NI Activation Wizard to obtain an activation code for your software. You can launch the NI Activation Wizard two ways:

- Launch the product and choose to activate your software from the list of options presented.
- Launch NI License Manager by selecting Start»All Programs»National Instruments»NI License Manager. Click the Activate button in the toolbar.

Notes If your software is a part of a Volume License Agreement (VLA), contact your VLA administrator for installation and activation instructions.

NI software for Mac OS X and Linux operating systems does not require activation.

What is Activation?

Activation is the process of obtaining an activation code to enable your software to run on your computer. An activation code is an alphanumeric string that verifies the software, version, and computer ID to enable features on your computer. Activation codes are unique and are valid on only one computer.

What is the NI Activation Wizard?

The NI Activation Wizard is a part of NI License Manager that steps you through the process of enabling software to run on your machine.

What Information Do I Need to Activate?

You need your product serial number, user name, and organization. The NI Activation Wizard determines the rest of the information. Certain activation methods may require additional information for delivery. This information is used only to activate your product. Complete disclosure of
the National Instruments software licensing information privacy policy is available at ni.com/activate/privacy. If you optionally choose to register your software, your information is protected under the National Instruments privacy policy, available at ni.com/privacy.

**How Do I Find My Product Serial Number?**

Your serial number uniquely identifies your purchase of NI software. You can find your serial number on the Certificate of Ownership included in your software kit. If your software kit does not include a Certificate of Ownership, you can find your serial number on the product packing slip or on the shipping label.

If you have installed a previous version using your serial number, you can find the serial number by selecting the Help> About menu item within the application or by selecting your product within NI License Manager (Start> All Programs> National Instruments> NI License Manager). You can also contact your local National Instruments branch.

**What is a Computer ID?**

The computer ID contains unique information about your computer. National Instruments requires this information to enable your software. You can find your computer ID through the NI Activation Wizard or by using NI License Manager, as follows:

1. Launch NI License Manager by selecting Start> All Programs> National Instruments> NI License Manager.
2. Click the Display Computer Information button in the toolbar.

For more information about product activation and licensing, refer to ni.com/activate.

**How Can I Evaluate NI Software?**

You can install and run most NI application software in evaluation mode. This mode lets you use a product with certain limitations, such as reduced functionality or limited execution time. Refer to your product documentation for specific information on the product’s evaluation mode.
Moving Software After Activation

To transfer your software to another computer, install and activate it on the second computer. You are not prohibited from transferring your software from one computer to another and you do not need to contact or inform NI of the transfer. Because activation codes are unique to each computer, you will need a new activation code. Refer to How Do I Activate My Software? to acquire a new activation code and reactivate your software.

Deactivating a Product

To deactivate a product and return the product to the state it was in before you activated it, right-click the product in the NI License Manager tree and select Deactivate. If the product was in evaluation mode before you activated it, the properties of the evaluation mode may not be restored.

Using Windows Guest Accounts

NI License Manager does not support Microsoft Windows Guest accounts. You must log in to a non-Guest account to run licensed NI application software.
Introduction

The ECU Measurement and Calibration (ECU M&C) Toolkit contains a development system for an electronic control unit (ECU) based on existing ASAM standards. The function set of the ECU M&C Toolkit enables engineers to optimize and verify the functionality of electronic controller devices. Most ECUs interact with other ECUs, external sensors, and actuators in a Controller Area Network (CAN). During the development and verification phase of an ECU, engineers access the ECU for acquired data (Measurement), or to adjust parameters inside the ECU itself (calibration). Since the bandwidth and number of identifiers for a CAN network are limited the Association for Standardization of Automation and Measuring Systems (ASAM e.V.) has specified the CAN Calibration Protocol (CCP), a protocol layer based on CAN, to access the measurement and calibration data in an ECU.

To build on the functionality of the CAN Calibration Protocol (CCP), ASAM defined the new protocol specification XCP (Universal Measurement and Calibration Protocol) which can be considered an improved and generalized version of CCP. The X represents the various transportation layers used by the members of the XCP protocol family—for instance, XCP on CAN, XCP on TCP/IP, XCP on UDP/IP, XCP on USB, etc.

The ECU M&C Toolkit is particularly suited to the automotive industry and their component suppliers. It provides a function set that can be used in the development or verification phase of an ECU. Access to the data inside an ECU takes place based on information stored in an ASAM MCD 2MC (*.A2L) database file provided by the ECU supplier. Selecting each signal by its name provides convenient access to the data inside an ECU. The ECU M&C Toolkit uses CCP and XCP as the fundamental communication protocols and to support ECU database (*.A2L) files. You can easily switch between the CCP and XCP protocol layers through software.
CAN Calibration Protocol (CCP) Overview

The CAN Calibration Protocol is a CAN-based master-slave protocol for calibration and data acquisition. A single master device (host) can be connected to one or more slave devices. The host must establish a logical point-to-point connection to the slave device before the slave device may accept commands from the host. The slave device must acknowledge each command received from the host within a specified time after the connection between host and slave has been established.

CCP defines two function sets—one for control/memory transfer and one for data acquisitions that are independent of each other and may run asynchronously. The control commands are used to carry out functions in the slave device and may use the slave to perform tasks on other devices. The data acquisition commands are used for continuous data acquisition from a slave device. The devices continuously transmit internal data according to a list that has been configured by the host. Data acquisition is initiated by the host, then executed by the slave device, and may be based on a fixed sampling rate or be event-driven.

The communication of controllers with a master device through CCP is based on the CAN 2.0B standard (11-bit and 29-bit identifier), which includes 2.0A (11-bit identifier) for data acquisition from the controllers, memory transfers to the controllers, and control functions in the controllers for calibration.

The ECU M&C Toolkit abstracts the CCP communication layer so that it is transparent to the user. For most cases it is sufficient that the underlying CCP communication is handled by the toolkit kernel itself. Nevertheless, the ECU M&C Toolkit offers direct access to the low level CCP commands if a non-standard timing behavior or independent user defined command sequence is needed.

CCP Protocol Version

The ECU M&C Toolkit supports the CAN Calibration Protocol specification, version 2.1.
Universal Measurement and Calibration Protocol (XCP) Overview

The Universal Measurement and Calibration Protocol (XCP) is a single-master/single-slave protocol for calibration and data acquisition based on various transport layers. Communication is always initiated by the XCP master. An XCP slave must respond to requests from the master within a specified time. The XCP protocol uses a soft master/slave principle: once the master establishes a communication channel with the slave, the slave can send certain messages (Events, Service Requests and Data Acquisition messages) autonomously. In addition, the master sends Data Stimulation messages without expecting a direct response from the slave.

The XCP builds a continuous, logical, unambiguous point-to-point connection with 1 specific slave when establishing a communication channel. The XCP slave cannot handle multiple connections. The master is not allowed to broadcast XCP messages to multiple slaves at the same time. The identification parameters of the Transport Layer (for instance, CAN identifiers on CAN) must be chosen in such a way that they build independent and unambiguously distinguishable communication channels.

The ECU M&C Toolkit abstracts the XCP communication layer so that it is transparent to the user. For most cases it is sufficient that the underlying XCP communication is handled by the toolkit kernel. Nevertheless, the ECU M&C Toolkit offers direct access to the low level XCP commands if a non-standard timing behavior or independent user defined command sequence is required.

XCP Protocol Version

The ECU M&C Toolkit supports the XCP Calibration Protocol Specification, version 1.0.

For further information related to the XCP protocol, refer to the XCP Calibration Protocol Specification, version 1.0, The Universal Measurement and Calibration Protocol Family, Part 1, by ASAM e.V.
Measurement and Calibration Databases

The ASAP description file (ASAP2 or ASAM MCD 2MC) is used to describe the ECU internal memory configuration. An ASAM MCD 2MC description file with the file extension .A2L contains information and access locations for the relevant data objects in the ECU, such as:

- Project relevant information
- ECU data structure
- Conversion procedures for representation in physical units
- Descriptions of the available Measurement channels inside the ECU
- Descriptions of the available Characteristics inside the ECU
- Descriptions of how to access the ECU over CAN

Note: Use of the ECU M&C Toolkit requires an existing ASAM MCD 2MC database file. These files can be generated by various third-party utilities. A database editor for ASAM MCD 2MC databases is not part of the ECU M&C Toolkit.

ECU Measurements

The ECU M&C Toolkit provides the user access to ECU internal physical values defined by their names in the ASAM MCD 2MC database file. Based on this information, the ECU M&C Toolkit communicates through CCP or XCP to the ECU. A DAQ (data acquisition) list can be set up, which sends ECU internal data synchronously or asynchronously to the CCP or XCP master. The ECU M&C Toolkit provides a way to configure several Measurement channels into a single Measurement task. The term task refers to a list of measurements (channels) read or written together. A common use of the task concept is to read DAQ channels available on the ECU.

ECU Characteristics

ECU Characteristics are maps of ECU internal variables, which may be used as calibration information or set-point information. The ECU memory content of Characteristics can be read or even changed with the help of the ECU M&C Toolkit.
Installation and Configuration

This chapter explains how to install and configure the ECU M&C Toolkit.

Installation

This section discusses the installation of the ECU M&C Toolkit for Microsoft Windows.

Note You need administrator rights to install the ECU M&C Toolkit on your computer.

1. Insert the ECU M&C Toolkit CD into the CD-ROM drive.
2. Open Windows Explorer.
3. Access the CD-ROM drive.
4. Double-click on autorun.exe. This will launch the software interface.
5. Start the installation. The installation program will guide you through the rest of the installation process.
6. When installation is complete, the National Instruments License Manager will launch automatically to activate your license.

License Management Overview

License management is the process of controlling access to products based on an explicit license agreement. The ECU M&C Toolkit requires an activated license in order to launch, so a license must be acquired and activated before the product can be used. The activation process involves using the Activation Wizard to send the following information to National Instruments:

- The product you are activating: ECU Measurement and Calibration Toolkit
- The serial number of the product
- The version of the product
- Your name
Your organization
A computer ID that uniquely identifies your system

National Instruments uses this information to generate an activation code, which is used to activate the ECU M&C Toolkit on your system. National Instruments does not use this information for any other purpose. Refer to the Privacy Policy section for information on the National Instruments privacy policy regarding your personal information.

The Activation Wizard offers a variety of options you can use to obtain an activation code from National Instruments, including an automatic option through an Internet connection, or through email, by telephone, or by fax.

Activate ECU M&C Toolkit

The ECU M&C Toolkit must be activated before using it, in accordance with its license agreement. To activate the ECU M&C Toolkit, you must first purchase a license. For information on purchasing licenses, contact your local National Instruments sales representative or visit ni.com.

Once you have purchased a license, you can activate your product using the Activation Wizard. Activation is simple and you can activate your software 24 hours a day, 7 days a week.

Complete the following steps to activate the ECU M&C Toolkit.

1. Locate your serial number.
   Your serial number uniquely identifies your purchase of NI software. You can find it on the Certificate of Ownership included in your software kit. If you subscribe to NI Developer Suite or Academic Software Solutions, use the original serial number you received with your initial purchase.

2. Install your software.
3. Launch the License Activation Wizard.
   If you installed your software for the first time and the installer did not launch the License Activation Wizard for you, perform the following steps:
   a. Launch the NI License Manager by selecting Start»Programs»National Instruments»NI License Manager.
   b. Click the Activate button on the toolbar.
      The wizard will guide you through the activation process.
4. Save your activation code for future use (optional).
   You can reactivate your software at any time. The activation wizard provides you with the option to receive an email confirmation of your activation code. To apply this activation code in the future, launch the Activation Wizard and choose to apply a 20-character activation code. If you reinstall your software on the same computer, the same activation code will work.
   For more information on activation, refer to your product documentation, or visit ni.com/activate.

   National Instruments uses activation to better support evaluation of our software, to enable additional software features, and to support license management in large organizations. To find out more about National Instruments software licensing, visit ni.com/activate to find frequently asked questions, resources, and technical support.

**Terms**

<table>
<thead>
<tr>
<th><strong>Table 2-1. Definition of Activation Terms</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Serial Number</strong></td>
</tr>
<tr>
<td><strong>Computer ID or Device ID</strong></td>
</tr>
<tr>
<td><strong>Activation Code</strong></td>
</tr>
</tbody>
</table>
Moving Software After Activation

To transfer your software to another computer, install and reactivate it on the second computer. You are not prohibited from transferring your software from one computer to another. Because activation codes are unique to each computer, you will need a new activation code. Follow the steps on the previous page to acquire a new activation code and reactivate your software.

Volume License Program

National Instruments offers volume licenses through the NI Volume License Program. The NI Volume License Program makes managing software licenses and maintenance easy. For more information, refer to ni.com/vlp.

Online Activation

Activation is available on ni.com/activate 24 hours a day, 7 days a week. You can retrieve an activation code from any computer that has an Internet connection. NI does not require that the computer on which you run NI software have Internet or email access.

Home Computer Use

National Instruments permits you to use this software at home. Refer to the NI License Manager help file or the software end-user license agreement in the installer or online at ni.com/legal/license for more information.

Privacy Policy

National Instruments respects your privacy. For more information about the National Instruments activation information privacy policy, go to ni.com/activate/privacy.

Upon successful activation, you can use the product immediately.

Note  If the ECU M&C Toolkit was in use before you began the activation process, you may need to restart it for the change to take effect.

Tip  In the NI License Manager, products that have not been activated are denoted either by a yellow stoplight or a red stoplight, depending whether the product is in evaluation mode or is unusable. Activated products are denoted by a green stoplight.
LabVIEW Real-Time (RT) Configuration

LabVIEW Real-Time (RT) combines easy-to-use LabVIEW programming with the power of real-time systems.

PXI System

When you use a National Instruments PXI controller as a LabVIEW RT system, you can install a PXI CAN or PXI XNET card and use the ECU M&C Toolkit to develop real-time applications for CCP or XCP. As with any other NI product for LabVIEW RT, you must download the ECU M&C Toolkit software to the LabVIEW RT system using the Remote Systems branch in MAX. For more information, refer to the LabVIEW RT documentation.

After installing the PXI CAN cards and downloading the NI-CAN or NI-XNET and ECU M&C Toolkit software to the LabVIEW RT system, you need to verify the installation.

NI-CAN on PXI RT System

Within the MAX Tools menu, select NI-CAN»RT Hardware Configuration. The RT Hardware Configuration tool provides features similar to Devices and Interfaces on the local system. Use the RT Hardware Configuration tool to self-test the CAN cards and assign an interface name to each physical CAN port.

NI-XNET on PXI RT System

After you install the PXI XNET cards and download the NI-XNET software to the LabVIEW RT system, you need to verify the installation. Find your PXI target device in MAX under Network Devices and expand the tree. Browse to Devices and Interfaces and open the NI-XNET Devices group. Perform a self-test for all installed NI-XNET devices. On the RT target, you can configure your NI-XNET hardware the same way as on the local system.

CompactRIO System

After you have installed the CompactRIO CAN modules and downloaded the NI-RIO and ECU M&C Toolkit software, you need to enable the CompactRIO Reconfigurable Embedded Chassis for use in LabVIEW. For more information, refer to the MAX help.
Chapter 2  Installation and Configuration

**Note** You can use the ECU M&C Toolkit with LabVIEW 2009 or newer on CompactRIO Systems only.

To use the ECU M&C Toolkit on the LabVIEW RT system, you must also download the ASAM MCD 2MC database file to the RT target. The LabVIEW Real-Time Engine that runs on the PXI LabVIEW Real-Time controller supports a File Transfer Protocol (FTP) server. You can access the LabVIEW RT target FTP server using any standard FTP utility for transferring files to and from the hard drive or compact flash. The following sections demonstrate how to transfer files from and to your LabVIEW Real-Time target using various FTP clients.

**DOS Command Prompt**

You can run a native FTP client from the DOS command prompt on a Windows PC. To open the FTP client, click **Start»Run** to open the user-command dialog box. Type `command`, and click **Enter**. This opens a window with a DOS prompt.

Then use the following table to enter a sequence of commands that may be used to access the FTP server of your RT target.

**Note** \( w.x.y.z \) represents the IP address of the RT target in this document.

<table>
<thead>
<tr>
<th>Command</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ftp</code></td>
<td>Open a connection to the FTP server.</td>
</tr>
<tr>
<td><code>open w.x.y.z</code></td>
<td>Enter your username and password here or press the Enter key twice if these security settings have not been applied.</td>
</tr>
<tr>
<td><code>(username)</code></td>
<td></td>
</tr>
<tr>
<td><code>(password)</code></td>
<td></td>
</tr>
<tr>
<td><code>help</code></td>
<td>View a list of commands.</td>
</tr>
<tr>
<td><code>cd ni-rt\system\www</code></td>
<td>Change to the desired directory.</td>
</tr>
<tr>
<td><code>dir</code></td>
<td>View the files present.</td>
</tr>
<tr>
<td><code>get index.htm c:\index.htm</code></td>
<td>Copy the file.</td>
</tr>
<tr>
<td><code>cd \</code></td>
<td>Change directory back to the root (c:).</td>
</tr>
<tr>
<td><code>cd d:</code></td>
<td>Change directories to the external compact flash.</td>
</tr>
</tbody>
</table>
Web Browsers

You can also use Internet Explorer or Netscape Navigator to ftp files to and from the controller. This is an easier method of transfer, since there is no need to learn ftp commands—instead the files are simply copied and pasted as they would be in a Windows Explorer window. The disadvantage of this method is that Internet Explorer sometimes caches old information, so you will need to refresh occasionally.

If w.x.y.z is the IP address of your RT target, open Internet Explorer to access the hard drive or internal compact flash, or type the following in the address field:

```
ftp://w.x.y.z/
```

If a username and password are required, then use the following format:

```
ftp://username:password@w.x.y.z/
```

To access the external compact flash, open Internet Explorer and type the following in the address field:

```
ftp://w.x.y.z/d:
```

To enter a directory, double-click on its icon. Right-click on a file or folder and choose cut, copy, paste or delete to perform those actions.

### LabVIEW Real-Time Graphical File Transfer Utility

LabVIEW Real-Time Module versions 7.0 and later include a File Transfer Utility that can be used to access your RT target. This method helps you avoid the caching problem encountered when using web browsers. You can find this utility in the Measurement and Automation Explorer (MAX). To open the utility, right-click on the desired RT target under the Remote Systems list and choose File Transfer, as shown in Figure 2-1.
At this point, you are prompted for a username and password. If these security features have not been enabled, check the Anonymous Login box as shown in Figure 2-2.

The upper section of the utility interface shows the current directory and contents on the remote RT target, while the lower section gives information for the host or local machine. To copy a file (TestECU.a2l, for instance) to the RT target, complete the following steps, referring to Figure 2-3 for details.

1. In the Current Directory section, navigate through the tree structure to the System folder.
2. In the local directory section, navigate through the tree structure to the location of the file you want to transfer and highlight the file.
3. Click the To Remote button to copy the file.
Figure 2-3. Transferring Files With the FTP Utility

LabVIEW

You also can use LabVIEW to programmatically access the FTP server of a LabVIEW Real-Time target.

The **DataSocket Read** function has the ability to read raw text, tabbed text, and .wav files from an FTP server. For more information on this, refer to the *LabVIEW User Manual*.

The **LabVIEW Internet Developers Toolkit** allows you to send files or raw data to an FTP server, as well as sending emails and adding security to your web-based applications.
Hardware and Software Requirements

You can use the ECU M&C Toolkit on the following hardware:

- National Instruments NI-CAN hardware Series 1 or 2 with the NI-CAN driver software version 2.3 or later installed.
- National Instruments NI-XNET hardware with the NI-XNET driver software version 1.0 or later installed.
- National Instruments CompactRIO or R Series Multifunction RIO hardware and the NI 9853 or NI 9852 CompactRIO CAN modules.

**Note**  You can use the ECU M&C Toolkit with LabVIEW 2009 or newer on CompactRIO systems or National Instruments R Series Multifunction RIO hardware.
Application Development

This chapter explains how to develop an application using the ECU M&C API.

Choose the Programming Language

The programming language you use for application development determines how to access the ECU M&C Toolkit APIs.

LabVIEW

ECU M&C Toolkit functions and controls are available in the LabVIEW palettes. In LabVIEW, the ECU M&C Toolkit palette is located:

- Within the All Functions palette for LabVIEW 7.1
- Within the Addons palette for LabVIEW 8.0 and 8.1

The reference for each ECU M&C Toolkit API function is in Chapter 5, ECU M&C API for LabVIEW. To access the reference for a function from within LabVIEW, press <Ctrl-H> to open the Help window, click the appropriate ECU M&C function, and then follow the link. The ECU M&C Toolkit software includes a full set of examples for LabVIEW. These examples teach programming basics as well as advanced topics. The example help describes each example and includes a link you can use to open the VI.

LabWindows/CVI

Within LabWindows™/CVI™, the ECU M&C Toolkit function panel is in Libraries»ECU Measurement and Calibration Toolkit. Like other LabWindows/CVI function panels, the ECU M&C Toolkit function panel provides help for each function and the ability to generate code. The reference for each API function is located in Chapter 6, ECU M&C API for C. You can access the reference for each function directly from within the function panel. The header file for the ECU M&C Toolkit APIs is niemc.h. The library for the ECU M&C Toolkit APIs is niemcc.lib.
The toolkit software includes a full set of examples for LabWindows/CVI. The examples are installed in the LabWindows/CVI directory under samples\ecumc. Each example provides a complete LabWindows/CVI project (.prj file).

A description of each example is provided in comments at the top of the .c file.

Visual C++ 6

The ECU M&C Toolkit software supports Microsoft Visual C/C++ 6. The header file for Visual C/C++ 6 is in the Program Files\National Instruments\Shared\ExternalCompilerSupport\C\include folder.

To use the ECU M&C API, include the niemc.h header file in the code, then link with the niemcc.lib library file.

The niemcc.lib library file is in the Program Files\National Instruments\Shared\ExternalCompilerSupport\C\lib32\msvc folder.

For C applications (files with a .c extension), include the header file by adding a #include to the beginning of the code, like this:
#include "niemc.h"

For C++ applications (files with a .cpp extension), define __cplusplus before including the header, like this:
#define __cplusplus
#include "niemc.h"

The __cplusplus define enables the transition from C++ to the C language functions.

The reference for each API function is in Chapter 6, ECU M&C API for C.

On Windows Vista (with Standard User Account), the typical path to the C examples folder is \Users\Public\Documents\National Instruments\ECU Measurement and Calibration Toolkit\Examples\MS Visual C.

On Windows XP/2000, the typical path to the C examples folder is \Documents and Settings\All Users\Documents\National Instruments\ECU Measurement and Calibration Toolkit\Examples\MS Visual C.
Each example is in a separate folder. A description of each example is in comments at the top of the .c file. At the command prompt, after setting MSVC environment variables (such as with MS vcvars32.bat), you can build each example using a command such as:

```
cl /I<HDir> measure.c <LibDir>\niemcc.lib
```

<HDir> is the folder where niemc.h can be found.

<LibDir> is the folder where niemcc.lib can be found.

### Other Programming Languages

The ECU M&C Toolkit software does not provide formal support for programming languages other than those described in the preceding sections. If the programming language provides a mechanism to call a Dynamic Link Library (DLL), you can create code to call ECU M&C Toolkit functions. All functions for the ECU M&C API are located in niemcc.dll. If the programming language supports the Microsoft Win32 APIs, you can load pointers to ECU M&C Toolkit functions in the application. The following text demonstrates use of the Win32 functions for C/C++ environments other than Visual C/C++ 6. For more detailed information, refer to Microsoft documentation.

The following C language code fragment illustrates how to call Win32 LoadLibrary to load the DLL for the ECU M&C API:

```c
#include <windows.h>
#include "niemc.h"
HINSTANCE NiMcLib = NULL;
NiMcLib = LoadLibrary("niemcc.dll");
```

Next, the application must call the Win32 GetProcAddress function to obtain a pointer to each ECU M&C Toolkit function that the application will use. For each function, you must declare a pointer variable using the prototype of the function. For the prototypes of each ECU M&C Toolkit function, refer to Chapter 6, *ECU M&C API for C*.

Before exiting the application, you must unload the ECU M&C Toolkit DLL as follows:

```
FreeLibrary (NiMcLib);
```
Chapter 3  Application Development

Application Development on CompactRIO or R Series
Using an NI 985x or NI 986x C Series Module

To run a project on an FPGA target with an NI 985x C Series module, you need an FPGA bitfile (.lvbitx). The FPGA bitfile is downloaded to the FPGA target on the execution host. A bitfile is a compiled version of an FPGA VI. FPGA VIs, and thus bitfiles, define the CAN, analog, digital, and pulse width modulation (PWM) inputs and outputs of an FPGA target. The ECU M&C Toolkit includes FPGA bitfiles for several FPGA targets. If your target is not included in the examples, you can use the examples as a template and adjust them based on your installed FPGA target.

The default bitfiles are sufficient for a basic ECU M&C application. However, in some situations you may need to modify the existing FPGA code or create a custom bitfile. For example, to use additional I/O on the FPGA target, you must add these I/O to the FPGA VI. You must install the LabVIEW FPGA Module to create these files.

Modify the FPGA VI according to the following guidelines:

- Do not modify, remove, or rename any block diagram controls and indicators named __CAN0 Rx Data, __CAN0 Rx Ready, __CAN0 Tx Data Frame, __CAN0 Tx Ready, __CAN0 Bit Timing, __CAN0 FPGA Is Running, __CAN0 Start, __CAN0 FIFO Full, or __CAN0 FIFO Empty. If you intend to use multiple CAN 985x modules on your FPGA, you need to duplicate and rename all controls and indicators accordingly.

- Do not modify the CAN read and write code except to filter CAN IDs on the receiving side to minimize the amount of CAN data transfers to the host.

- As you create controls or indicators, ensure that each control name is unique within the VI.

Refer to the LabVIEW FPGA Module documentation for more information about creating FPGA VIs and bitfiles for an FPGA target.

When using the ECU M&C Toolkit on CompactRIO with an NI 985x C Series module, the interface name is based on the bitfile you use and the interface name you set. For example, MyInterface@MyBitfile.lvbitx, CAN0@lvbitfile.lvbitx, or CAN0@mybitfile.lvbitx.
The interface name you use must be part of all parameters in the FPGA code for the CAN communication. Also, the ECU M&C Toolkit needs the interface name for correct functionality.

If you define the interface name to be CAN0, you must name the parameters as follows:
- __CAN0 Rx Data
- __CAN0 Rx Ready
- __CAN0 Tx Data Frame
- __CAN0 Tx Ready
- __CAN0 Bit Timing
- __CAN0 FPGA Is Running
- __CAN0 Start
- __CAN0 FIFO Full
- __CAN0 FIFO Ready

In addition, you need to set the name of the internally used FIFO to __CAN0 FIFO (the FIFO is set to U32, 1029 elements, target scoped, and block memory).

After recompiling your FPGA VI, copy the bitfile to the root directory of your CompactRIO controller and specify the bitfile in the interface name. Or copy the file to any location on the CompactRIO controller and specify an absolute path or path relative to the root for the bitfile.

If you are using an NI-XNET 986x C Series module on your CompactRIO target, you need to start an FPGA VI on the target before accessing the port with the ECU M&C Toolkit. Refer to the Getting Started with CompactRIO section in the NI-XNET Hardware and Software Manual for more information about compiling the FPGA VI. When the VI is running, you can access the NI 986x module as you would program on a Windows or PXI LabVIEW Real-Time target.
Debugging An Application

NI I/O Trace

The NI I/O Trace (formerly NI-Spy) tool monitors function calls to the ECU M&C API to aid in the debugging of an application. To launch this tool, open the Software branch of the MAX configuration tree, right-click NI I/O Trace, and select Launch NI I/O Trace.

If you have more than one National Instruments driver installed on your computer, you can specify which APIs you want to monitor at any time. By default, all installed APIs are enabled. To select the APIs to monitor, select Tools»Options, select the View Selection tab, and select the desired APIs under Installed API Choices.

CCP/XCP-Spy

The CCP/XCP-Spy tool monitors CCP and XCP protocol communication to aid in the debugging of an application. Launch this tool from the Start menu in Start»Programs»National Instruments»ECU Measurement and Calibration Toolkit»CCP and XCP Spy.
CCP/XCP-Spy is an application that monitors, records, and displays CCP and XCP communication commands and parameters called by your ECU M&C application using the CCP or XCP protocol. Use CCP/XCP-Spy to analyze your application's communication and to verify that the communication with your ECU slave is correct.

You can use this application on Windows only when the ECU M&C master also is running on Windows.

CCP/XCP-Spy may slow down the performance of your application, communication to your ECU slave, and the entire system. You should use CCP/XCP-Spy only while you are debugging or when performance is not critical.

For further information about the displayed CCP or XCP commands and parameters, refer to the ASAM XCP Part 2 Protocol Layer Specification or CAN Calibration Protocol Version 2.1 specification documents.
Saving Captured Communication Data

To save the information displayed in the CCP/XCP-Spy capture window, select **File>Save As**. In the dialog box that appears, select a name for the capture file. A `.xlg` extension usually is used for saving CCP/XCP-Spy capture information. The CCP/XCP-Spy log is stored in ASCII format, so you can view the `.xlg` file in any ASCII editor.

Capture Options

To view or modify the CCP/XCP-Spy capture options, select **Spy>Options**. By default, CCP/XCP-Spy displays 250 calls in the capture window.

Call History Depth

The call history depth reflects the maximum number of API calls that CCP/XCP-Spy can display. When the number of captured API calls exceeds the call history depth, only the most recent calls are kept.

Capturing Data

By default, capture is activated when you open CCP/XCP-Spy. When capture is off, the blue arrow (start button) is enabled. When capture is on, the red X (stop button) is enabled. To turn capture on, click the blue arrow button on the toolbar. To turn capture off, click the red button on the toolbar.

Selecting Which CCP and XCP Commands to View

You can specify which command you want to spy on at any time. By default, CCP/XCP commands are enabled. To select/deselect the CCP/XCP commands to spy on, select **Spy>Options**, then select the commands under **Capture**.

- **Commands**—Captures all CCP/XCP commands.
- **DAQ Messages**—Captures all DAQ list commands (ECU measurement commands).
- **STIM Messages**—Captures all STIM list (ECU slave stimulation commands).
This chapter helps you get started with the ECU M&C API.

**Structure of the ECU M&C API**

The ECU M&C API is divided into three main function categories, the high-level Channel-based functions, and the generic low-level CCP and XCP functions. The ECU M&C Channel functions provide an easy way to access ECU internal data through named channels. The ECU M&C CCP functions provide direct access to the CCP commands on a very low programming level. The ECU M&C XCP functions provide direct access to the XCP commands on a very low programming level. Figure 4-1 outlines the three function categories.
**ECU M&C Channel Functions**

With the ECU M&C Channel functions there are a number of ways to access memory content in an ECU. The starting point is always the creation of a database task, which is the link to a valid ASAM MCD 2MC database file (`.A2L` file), and the selection of the protocol (CCP or XCP). With the database task reference it is possible to create an ECU task reference, which links to the selected ECU. Depending on the application scenario, the ECU task reference can be used for the following:

- Creation of a Measurement task to measure ECU internal data continuously or on demand
- Direct read/write of 0- to 2-dimensional Characteristics
- Read/write of single Measurement values on demand

**What is an ECU Measurement?**

An ECU Measurement, called *ECU Data Acquisition (DAQ)* in the CCP and XCP specifications, is a definition of specific procedures and CAN messages sent from the slave device (ECU) to the master device for fast data acquisition (DAQ).

The XCP protocol supports synchronous data transfer in both directions, from Master to Slave (DAQ list) and from Slave to Master (STIM list). XCP allows several DAQ lists, which may be simultaneously active. The sampling and transfer of each DAQ list is triggered by individual events in the slave. To allow reduction of the transfer rate, a transfer rate prescaler may be applied to the DAQ lists.

**What is an ECU Characteristic?**

An ECU Characteristic represents an ECU internal memory range with defined access methods through the CCP protocol. The memory range of a single Characteristic can be structured in three ways:

- 0-dimensional—a single value
- 1-dimensional—a curve of values
- 2-dimensional—a field of values

A Characteristic may be defined as read-only or read and write accessible.
ECU M&C CCP and XCP Functions

The ECU M&C Channel functions do not expose the method used for ECU memory access. However, some applications may need specific CCP or XCP command sequences, or custom designed commands, which are not supported by the CCP or XCP protocols. For these applications, the ECU M&C CCP functions and the ECU M&C XCP functions provide access to the ECU information at a very low level.

Basic Programming Model

The flowchart in Figure 4-2 illustrates the process to initiate communication to an ECU with the ECU M&C Channel functions. A description of each step in the decision process follows the flowchart.
Figure 4-2. ECU Communication Decision Chart
ECU Open

The **ECU Open** function combines the opening of a selected ASAM MCD 2MC database file with the .A2L file extension and the selection of a stored ECU name. The required parameters are the ASAM MCD 2MC database path and filename, and the dedicated CAN interface if you are using CCP or XCP with CAN. The CAN interface is used for communication with the ECU. If you are using XCP with UDP or TCP, a port number and IP address or hostname must be defined in the A2L database.

The function to open and select an ECU is **MC ECU Open.vi** in LabVIEW or **mcDatabaseOpen** followed by **mcECUSelectEx** in C.

**Note** The import of ASAM MCD 2MC database files into MAX is not supported.

**ASAM MCD 2MC Communication Properties for CCP or XCP with CAN**

If your ASAM MCD 2MC database file already contains communication properties, you can directly open the communication to your selected ECU.

If the communication properties are not stored in the ASAM MCD 2MC file, the communication properties must be manually set. To establish communication through CCP or XCP with CAN, the target ECU slave should be addressed by setting the following properties.

**CRO ID**

The **CRO ID** (Command Receive Object) is used to send commands and data from the host to the slave device.

**DTO ID**

The **DTO ID** (Data Transmission Object) is used by the ECU to respond to CCP commands, and to send data and status information to the CCP master.

**Station Address**

CCP is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a **Station Address** that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its **Station Address**, the ECU must not react to CCP commands sent by the CCP master.
Chapter 4 Using the ECU M&C API

Baudrate

The baudrate property may be missing in an A2L database file and can be set explicitly within the application. This property provides the baud rate at which communication will occur, and applies to all tasks initialized with the interface. You can specify one of the predefined baud rates, or specify advanced baud rates which refer to the settings of the Bit Timing Register 0 (BTR0) and 1 (BTR1). For more information, refer to the Interface Properties dialog in MAX, or the NI-CAN Hardware and Software Manual. The baud rate is originally set within MAX.

ASAM MCD 2MC Communication Properties for XCP with UDP or TCP

If the XCP communication properties are not stored in the ASAM MCD 2MC file, the communication properties must be manually set. To establish communication through XCP with UDP or TCP the target ECU slave should be addressed by setting the following properties.

IP Address or hostname

The IP address refers to the identifier for a computer or device on a TCP/IP network. Networks using the TCP/IP protocol route messages based on the IP address of the destination.

A hostname describes the unique name by which a device is known on a network. Hostnames are used by various naming systems: NIS, DNS, SMB, etc. Hostnames are high-level aliases which ultimately correlate to unique network hardware MAC addresses.

Port number

In TCP/IP and UDP networks, a port is an end-point to a logical connection through which a client program specifies a server program on a computer in a network. Port numbers range from 0 to 65536, but only port numbers 0 to 1024 are reserved for privileged services and designated as well-known ports.

ECU Connect

The ECU Connect function establishes communication to the selected ECU through CCP using the CCP CONNECT command or through XCP using the CONNECT command. It establishes a logical connection to an ECU. Unless a slave device (ECU) is unconnected, it must not execute or respond to any command sent by the application. The only exception to this
rule is the Test command, in which case the slave with the specified address may acknowledge the command. Only a single slave can be connected to the application at a time. After a successful ECU Connect you can create a Measurement Task or read/write a Characteristic.

The function to open and select an ECU is MC ECU Connect.vi in LabVIEW and mcECUConnect in C.

**ECU Disconnect**

The ECU Disconnect function permanently disconnects the specified slave and ends the measurement and calibration session. When the measurement and calibration session is terminated, all DAQ lists for the device are stopped and cleared, and the protection masks of the device are set to their default values.

The function to disconnect an ECU is MC ECU Disconnect.vi in LabVIEW or mcECUDisconnect in C.

**ECU Close**

The MC ECU Close function deselects the ECU and closes the remaining database reference handle. MC ECU Close must always be the final ECU M&C function call. If you do not use MC ECU Close, the remaining task configurations can cause problems in the execution of subsequent ECU M&C applications.

The function to close an ECU is MC ECU Close.vi in LabVIEW. To deselect the ECU and close the database reference handle in C, call the function mcECUDeselect followed by mcDatabaseClose.

**Characteristic Read and Write**

**Access Characteristics**

To access the Characteristics of an ECU you must select and connect to the specified ECU through the procedure given above. The function to open and select an ECU is MC ECU Open.vi in LabVIEW, or mcDatabaseOpen followed by mcECUSelectEx in C. Once the ECU has been connected an ECU Reference handle (ECU ref out in LabVIEW, ECURefNum in C) must be acquired before any additional actions can be performed.
Characteristic Read

The application must call the Read Characteristic function to obtain scaled floating point samples. The application typically calls Read Characteristic on demand. Calling Read Characteristic in a loop can cause significant CAN network traffic, as Characteristics may contain large amounts of data.

The function to read 0- to 2-dimensional Characteristics is MC Characteristic Read.vi in LabVIEW or mcCharacteristicRead in C. The function to read single double values as Characteristics is MC Characteristic Read Single Value.vi in LabVIEW or mcCharacteristicReadSingleValue in C.

Before reading a Characteristic, it may be helpful to verify the dimension of the Characteristic based on the definition in the ASAM MCD 2MC database file. Depending on the dimension of the Characteristic, use the appropriate Read function for reading a double, a 1D array of doubles, or a 2D array of doubles.

The function to verify a dimension of a named Characteristic is MC Get Property.vi with the parameter Characteristic/Dimension in LabVIEW or mcGetProperty with the parameter mcPropChar_Dimension in C.

Characteristic Write

The application must call the Write Characteristic function to output scaled floating-point samples. The application typically calls Write Characteristic on demand. Calling Write Characteristic in a loop can cause significant network traffic, as Characteristics may contain large amounts of data.

The function to write a Characteristic is MC Characteristic Write.vi in LabVIEW or mcCharacteristicWrite in C.

Before writing a Characteristic, it may be helpful to verify the dimension of the Characteristic based on the definition in the ASAM MCD 2MC database file. Depending on the dimension of the Characteristic, use the appropriate Write function for writing a double, a 1D array of doubles, or a 2D array of doubles.

The function to verify a dimension of a named Characteristic is MC Get Property.vi with the parameter Characteristic/Dimension in LabVIEW or mcGetProperty with the parameter mcPropChar_Dimension in C.
**Measurement Task**

To create a Measurement task you need to select available Measurement signals from an ASAM MCD 2MC database file. Create a valid ECU Reference handle as described in the *Access Characteristics* section.

The flowchart in Figure 4-3 shows the process to perform an ECU Measurement task. A description of each step in the decision process follows the flowchart.

![Figure 4-3. ECU Measurement Setup Flowchart](image-url)
**DAQ Initialize**

The **DAQ Initialize** function initializes a list of Measurement channels as a single Measurement task. The communication for that Measurement task is started by the first **DAQ Read** function. The **DAQ Initialize** function is **MC DAQ Initialize.vi** in LabVIEW or **mcDAQInitialize** in other languages.

The **DAQ Initialize** function uses the following input parameters:

**Measurement list**

Specifies the list of channels for the task with one string for each channel.

**ECU Reference handle**

Typically, the **ECU Reference handle** is created by opening the ASAM MCD 2MC database using the **ECU Open** function, then connecting to an ECU using the **ECU Connect** function.

**Mode**

Specifies the input mode to use for the task. This determines the data transfer for the task (Polling, DAQ list, or STIM list).

**SampleRate**

Specifies the sampling rate for a specific DAQ list or STIM list. The sample rate is specified in Hertz (samples per second). For more information, refer to the **DAQ Read** section.

**DTO ID**

If you are using the CCP protocol, the **DTO ID** (Data Transmission Object) is used by the ECU to respond to CCP commands, and to send data and status information to the CCP master.

**DAQ Start Stop**

The optional function **DAQ Start Stop** starts or stops the transmission of the DAQ lists for an ECU M&C Measurement task. If you do not specify **MC DAQ Start Stop.vi** before your first **DAQ Read** or **DAQ Write** function, **MC DAQ Start Stop.vi** is implicitly performed by the first **DAQ Read** or **DAQ Write** call. After you start the transmission of the DAQ lists or STIM lists, you can no longer change the configuration of the Measurement task with **Set Property**. **MC DAQ Start Stop.vi** is implicitly performed by **DAQ Clear** to stop transmission of the DAQ lists.
The function to start a DAQ list is **MC DAQ Start Stop.vi** in LabVIEW or **mcDAQStartStop** in C.

**DAQ Read**

The application must call the **DAQ Read** function to obtain floating-point samples. The application typically calls **DAQ Read** in a loop until done. The **Read** function is **MC DAQ Read.vi** in LabVIEW (all types that do not end in **Time & Dbl**) or **mcDAQRead** in other languages.

The behavior of **Read** depends on the initialized sample rate and the selected mode.

**sample rate = 0**

**DAQ Read** returns a single sample from the most recent message(s) received from the network. One sample is returned for every channel in the **DAQ Initialize** list.

Figure 4-4 shows an example of **DAQ Read** with a sample rate = 0. A, B, and C represent messages for the initialized channels. **def** represents the default value 0. If no message is received after the start of the application, the default value 0 is returned along with a warning.

![Figure 4-4. Example of Read With Sample Rate = 0](image)
sample rate > 0

**DAQ Read** returns an array of samples for every channel in the **DAQ Initialize** list. Each time the clock ticks at the specified rate, a sample from the most recent message(s) is inserted into the arrays. In other words, the samples are repeated in the array at the specified rate until a new message is received. By using the same sample rate with NI-DAQ Analog Input channels or NI-DAQmx Analog Input channels, you can compare ECU DAQ and NI-DAQ/NI-DAQmx samples over time.

Figure 4-5 shows an example of **DAQ Read** with a sample rate > 0. A, B, and C represent messages for the initialized channels. $\Delta t$ represents the time between samples as specified by the sample rate. $\text{def}$ represents the default value 0.

**Figure 4-5. Example of Read With Sample Rate > 0**

**DAQ Write**

If you are using XCP and the DAQ initialize mode is set to **STIM list** the application must call the **DAQ Write** function to output floating-point samples. The application typically calls **DAQ Write** in a loop until done. The **DAQ Write** function is **MC DAQ Write.vi** in LabVIEW or **mcDAQWrite** in other languages.
DAQ Clear

**DAQ Clear** must always be the final function called for a specific Measurement task. If you do not use **DAQ Clear**, the remaining Measurement task configuration can cause problems in the execution of subsequent ECU M&C applications. Because this function clears the Measurement task, the Measurement task reference is transferred into an ECU reference task handle. To change the properties of a running Measurement task, use **DAQ Start Stop** to stop the task, **Set Property** to change the desired DAQ property, then **DAQ Start Stop** to restart the Measurement task again.

The function to clear a DAQ list is **MC DAQ Clear.vi** in LabVIEW or **mcDAQClear** in C.

Memory Programming

The ECU Measurement and Calibration Toolkit allows you to issue a memory programming sequence for your ECU after you create an ECU Reference handle as described in the *Basic Programming Model* section.

The flowchart in Figure 4-6 illustrates the general process of a memory programming sequence of an ECU with the ECU M&C functions. A description of each step in the decision process follows the flowchart.
Program Start

The **Program Start** function sets the ECU into the memory programming mode. Note that in this mode specific features might be restricted, for instance, the ECU might refuse to change into the programming mode while a DAQ list is running. The **Program Start** function is **MC Program Start.vi** in LabVIEW and **mcProgramStart** in other languages.

Using XCP?

Yes

Optional: XCP Program Prepare

Optional: Set XCP Program Properties

No

Program Start

Clear Memory before Programming?

Yes

Clear Memory

No

Program

Program Reset

Using XCP?

Yes

Optional: XCP Program Verify

No
Clear Memory

It might be necessary to clear the memory before it is reprogrammed. The details are ECU-dependent. The Clear Memory function performs the memory clearing operation. It is MC Clear Memory.vi in LabVIEW or mcClearMemory in other languages.

Program

The Program function actually downloads the new code to the ECU. It is MC Program.vi in LabVIEW or mcProgram in other languages.

Program Reset

The Program Reset function terminates a programming sequence. Note that for the XCP protocol, Program Reset performs a hardware reset of the ECU and causes a disconnect. You have to reconnect to the ECU using the ECU Connect function to perform further operations. The Program Reset function is MC Program Reset.vi in LabVIEW and mcProgramReset in other languages.

Optional Steps for the XCP Protocol

XCP Program Prepare

An ECU using the XCP protocol might require an XCP PROGRAM_PREPARE command before a programming sequence is started. This command can be issued with the XCP Program Prepare function. It is MC XCP Program Prepare.vi in LabVIEW and mcXCPProgramPrepare in other languages.

Set XCP Programming Properties

XCP allows the programming process to be controlled by several variables. These are the Compression Method, Encryption Method, Programming Method, and Access Method properties. They default to 0, but can be set to any value before the programming process starts. The allowed values for these properties are ECU-specific. If any of these properties is set to a nonzero value, an appropriate PROGRAM_FORMAT XCP command is issued before the programming takes place. Note that the Access Method property also affects the Clear Memory function.
**XCP Program Verify**

After the memory programming XCP allows to verify whether the operation was successful by the PROGRAM_VERIFY XCP command. The details of this command are highly ECU-specific. This command can be issued using the XCP Program Verify function. It is **MC XCP Program Verify.vi** in LabVIEW and **mcXCPProgramVerify** in other languages.

**Additional Programming Topics**

The following sections provide information you can use to extend the basic programming model.

**Get Names**

If you are developing an application that another person will use, you may not want to specify a fixed channel list for a Measurement task or a fixed channel for a Characteristic in the application. Ideally, you want the end-user to select the channels of interest from user interface controls such as list boxes. The **Get Names** function queries an ASAM MCD 2MC database and returns a list of all channels in that database regarding the selected query mode. You can use this list to populate user-interface controls. The user can then select channels from these controls, avoiding the need to type in each name. Once the user makes the selections, the application can pass the resulting list to the appropriate function, such as **DAQ Initialize**, for an ECU Measurement channel list. The **Get Names** function is **MC Get Names.vi** in LabVIEW or **mcGetNames** in C.

**Set/Get Properties**

If you need to change particular parameters within an application, such as the DTO ID, use the following sequence:

1. Initialize the Measurement task as stopped. The **Initialize** function is **MC DAQ Initialize.vi** in LabVIEW or **mcDAQInitialize** in C.  
2. Use **Set Property** to specify the new value for the DTO_ID property. The **Set Property** function is **MC Set Property.vi** in LabVIEW or **mcSetProperty** in C.  
3. Start the Measurement task with the **DAQ Start Stop** function. The **DAQ Start Stop** function is **MC DAQ Start Stop.vi** in LabVIEW or **mcDAQStartStop** in C. You can also start the Measurement task implicitly by issuing **DAQ Read**.

**ECU M&C Toolkit User Manual**

4-16

**ni.com**
After the task is started you may need to change properties again. To change properties within the application, use the DAQ Start Stop function to stop the Measurement task, Set Property to change properties, then start the task again.

You also can use the Get Property function to get the value of any property. The Get Property function returns values whether the task is running or not. The Get Property function is MC Get Property.vi in LabVIEW or mcGetProperty in C.

Generic CCP Functions

The generic ECU M&C CCP functions provide direct access to the CCP commands on a very low programming level. For further information for the use and parameters of the CCP commands, refer to the CAN Calibration Protocol Specification, Version 2.1. Table 4-1 provides an overview of the CCP commands and their corresponding LabVIEW VIs or C functions.

<table>
<thead>
<tr>
<th>CCP Command</th>
<th>LabVIEW VI Name</th>
<th>C Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION_SERVICE</td>
<td>MC CCP Action Service.vi</td>
<td>mcCCPActionService</td>
</tr>
<tr>
<td>BUILD_CHKSUM</td>
<td>MC Build Checksum.vi</td>
<td>mcBuildChecksum</td>
</tr>
<tr>
<td>CLEAR_MEMORY</td>
<td>MC Clear Memory.vi</td>
<td>mcClearMemory</td>
</tr>
<tr>
<td>DIAG_SERVICE</td>
<td>MC CCP Diag Service.vi</td>
<td>mcCCPDiaService</td>
</tr>
<tr>
<td>DNLOAD</td>
<td>MC Download.vi</td>
<td>mcDownload</td>
</tr>
<tr>
<td>GET_ACTIVE_CAL_PAGE</td>
<td>MC CCP Get Active Cal Page.vi</td>
<td>mcCCPGetActiveCalPage</td>
</tr>
<tr>
<td>GET_CCP_VERSION</td>
<td>MC CCP Get Version.vi</td>
<td>mcCCPGetVersion</td>
</tr>
<tr>
<td>GET_S_STATUS</td>
<td>MC CCP Get Session Status.vi</td>
<td>mcCCPGetSessionStatus</td>
</tr>
<tr>
<td>MOVE</td>
<td>MC CCP Move Memory.vi</td>
<td>mcCCPMoveMemory</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>MC Program.vi</td>
<td>mcProgram</td>
</tr>
<tr>
<td>SELECT_CAL_PAGE</td>
<td>MC CCP Select Cal Page.vi</td>
<td>mcCCPSelectCalPage</td>
</tr>
</tbody>
</table>
Chapter 4  Using the ECU M&C API

Table 4-1. Overview of the CCP Commands with Related VIs and C Functions (Continued)

<table>
<thead>
<tr>
<th>CCP Command</th>
<th>LabVIEW VI Name</th>
<th>C Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET_S_STATUS</td>
<td>MC CCP Set Session Status.vi</td>
<td>mcCCPSetSessionStatus</td>
</tr>
<tr>
<td>UPLOAD</td>
<td>MC Upload.vi</td>
<td>mcUpload</td>
</tr>
</tbody>
</table>

Generic XCP Functions

The generic ECU M&C XCP functions provide direct access to the XCP commands on a very low programming level. For more information about the use and parameters of the XCP commands, refer to the ASAM XCP Part 2 Protocol Layer Specification. Table 4-2 provides an overview of the XCP commands with their corresponding LabVIEW VIs or C functions.

Table 4-2. Overview of the XCP Commands with Related VIs and C Functions

<table>
<thead>
<tr>
<th>XCP Command</th>
<th>LabVIEW VI Name</th>
<th>C Function Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD_CHKSUM</td>
<td>MC Build Checksum.vi</td>
<td>mcBuildChecksum</td>
</tr>
<tr>
<td>CLEAR_MEMORY</td>
<td>MC Clear Memory.vi</td>
<td>mcClearMemory</td>
</tr>
<tr>
<td>COPY_CAL_PAGE</td>
<td>MC XCP Copy Cal Page.vi</td>
<td>mcXCPCopyCalPage</td>
</tr>
<tr>
<td>DOWNLOAD</td>
<td>MC Download.vi</td>
<td>mcDownload</td>
</tr>
<tr>
<td>GET_CAL_PAGE</td>
<td>MC XCP Get Cal Page.vi</td>
<td>mcCCPGetActiveCalPage</td>
</tr>
<tr>
<td>GET_ID</td>
<td>MC XCP Get ID.vi</td>
<td>mcXCPGetId</td>
</tr>
<tr>
<td>GET_STATUS</td>
<td>MC XCP Get Status.vi</td>
<td>mcXCPGetStatus</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>MC Program.vi</td>
<td>mcProgram</td>
</tr>
<tr>
<td>PROGRAM_PREPARE</td>
<td>MC XCP Program Prepare.vi</td>
<td>mcXCPProgramPrepare</td>
</tr>
<tr>
<td>PROGRAM_RESET</td>
<td>MC Program Reset.vi</td>
<td>mcProgramReset</td>
</tr>
<tr>
<td>PROGRAM_START</td>
<td>MC Program Start.vi</td>
<td>mcProgramStart</td>
</tr>
<tr>
<td>PROGRAM_VERIFY</td>
<td>MC XCP Program Verify.vi</td>
<td>mcXCPProgramVerify</td>
</tr>
<tr>
<td>SET_CAL_PAGE</td>
<td>MC XCP Set Cal Page.vi</td>
<td>mcXCPSetCalPage</td>
</tr>
<tr>
<td>SET_REQUEST</td>
<td>MC XCP Set Request.vi</td>
<td>mcXCPSetRequest</td>
</tr>
</tbody>
</table>
Seed and Key Algorithm

To restrict access to an ECU, you can add a defined login mechanism to ECU software. The Association for Standardization of Automation and Measuring Systems (ASAM) defines this seed, which may be stored in the A2L file. A typical login mechanism may happen as follows:

1. Connect to the ECU.
2. Exchange station identifications.
3. Get the seed for the key.
4. Calculate the key using a seed and key DLL as ASAM defines.
5. Unlock the ECU protection by sending the calculated key.

ASAM AE Common defines the seed and key algorithm in the Seed and Key and Checksum Calculation API Version 1.0. The specification defines the Win32 APIs for seed and key calculation and checksum calculation.

Definition for Seed and Key Algorithm

Function name: ASAP1A_CCP_ComputeKeyFromSeed

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pointer to the seed data, retrieved from the ECU GET_SEED command.</td>
</tr>
<tr>
<td>2</td>
<td>Seed data size in number of bytes.</td>
</tr>
<tr>
<td>3</td>
<td>Pointer to key data, returning the calculated.</td>
</tr>
<tr>
<td>4</td>
<td>Key data size in number of bytes.</td>
</tr>
<tr>
<td>5</td>
<td>Key data size in number of bytes.</td>
</tr>
</tbody>
</table>

The calling convention is as defined in the WIN32 API Specification for ASAP1b, section 2.4.
Seed and Key Example

The following example shows a possible header file for a library for key calculation.

/*
// Header file for ASAP1a CCP V2.1 Seed and Key Algorithm
*/
#ifndef _SEEDKEY_H_
#define _SEEDKEY_H_
#ifndef DllImport
#define DllImport __declspec( dllimport )
#endif
#ifndef DllExport
#define DllExport __declspec( dllexport )
#endif
#ifdef SEEDKEYAPI_IMPL // only defined by implementor of SeedKeyApi
#define SEEDKEYAPI DllExport __cdecl
#else
#define SEEDKEYAPI DllImport __cdecl
#endif
#ifdef __cplusplus
extern "C" {
#endif
BOOL SEEDKEYAPI ASAP1A_CCP_ComputeKeyFromSeed (BYTE *Seed,
unsigned short SizeSeed, BYTE *Key, unsigned short MaxSizeKey,
unsigned short *SizeKey);
// Seed: Pointer to seed data
// SizeSeed: Size of seed data (length of "Seed")
// Key: Pointer, where DLL should insert the calculated key data.
// MaxSizeKey: Maximum size of "Key".
// SizeKey: Should be set from DLL corresponding to the number of data
// inserted to "Key" (at most "MaxSizeKey")
// Result: The value FALSE (= 0) indicates that the key could not be
// calculated from seed data (for example, "MaxSizeKey" is too small).
// TRUE (!= 0) indicates success of key calculation.
#endif
#endif // _SEEDKEY_H_
Checksum Algorithm

ASAM proposed a WIN32 API function to have a common interface to implement the checksum algorithms for verifying ECU calibration and program data. For details, refer to the ASAM Seed and Key and Checksum Calculation API Version 1.0.

Definition for a Checksum Algorithm

Function name: BOOL CalcChecksum(struct TRange *ptr, int nRanges, BYTE *pnChecksum, int *pnSignificant, WORD nFlags)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Pointer to an array of ranges, stored in structures of type TRange.</td>
</tr>
<tr>
<td>2</td>
<td>Number of ranges stored in the array that parameter 1 points to.</td>
</tr>
<tr>
<td>3</td>
<td>Pointer to a byte array where the checksum must be stored. The DLL writes a maximum of 8 bytes, so the caller should reserve space for 8 bytes of data.</td>
</tr>
<tr>
<td>4</td>
<td>Length of actually calculated checksum (1...8).</td>
</tr>
</tbody>
</table>
| 5         | Flag field for commanding how the algorithm works. Currently, only bit 0 is defined:  

  **Bit 0 = 0**: pnChecksum receives the algorithm checksum calculation result.  
  **Bit 0 = 1**: pnChecksum points to a checksum that is compared within the DLL with the checksum that the algorithm calculates. Returns TRUE if the checksums are identical, FALSE otherwise.  

All other bits are reserved and should be set to 0. |

TRange is defined as follows:

```c
struct TRange  
{
    char *pMem;
    unsigned long lLen;
}
```
The calling convention is as defined in the *WIN32 API Specification for ASAP1b*, chapter 2.4.

**Checksum Algorithm Example**

The following example shows a possible header file for a library for checksum calculation.

```c
/*
// checksum.h
// Header file for Checksum Algorithm
*/
#ifndef _CHECKSUM_H
#define _CHECKSUM_H
#ifdef __cplusplus
extern "C" {
#endif
#ifndef DllImport
#define DllImport __declspec( dllimport )
#endif
#ifndef DllExport
#define DllExport __declspec( dllexport )
#endif
#ifdef CHECKSUMAPI_IMPL // only defined by implementor of ChecksumApi
#define CHECKSUMAPI DllExport __cdecl
#else
#define CHECKSUMAPI DllImport __cdecl
#endif
struct TRange
{
  char *pMem;
  unsigned long lLen;
};
#ifdef __cplusplus
extern "C" {
#endif
BOOL CHECKSUMAPI CalcChecksum(struct TRange *ptr,
int nRanges,
BYTE *pnChecksum,
int *pnSignificant,
WORD nFlags);
#ifdef __cplusplus
}
#endif
#endif //_CHECKSUM_H
```
Seed and Key and Checksum Algorithms for VxWorks Targets

LabVIEW RT users can run the ECU Measurement and Calibration Toolkit on either a LabVIEW RT target such as a PXI controller or an Intel-based CompactRIO running the Pharlap operating system, which supports Win32 calls, or on a PowerPC-based CompactRIO controller running a Windriver VxWorks operating system.

If you are using a CompactRIO target with a PowerPC controller running a VxWorks operating system, you cannot use any Win32 function calls based on a DLL. However, the GNU tool chain distributed with VxWorks can compile shared libraries for controllers running Wind River VxWorks, including the CompactRIO 901x and 907x series. You can access the shared libraries (*.OUT modules) for VxWorks through the ECU Measurement and Calibration Toolkit by using a C/C++ function definition that is slightly different from the ASAM specification, due to the differences between Win32 DLLs and VxWorks OUT modules.

You can obtain the GNU tool chain for VxWorks by either purchasing a VxWorks development license from Wind River or downloading the redistributable GNU tool chain from ni.com. If you purchase VxWorks, you can use the Wind River Workbench IDE, featuring source code-level debugging and build management. The redistributable GNU tool chain downloadable on ni.com offers debugging only at the assembly code level, and you must use the included GNU Make to build binaries.

Note  LabVIEW 2009 RT installs version 6.3 of the VxWorks OS to compatible targets. All builds should be targeted to corresponding versions and use corresponding header files. As new versions of LabVIEW RT become available, different versions of VxWorks may be installed and may require you to rebuild your libraries. Refer to the readme file for LabVIEW RT to find the corresponding VxWorks OS version.

Example of a Header for a Seed and Key and Checksum Algorithm for a VxWorks Target

The module name of the compiled out file must correspond to the seed and key and checksum function name defined in the ASAM A2L database.

The following example uses the seed and key module name ccpecu.out. Therefore, the seed and key function is named ccpecu_ASAP1A_CCP_ComputeKeyFromSeed. The example uses the prefix in addition to the ASAM standard, because the VxWorks OS requires unique function names across all loaded modules. Therefore, multiple
modules must not export functions with the same names. To support multiple ECUs with the ECU Measurement and Calibration Toolkit, each seed and key and checksum function must have a unique name. To achieve unique function names for the seed and key and checksum functions, these functions have the module name (in lower case) followed by an underline as a prefix.

The following example shows a possible header file for a module used for seed and key and checksum calculation under VxWorks targets.

```c
#ifndef __CCPECU_h__
#define __CCPECU_h__

/// \brief defines the name of the Seed-Key function
///
/// Here the name of the seed key function is defined.
/// The name of the seed key function is the name of the module in lower case letters followed by an underscore and the function name "ASAP1A_CCP_ComputeKeyFromSeed".
/// \todo replace the prefix "ccpecu_" by the name of your module in lower case letters.
#define SEED_KEY_NAME ccpecu_ASAP1A_CCP_ComputeKeyFromSeed

/// \brief defines the name of the Checksum function
///
/// Here the name of the Checksum function is defined.
/// The name of the Checksum function is the name of the module in lower case letters followed by an underscore and the function name "CalcChecksum".
/// \todo replace the prefix "ccpecu_" by the name of your module in lower case letters.
#define CALC_CHECKSUM_NAME ccpecu_CalcChecksum

struct TRange
{
    char      *pMem;
    unsigned long  lLen;
};

#endif cplusplus
.extern "C" {
#endif
```
/// \brief Function to calculate a key from a given seed to unlock an ECU resource.
///
/// This function calculates a key from a given seed so that you are able to unlock the access to an ECU resource. The seed is generated by the ECU and needs to be queried before you can unlock an ECU resource.
bool SEED_KEY_NAME(
    unsigned char *Seed,       ///< Seed provided by the ECU
    unsigned short SizeSeed,   ///< Size of the seed provided by the ECU
    unsigned char *Key,        ///< Pointer to a buffer to return the key
    unsigned short MaxSizeKey, ///< Size of the buffer provided to return the key
    unsigned short *SizeKey    ///< returns the size of the calculated key
)
__attribute__((section (".export")));

/// \brief Function to calculate a checksum over a given memory range.
///
/// This function calculates a checksum over a given memory range. The function is used, for example, to verify data after a download or programming action.
bool CALC_CHECKSUM_NAME (
    struct TRange *ptr,            ///< Description of the memory area to be checked
    int nRanges,        ///< Number of memory blocks to be checked
    unsigned char *pnCheckSum,     ///< Pointer to a buffer to return the checksum
    int *pnSignificant,  ///< Size of the buffer to return the checksum
    unsigned short nFlags       ///< flags for calculating the checksum
)
__attribute__((section (".export")));

#endif // __cplusplus
#endif // __CCPECU_h__
This chapter lists the LabVIEW VIs for the ECU M&C API and describes the format, purpose, and parameters for each VI. The VIs in this chapter are listed alphabetically. Unless otherwise stated, each VI suspends execution of the calling thread until it completes.

Section Headings

The following are section headings found in the ECU M&C API for LabVIEW VIs.

Purpose

Each VI description includes a brief statement of the purpose of the VI.

Format

The format section describes the format of each VI.

Input and Output

The input and output parameters for each VI are listed.

Description

The description section gives details about the purpose and effect of each VI.

List of VIs

The following table is an alphabetical list of the ECU M&C Toolkit VIs.

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC Build Checksum.vi</td>
<td>Calculates a checksum over a defined memory range within the ECU.</td>
</tr>
<tr>
<td>MC Calc Checksum.vi</td>
<td>Calculates the checksum of a data block in memory.</td>
</tr>
<tr>
<td>MC CCP Action Service.vi</td>
<td>Calls an implementation-specific action service on the ECU.</td>
</tr>
</tbody>
</table>
### Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC CCP Diag Service.vi</td>
<td>Calls a diagnostic service on the ECU.</td>
</tr>
<tr>
<td>MC CCP Get Active Cal Page.vi</td>
<td>Retrieves the ECU Memory Transfer Address pointer to the calibration data page.</td>
</tr>
<tr>
<td>MC CCP Get Result.vi</td>
<td>Uploads requested data.</td>
</tr>
<tr>
<td>MC CCP Get Session Status.vi</td>
<td>Retrieves the current calibration status of the ECU.</td>
</tr>
<tr>
<td>MC CCP Get Version.vi</td>
<td>Retrieves the version of the CCP implemented in the ECU.</td>
</tr>
<tr>
<td>MC CCP Move Memory.vi</td>
<td>Moves a memory block on the ECU.</td>
</tr>
<tr>
<td>MC CCP Select Cal Page.vi</td>
<td>Sets the beginning of the calibration data page.</td>
</tr>
<tr>
<td>MC CCP Set Session Status.vi</td>
<td>Updates the ECU with the current state of the calibration session.</td>
</tr>
<tr>
<td>MC Characteristic Read.vi</td>
<td>Reads data from a named Characteristic on the ECU which is identified by the ECU Reference handle. The Poly VI returns a specific double, 1D, or 2D double array.</td>
</tr>
<tr>
<td>MC Characteristic Read Single Value.vi</td>
<td>Reads a value from a named Characteristic on the ECU which is identified by the ECU Reference handle.</td>
</tr>
<tr>
<td>MC Characteristic Write.vi</td>
<td>Writes the value(s) of a named Characteristic to an ECU identified by the ECU ref handle. The Poly VI writes the selected type double, 1D or 2D array.</td>
</tr>
<tr>
<td>MC Characteristic Write Single Value.vi</td>
<td>Writes a value to a named Characteristic on the ECU.</td>
</tr>
<tr>
<td>MC Clear Memory.vi</td>
<td>Clears the contents of a specified memory block.</td>
</tr>
<tr>
<td>MC Conversion Create.vi</td>
<td>Creates a signal conversion object in memory.</td>
</tr>
<tr>
<td>MC DAQ Clear.vi</td>
<td>Stops communication for the Measurement task and then clears the configuration.</td>
</tr>
<tr>
<td>MC DAQ Initialize.vi</td>
<td>Initializes a Measurement task for the specified Measurement channel list.</td>
</tr>
<tr>
<td>MC DAQ List Initialize.vi</td>
<td>Defines a DAQ list on a specific DAQ list number and initializes the Measurement task for the specified Measurement channel list.</td>
</tr>
</tbody>
</table>
Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC DAQ Read.vi</td>
<td>Reads samples from a Measurement task. Samples are obtained from received CAN messages.</td>
</tr>
<tr>
<td>MC DAQ Start Stop.vi</td>
<td>Starts or stops transmission of the DAQ lists for the specified Measurement task.</td>
</tr>
<tr>
<td>MC DAQ Write.vi</td>
<td>Writes samples to a Measurement task.</td>
</tr>
<tr>
<td>MC Database Close.vi</td>
<td>Closes a specified A2L Database.</td>
</tr>
<tr>
<td>MC Database Create.vi</td>
<td>Creates an A2L database in memory, for using the ECU M&amp;C Toolkit VIs without access to a valid ASAM A2L file.</td>
</tr>
<tr>
<td>MC Database Open.vi</td>
<td>Opens a specified A2L Database.</td>
</tr>
<tr>
<td>MC Double to Text.vi</td>
<td>Converts a numerical value to a text string using an enumeration or range text type scaling.</td>
</tr>
<tr>
<td>MC Download.vi</td>
<td>Downloads data to an ECU.</td>
</tr>
<tr>
<td>MC ECU Close.vi</td>
<td>Closes the selected ECU and the associated A2L database.</td>
</tr>
<tr>
<td>MC ECU Connect.vi</td>
<td>Establishes the communication to the selected ECU through the CCP protocol. After a successful ECU Connect you can create a Measurement Task or read/write a Characteristic.</td>
</tr>
<tr>
<td>MC ECU Create.vi</td>
<td>Creates an ECU object in memory.</td>
</tr>
<tr>
<td>MC ECU Deselect.vi</td>
<td>Deselects an ECU and invalidates the ECU reference handle.</td>
</tr>
<tr>
<td>MC ECU Disconnect.vi</td>
<td>Permanently disconnects the CCP communication to the selected ECU and ends the calibration session.</td>
</tr>
<tr>
<td>MC ECU Open.vi</td>
<td>Opens a specified A2L database and selects the first ECU found in the database. If there are several ECUs stored in the A2L database use the Database Open and ECU Select VIs.</td>
</tr>
<tr>
<td>MC ECU Select.vi</td>
<td>Selects an ECU from the names stored in an A2L database.</td>
</tr>
<tr>
<td>MC Event Create.vi</td>
<td>Creates an Event object in memory.</td>
</tr>
<tr>
<td>MC Generic.vi</td>
<td>Sends a generic CCP or XCP command.</td>
</tr>
<tr>
<td>MC Get Names.vi</td>
<td>Gets an array of ECU names, Measurement names, Characteristic names, Event names, Calibration page names, or Group names from a specified A2L database file.</td>
</tr>
</tbody>
</table>
### Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC Get Property.vi</td>
<td>Gets a property for the object referenced by the <strong>reference in</strong> terminal. The poly VI selection determines the property to get.</td>
</tr>
<tr>
<td>MC Measurement Create.vi</td>
<td>Creates a Measurement object in memory.</td>
</tr>
<tr>
<td>MC Measurement Read.vi</td>
<td>Reads a single Measurement value from the ECU.</td>
</tr>
<tr>
<td>MC Measurement Write.vi</td>
<td>Writes a single Measurement value to the ECU.</td>
</tr>
<tr>
<td>MC Program.vi</td>
<td>Programs a memory block on the ECU.</td>
</tr>
<tr>
<td>MC Program Reset.vi</td>
<td>Indicates the end of a programming sequence.</td>
</tr>
<tr>
<td>MC Program Start.vi</td>
<td>Indicates the start of a programming sequence.</td>
</tr>
<tr>
<td>MC Set Property.vi</td>
<td>Sets a property for the specified A2L database file, Measurement task or Characteristic. The poly VI selection determines the property to set.</td>
</tr>
<tr>
<td>MC Text To Double.vi</td>
<td>Converts a text value into the numeric representation using an enumeration or range text type scaling.</td>
</tr>
<tr>
<td>MC Upload.vi</td>
<td>Uploads data from an ECU.</td>
</tr>
<tr>
<td>MC XCP Copy Cal Page.vi</td>
<td>Forces a copy transaction of one calibration page to another.</td>
</tr>
<tr>
<td>MC XCP Get Cal Page.vi</td>
<td>Queries a calibration page setting.</td>
</tr>
<tr>
<td>MC XCP Get ID.vi</td>
<td>Queries session configuration or slave device identification.</td>
</tr>
<tr>
<td>MC XCP Get Status.vi</td>
<td>Queries the current session status from an ECU slave device.</td>
</tr>
<tr>
<td>MC XCP Program Prepare.vi</td>
<td>Prepares the programming of non volatile memory.</td>
</tr>
<tr>
<td>MC XCP Program Verify.vi</td>
<td>Performs a non-volatile memory certification task on the ECU device.</td>
</tr>
<tr>
<td>MC XCP Set Cal Page.vi</td>
<td>Sets a calibration page.</td>
</tr>
<tr>
<td>MC XCP Set Request.vi</td>
<td>Performs a request to save session and device information to non-volatile memory.</td>
</tr>
<tr>
<td>MC XCP Set Segment Mode.vi</td>
<td>Sets the mode of a specified segment.</td>
</tr>
</tbody>
</table>
**MC Build Checksum.vi**

**Purpose**
Calculates a checksum over a defined memory range within the ECU.

**Format**

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>ECU ref out</td>
</tr>
<tr>
<td>address</td>
<td>type of checksum</td>
</tr>
<tr>
<td>block size</td>
<td>checksum</td>
</tr>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>
```

**Input**

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Address is a cluster which contains the following values.

- **Address** specifies the address part of the source address.
- **Extension** contains the extension part of the source address.

Block size determines the size of the block for which the checksum must be calculated.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

- **status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error; VI did not execute the intended operation. A positive value means warning; VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

- **source** identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Type of checksum returns the type of the calculated checksum. If you are using the CCP protocol, type of checksum is 0xFF. For XCP, refer to the Description section.

Checksum returns the calculated checksum.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Build Checksum.vi calculates the checksum of a specified memory block inside the ECU starting at the selected Memory Transfer Address (MTA). The checksum algorithm is not specified by CCP and the checksum algorithm may be different on different devices.

If you are using the CCP protocol, MC Build Checksum.vi implements the CCP BUILD_CHKSUM command. The checksum algorithm is not specified by CCP and the checksum algorithm may be different on different devices.
If you are using the XCP protocol, **MC Build Checksum.vi** implements the BUILD_CHECKSUM command of the XCP specification. The result of the checksum calculation is returned in **Checksum** regardless of the checksum type. The following values for **type of checksum** are defined in the XCP specification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>XCP_ADD_11</td>
<td>Add BYTE into a BYTE checksum, ignore overflows</td>
</tr>
<tr>
<td>0x02</td>
<td>XCP_ADD_12</td>
<td>Add BYTE into a WORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x03</td>
<td>XCP_ADD_14</td>
<td>Add BYTE into a DWORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x04</td>
<td>XCP_ADD_22</td>
<td>Add WORD into a WORD checksum, ignore overflows, <strong>block size</strong> must be modulo 2</td>
</tr>
<tr>
<td>0x05</td>
<td>XCP_ADD_24</td>
<td>Add WORD into a DWORD checksum, ignore overflows, <strong>block size</strong> must be modulo 2</td>
</tr>
<tr>
<td>0x06</td>
<td>XCP_ADD_44</td>
<td>Add DWORD into DWORD, ignore overflows, <strong>block size</strong> must be modulo 4</td>
</tr>
<tr>
<td>0x07</td>
<td>XCP_CRC_16</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x08</td>
<td>XCP_CRC_16_CITT</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x09</td>
<td>XCP_CRC_32</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0xFF</td>
<td>XCP_USER_DEFINED</td>
<td>User defined algorithm in externally calculated function</td>
</tr>
</tbody>
</table>

If **type of checksum** is returned as 0xFF (XCP_USER_DEFINED), the slave can indicate that the master for calculating the checksum must use a user-defined algorithm implemented in an externally calculated function (for instance, Win32 DLL, UNIX shared object file, etc.). The master gets the name of the external function file to be used for this slave from the ASAM MCD 2MC description file or from a property which can be set.

For a detailed description of the checksum algorithm, refer to the **XCP Part 2 Protocol Layer Specification**.
MC Calc Checksum.vi

Purpose
Calculates the checksum of a data block in memory.

Format

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>The task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.</td>
</tr>
<tr>
<td>Data</td>
<td>A byte array upon which the checksum calculation is performed.</td>
</tr>
<tr>
<td>Type of checksum</td>
<td>Specifies the kind of checksum which is calculated.</td>
</tr>
<tr>
<td>Error in</td>
<td>A cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td>status</td>
<td>TRUE if an error occurred. This VI is not executed when status is TRUE.</td>
</tr>
<tr>
<td>code</td>
<td>The error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.</td>
</tr>
<tr>
<td>source</td>
<td>Identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Checksum is the calculated checksum.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Calc Checksum.vi implements a checksum calculation over a given data block. The checksum algorithm is performed by the ECU M&C toolkit using a predefined algorithm (XCP only) or over a dedicated checksum function provided by a specific DLL. The Checksum DLL is defined in the A2L data base and can be changed by the application by the MC Set Property.vi using the Checksum DLL Name property.

If you are using the CCP protocol, type of checksum must always be set to 0xFF, as CCP supports an external checksum DLL only. If using XCP, the following values for type of checksum are defined in the XCP specification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>XCP_ADD_11</td>
<td>Add BYTE into a BYTE checksum, ignore overflows</td>
</tr>
<tr>
<td>0x02</td>
<td>XCP_ADD_12</td>
<td>Add BYTE into a WORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x03</td>
<td>XCP_ADD_14</td>
<td>Add BYTE into a DWORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x04</td>
<td>XCP_ADD_22</td>
<td>Add WORD into a WORD checksum, ignore overflows, blocksize must be modulo 2</td>
</tr>
<tr>
<td>0x05</td>
<td>XCP_ADD_24</td>
<td>Add WORD into a DWORD checksum, ignore overflows, blocksize must be modulo 2</td>
</tr>
</tbody>
</table>
For a detailed description of the checksum algorithm, refer to the MC Build Checksum.vi or the XCP Part 2 Protocol Layer Specification.

For more detailed information about CRC algorithms, refer to the following site:
http://www.repairfaq.org/filipg/LINK/F_crc_v34.html
MC CCP Action Service.vi

Purpose

Calls an implementation-specific action service on the ECU (CCP only).

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>Service No</th>
<th>Params</th>
<th>ECU ref out</th>
<th>Error in</th>
<th>Error out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Service No** determines the service that is executed inside the ECU. For more information about the services that are implemented in the ECU, refer to the documentation for the ECU.

**Params** passes an array to the ECU that might be needed by the ECU to run the service. Since this VI has no knowledge about how the data is interpreted by the ECU, you are responsible for providing the data in the correct byte ordering.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

**status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.
Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Data type** is a data type qualifier that determines the data format of the result.

**Result** returns information from the action service.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

**status** is **TRUE** if an error occurred.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.

Description

**MC CCP Action Service.vi** implements the CCP command ACTION_SERVICE. The ECU carries out the requested service and automatically uploads the requested action service return information.
MC CCP Diag Service.vi

Purpose
Calls a diagnostic service on the ECU (CCP only).

Format

Input

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Service no determines the diagnostic service that is executed inside the ECU. For more information about the services that are implemented in the ECU, refer to the documentation for the ECU.

Params passes an array to the ECU that might be needed by the ECU to run the service. Since this VI has no knowledge about how the data is interpreted by the ECU, you are responsible for providing the data in the correct byte ordering.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Data Type** returns a Data Type Qualifier which provides information about the data type of the result of the diagnostic service.

**Result** contains the information returned from the diagnostic service, uploaded from the ECU by the CCP master.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is TRUE if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

Description

**MC CCP Diag Service.vi** implements the CCP command DIAG_SERVICE, which starts a diagnostic service on the ECU and waits until it is finished. The selected **Service no** specifies the diagnostic service that is executed inside the ECU. For more information about the available services that are implemented in the ECU, refer to the documentation for the ECU.
MC CCP Get Active Cal Page.vi

Purpose
Retrieves the ECU Memory Transfer Address pointer to the calibration data page (CCP only).

Format

**Input**

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

  - **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.

  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

  - **source** identifies the VI where the error occurred.

**Output**

- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

  - **Address** is a cluster which contains the following values.

    - **Address** specifies the address part of the active calibration page address.
Extension contains the extension part of the active calibration page address.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is True if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC CCP Get Active Cal Page.vi retrieves the ECU Memory Transfer Address pointer of the active calibration data page.

MC CCP Get Active Cal Page.vi implements the CCP command GET_ACTIVE_CAL_PAGE defined by the CCP specification.
MC CCP Get Result.vi

Purpose
Uploads requested data (CCP only).

Format

<table>
<thead>
<tr>
<th>Input</th>
<th>ECU ref in</th>
<th>block size</th>
<th>error in</th>
<th>ECU ref out</th>
<th>data</th>
<th>error out</th>
</tr>
</thead>
</table>

**Input**
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **Block size** is the size of the data block, in bytes, to be uploaded.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

**Output**
- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.
- **Data** is a byte array which receives the uploaded data information from the ECU.
Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC CCP Get Result.vi uploads data bytes from the ECU. It is assumed that the Memory Transfer Address 0 (MTA0) has been set by a previous VI like MC Generic.vi with the command SET_MTA.
MC CCP Get Session Status.vi

Purpose
Retrieves the current calibration status of the ECU (CCP only).

Format

Input

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

status qualifier describes an additional status qualifier. The additional status qualifier is manufacturer and/or project specific and is not part of the CCP protocol specification.

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.
session status is the actual session status which is returned from the ECU.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

additional status describes an additional status qualifier. If the status qualifier does not contain additional status information, the additional status parameter must be set to FALSE. If the additional status parameter is not FALSE, it may be used to determine the type of the additional status information.

Description

MC CCP Get Session Status.vi retrieves the session status of the ECU. The return value session status is a bit mask that represents several session states inside the ECU. status qualifier specifies the additional status information. additional status contains the additional status information. The content of these parameters is project specific and not defined by CCP. For more information about these parameters, refer to the documentation for the ECU.

MC CCP Get Session Status.vi implements the CCP command GET_S_STATUS defined by the CCP specification.
MC CCP Get Version.vi

Purpose
Retrieves version of the CCP implemented in the ECU (CCP only).

Format

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>ECU ref out</td>
</tr>
<tr>
<td>Error in</td>
<td>major version</td>
</tr>
<tr>
<td>status</td>
<td>minor version</td>
</tr>
<tr>
<td>code</td>
<td>error out</td>
</tr>
<tr>
<td>source</td>
<td></td>
</tr>
</tbody>
</table>
```

**Input**

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

**Output**

- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

- **Major version** returns the major version number of the CCP implementation.

- **Minor version** returns the minor version number of the CCP implementation.
Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC CCP Get Version.vi can be used to query the CCP version implemented in the ECU. This command performs a mutual identification of the protocol version in the slave device to agree on a common protocol version.

MC CCP Get Version.vi implements the CCP command GET_CCP_VERSION defined by the CCP specification.
MC CCP Move Memory.vi

Purpose

Moves a memory block on the ECU (CCP only).

Format

```
block size
ECU ref in
source
destination
error in
ECU ref out
error out
```

Input

**Block size** determines the size of memory block in bytes which should be moved from the source address to the destination address.

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Source** is a cluster which contains the following values.

- **Address** specifies the address part of the source address from which the memory block is copied.
- **Extension** specifies the extension part of the source address.

**Destination** is a cluster which contains the following values.

- **Address** specifies the address part of the destination address to which the memory block is copied.
- **Extension** specifies the extension part of the destination address.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC CCP Move Memory.vi is used to move the memory contents of an ECU from one memory location to another. Before calling the CCP MOVE command this function sets the Memory Transfer Address pointers MTA0 as defined in the source cluster and MTA1 as defined in the destination cluster to appropriate values.

MC CCP Move Memory.vi implements the CCP command MOVE defined by the CCP specification.
Chapter 5  ECU M&C API for LabVIEW

MC CCP Select Cal Page.vi

Purpose
Sets the beginning of the calibration data page (CCP only).

Format

```
<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>address</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error in</td>
<td></td>
<td>error out</td>
</tr>
<tr>
<td>status</td>
<td></td>
<td></td>
</tr>
<tr>
<td>code</td>
<td></td>
<td></td>
</tr>
<tr>
<td>source</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Input

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Address** is a cluster which contains the following values.

- **Address** specifies the address part of the address.
- **Extension** contains the extension part of the address.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC CCP Select Cal Page.vi implements the CCP command SELECT_CAL_PAGE. The operation of the command depends on the ECU implementation.
**MC CCP Set Session Status.vi**

**Purpose**
Updates the ECU with the current state of the calibration session (CCP only).

**Format**

```
ECU ref in  session status  ECU ref out
              error in          error out
```

**Input**

<table>
<thead>
<tr>
<th><strong>ECU ref in</strong></th>
<th>is the task reference which links to the selected ECU. This reference is originally returned from <code>MC ECU Open.vi</code> or <code>MC ECU Select.vi</code>, and then wired through subsequent VIs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Session status</strong></td>
<td>is the new status to be set in the ECU.</td>
</tr>
<tr>
<td><strong>Error in</strong></td>
<td>is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the <code>error in</code> cluster to <code>error out</code>.</td>
</tr>
<tr>
<td><strong>status</strong></td>
<td>is TRUE if an error occurred. This VI is not executed when <code>status</code> is TRUE.</td>
</tr>
<tr>
<td><strong>code</strong></td>
<td>is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the <code>code</code>, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.</td>
</tr>
<tr>
<td><strong>source</strong></td>
<td>identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>

**Output**

| **ECU ref out** | is the same as `ECU ref in`. Wire the task reference to subsequent VIs for this task. |
| **Error out** | describes error conditions. If the `Error in` cluster indicated an error, the `Error out` cluster contains the same information. Otherwise, `Error out` describes the error status of this VI. |
status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

### Description

This VI implements the CCP SET_S_STATUS command and is used to keep the ECU informed about the current state of the calibration session. The session status bits of an ECU can be read and written. Possible conditions are: reset on power-up, session log-off, and in applicable error conditions. The calibration session status is organized as a bit mask with the following assignment.

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAL</td>
<td>Calibration data initialized.</td>
</tr>
<tr>
<td>1</td>
<td>DAQ</td>
<td>DAQ list(s) initialized.</td>
</tr>
<tr>
<td>2</td>
<td>RESUME</td>
<td>Request to save DAQ set-up during shutdown in CCP slave.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CCP slave automatically restarts DAQ after start-up.</td>
</tr>
<tr>
<td>3</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>STORE</td>
<td>Request to save calibration data during shut-down in CCP slave.</td>
</tr>
<tr>
<td>7</td>
<td>RUN</td>
<td>Session in progress.</td>
</tr>
</tbody>
</table>
MC Characteristic Read.vi

Purpose
Reads data from a named Characteristic on the ECU which is identified by the ECU Reference handle. The Poly VI returns a specific double, 1D, or 2D double array.

Format

<table>
<thead>
<tr>
<th>Characteristic name</th>
<th>ECU ref in</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td></td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

Characteristic name is the name of the Characteristic defined in the A2L database.

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Characteristic is a poly output value which represents the data read from the ECU. The type of the poly output is determined by the poly VI selection. For information on the different poly VI types provided by MC Characteristic Read.vi, refer to the Poly VI Types section.

To select the data type, right-click the VI, go to Select Type, and select the type by name.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Poly VI Types

Table 5-3. Poly VI Types for the Value Parameter

<table>
<thead>
<tr>
<th>VI Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter (DBL)</td>
<td>Returns a single double value for the selected Characteristic name.</td>
</tr>
<tr>
<td>Curve (1D)</td>
<td>Returns a 1-dimensional array of double values for the selected Characteristic name.</td>
</tr>
<tr>
<td>Field (2D)</td>
<td>Returns a 2-dimensional array of double values for the selected Characteristic name.</td>
</tr>
</tbody>
</table>
MC Characteristic Read Single Value.vi

Purpose
Reads a value from a named Characteristic on the ECU which is identified by the ECU Reference handle.

Format

<table>
<thead>
<tr>
<th>Characteristic name</th>
<th>ECU ref in</th>
<th>x</th>
<th>y</th>
<th>ECU ref out</th>
<th>characteristic value</th>
<th>Error in</th>
<th>Error out</th>
<th>status</th>
<th>code</th>
<th>source</th>
</tr>
</thead>
</table>

Input

- **Characteristic name** is the name of the Characteristic defined in the A2L database.
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **x** is the horizontal index if the Characteristic consists of 1 or 2 dimensions.
- **y** is the vertical index if the Characteristic consists of 2 dimensions.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
- **status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.
Output

- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.
- **Characteristic value** returns a single sample for the specified Characteristic.
- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.
  - **status** is **TRUE** if an error occurred.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

Description

**MC Characteristic Read Single Value.vi** reads a value from a specified Characteristic on the ECU which is identified by the ECU Reference handle. The value to be read is identified by the x and y indices. If the Characteristic array has 0 or 1 dimensions, y and/or x can be left unwired.
MC Characteristic Write.vi

Purpose
Writes the value(s) of a named Characteristic to an ECU identified by the ECU ref handle. The Poly VI writes the selected type double, 1D or 2D array.

Format

Input

Characteristic name is the name of a Characteristic stored in the A2L database file to which one or more values may be written.

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Characteristic writes the data for the Characteristic channel initialized by Characteristic name. Characteristic values are listed in the Poly VI Types section.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Poly VI Types

Table 5-4. Poly VI Types for the Characteristic Parameter

<table>
<thead>
<tr>
<th>VI Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter (DBL)</td>
<td>Writes a single double value to the selected Characteristic name.</td>
</tr>
<tr>
<td>Curve (1D)</td>
<td>Writes a 1-dimensional array of double values to the selected Characteristic name.</td>
</tr>
<tr>
<td>Field (2D)</td>
<td>Writes a 2-dimensional array of double values to the selected Characteristic name.</td>
</tr>
</tbody>
</table>
MC Characteristic Write Single Value.vi

Purpose

Writes a value to a named Characteristic on the ECU.

Format

- **characteristic name** is the name of a Characteristic stored in the A2L database file to which one value may be written.
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **x** is an input that refers to the array offset if the Characteristic is defined in the A2L database file as 1- or 2-dimensional. If the Characteristic is defined as having 0 dimensions, the input can be left unwired.
- **y** is an input that refers to the array offset if the Characteristic is defined in the A2L database file as 2-dimensional. If the Characteristic is defined as having 0 or 1 dimensions, the input can be left unwired.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
- **status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
source identifies the VI where the error occurred.

**Characteristic value** is the value to be set for the Characteristic.

### Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

**status** is TRUE if an error occurred.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.

### Description

**MC Characteristic Write Single Value.vi** writes a value to a defined Characteristic on the ECU which is identified by the ECU Reference handle. The location to which the value is written is identified by the x and y indices. If the Characteristic array has 0 or 1 dimensions, y and/or x can be left unwired.
MC Clear Memory.vi

Purpose
Clears the contents of a specified memory block.

Format

<table>
<thead>
<tr>
<th>Input</th>
<th>ECU ref in</th>
<th>Address</th>
<th>Extension</th>
<th>Block size</th>
<th>Error in</th>
<th>Status</th>
<th>Code</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>U32</td>
<td>Address</td>
<td>Block size</td>
<td></td>
<td>Error in</td>
<td></td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Status</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Code</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Source</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Address is a cluster which contains the following values.

- **Address** specifies the address part of the source address.
- **Extension** contains the extension part of the source address.

Block size determines the size of the block that must be cleared.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

- **status** is TRUE if an error occurred. This VI is not executed when status is TRUE.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.
- **source** identifies the VI where the error occurred.
Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- status is **TRUE** if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

Description

**MC Clear Memory.vi** can be used to erase the FLASH EPROM prior to reprogramming. If you are using CCP, the CCP Memory Transfer address (MTA0) pointer is set to the memory location to be erased specified by the parameters **Address** and **Extension**. **MC Clear Memory.vi** implements the CCP CLEAR_MEMORY command defined by the CCP specification.

If you are using the XCP protocol, **MC Clear Memory.vi** implements the PROGRAM_CLEAR command.

For further details on how to clear parts of non-volatile memory in the ECU refer to the ASAM XCP Protocol Layer Specification.
MC Conversion Create.vi

Purpose

Creates a signal conversion object in memory.

Format

```
Conversion Name
ECU ref in
Factor
Offset
error in
Unit
ECU ref out
error out
```

Input

**Conversion Name** identifies the conversion object that handles the scaling of a measurement.

**ECU ref in** is the task reference that links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Create.vi**.

**Factor** configures the scaling factor used to convert raw measurement data in the message to/from scaled floating-point units. The factor is the $A$ in the linear scaling formula $AX+B$, where $X$ is the raw data, and $B$ is the scaling offset.

**Offset** configures the scaling offset used to convert raw data in the measurement message to/from scaled floating-point units. The scaling offset is the $B$ in the linear scaling formula $AX+B$, where $X$ is the raw data, and $A$ is the scaling factor.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI...
executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Unit configures the measurement channel unit string. You can use this value to display units (such as volts or RPM) along with the samples of the channel.

Output

ECU ref out is the task reference that links to the selected ECU.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Use MC Conversion Create.vi to create a conversion object in memory instead of referring to measurement properties defined in the A2L database.
MC DAQ Clear.vi

**Purpose**

Stops communication for the Measurement task and then clears the configuration.

**Format**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DAQ ref in</strong></td>
<td><strong>ECU ref out</strong></td>
</tr>
<tr>
<td><strong>error in</strong></td>
<td><strong>Error out</strong></td>
</tr>
</tbody>
</table>

**Input**

- **DAQ ref in** is the task reference which links to the Measurement task. This reference is originally returned from **MC DAQ Initialize.vi**, and then wired through subsequent VIs.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is **TRUE** if an error occurred. Unlike other VIs, this VI will execute when **status** is **TRUE**.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

**Output**

- **ECU ref out** is the ECU reference upon which **MC DAQ Initialize.vi** was called. Wire this to subsequent ECU operations.
- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.
  - **status** is **TRUE** if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC DAQ Clear.vi must always be the final ECU M&C VI called for a Measurement task. If you do not use the MC DAQ Clear.vi, the remaining task configurations can cause problems in execution of subsequent ECU M&C applications.

Because this VI clears the Measurement task, the Measurement task reference is not wired as an output but is transferred into an ECU reference task handle. To change properties of a running Measurement task, use MC DAQ Start Stop.vi to stop the task, MC Set Property.vi to change the desired DAQ property, and then MC DAQ Start Stop.vi to restart the Measurement task again.
MC DAQ Initialize.vi

Purpose
Initializes a Measurement task for the specified Measurement channel list.

Format

```
measurement list
ECU ref in
mode
sample rate
error in
DTO ID
```

Input

- **Measurement list** is the array of channel names to initialize as a Measurement task. Each channel name is provided in an array entry.

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **Mode** specifies the I/O mode for the task. For an overview of the I/O modes, including figures, refer to the *Basic Programming Model* section of Chapter 4, *Using the ECU M&C API*.

  Mode=0

  **DAQ List**: The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with the **MC DAQ Read.vi** as Single point data using a sample rate = 0 or as waveform using a sample rate > 0. Input channel data are received from the DAQ messages. Use **MC DAQ Read.vi** to obtain input samples as single-point, array, or waveform.

  Mode=1

  **Polling**: In this mode the data from the Measurement task are acquired from the ECU whenever the **MC DAQ Read.vi** is called.

  Mode=2
**STIM List**: In this mode the data from the Measurement task are sent to the ECU whenever `MC DAQ Write.vi` is called.

**Mode = 3**

**Timestamped read**: The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with `MC DAQ Read.vi` as timestamped data array. Input channel data are received from the DAQ messages. Use `MC DAQ Read.vi` to obtain input samples as an array of sample/timestamp pairs (poly VI types ending in Timestamped Dbl). Use this input mode to read samples with timestamps that indicate when each channel is received from the network.

**Sample rate** specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately. If the **Mode** is defined as DAQ list, a sample rate of zero means that `MC DAQ Read.vi` returns a single point from the most recent message received, and greater than zero means that `MC DAQ Read.vi` returns samples timed at the specified rate. If the **Mode** is defined as Polling, the sample rate is ignored.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

**DTO_ID** is the CAN identifier for the Data Transmission Object (DTO) used by the ECU to transmit the DAQ list data to the host. If the **DTO_ID** terminal is unwired the ECU will use the same identifier for sending the DAQ list data as for the normal CCP communication.
Output

**DAQ ref out** is a task reference for the Measurement task created. Wire this task reference to subsequent VIs for this Measurement task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

Description

**MC DAQ Initialize.vi** does not start the transmission of the DAQ lists from the ECU to or from the application through CCP or XCP. This enables you to use **MC Set Property.vi** to change the properties of a Measurement task. After you change properties use **MC DAQ Start Stop.vi** to start the communication for the Measurement task.
MC DAQ List Initialize.vi

Purpose
Defines a DAQ list on a specific DAQ list number and initializes the Measurement task for the specified Measurement channel list.

Format

```
DAQ List No
measurement list
ECU ref in
node
sample rate
error in
DTO_ID
```

Input

**DAQ List No** specifies which DAQ list entry number should be used for the defined Measurement channel list for the selected ECU. To query the available DAQ List numbers on the ECU use **MC Get Property.vi** and select **DAQ List Number** in the Poly VI.

**Measurement list** is the array of channel names to initialize as a Measurement task. Each channel name is provided in an array entry.

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Mode** specifies the I/O mode for the task. For an overview of the I/O modes, including figures, refer to the **Basic Programming Model** section of Chapter 4, *Using the ECU M&C API*.

**Mode=0**

**DAQ List**: The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with the **MC DAQ Read.vi** as Single point data using a sample rate = 0 or as waveform using a sample rate > 0. Input channel data are received from the DAQ messages. Use **MC DAQ Read.vi** to obtain input samples as single-point, array, or waveform.
Mode = 1

**Polling:** In this mode the data from the Measurement task are acquired from the ECU whenever the `MC DAQ Read.vi` is called.

Mode = 2

**STIM List:** In this mode the data from the Measurement task are sent to the ECU whenever `MC DAQ Write.vi` is called.

Mode = 3

**Timestamped read:** The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with `MC DAQ Read.vi` as timestamped data array. Input channel data are received from the DAQ messages. Use `MC DAQ Read.vi` to obtain input samples as an array of sample/timestamp pairs (Poly VI types ending in Timestamped Dbl). Use this input mode to read samples with timestamps that indicate when each channel is received from the network.

**Sample rate** specifies the timing to use for samples of the task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately. If the **Mode** is defined as **DAQ List**, a sample rate of zero means that `MC DAQ Read.vi` returns a single point from the most recent message received, and greater than zero means that `MC DAQ Read.vi` returns samples timed at the specified rate. If the **Mode** is defined as **Polling**, the sample rate is ignored.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

**status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.
**User Manual 5-48**

**Chapter 5 ECU M&C API for LabVIEW**

**DTO_ID** is the CAN identifier for the **Data Transmission Object (DTO)** used by the ECU to transmit the DAQ list data to the host. If the **DTO_ID** terminal is unwired the ECU will use the same identifier for sending the DAQ list data as for the normal CCP communication.

**Output**

**DAQ ref out** is a task reference for the Measurement task created. Wire this task reference to subsequent VIs for this Measurement task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

**Description**

If an ECU offers a reduced and specific range of DAQ list entry numbers use **MC DAQ List Initialize.vi** to setup your Measurement list. **MC DAQ List Initialize.vi** does not start the transmission of the DAQ lists from the ECU to the application or vice versa through CCP or XCP. This enables you to use **MC Set Property.vi** to change the properties of a Measurement task. After you change properties use **MC DAQ Start Stop.vi** to start the communication for the Measurement task. To query the available DAQ list entry numbers use **MC Get Property.vi** with the Poly option selection **DAQ List Numbers**.
MC DAQ Read.vi

Purpose
Reads samples from a Measurement task.

Format

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAQ ref in</td>
<td>is the task reference from the previous Measurement task VI. The task reference is originally returned from MC DAQ Initialize.vi, and then wired through subsequent Measurement task VIs.</td>
</tr>
<tr>
<td>Number of samples</td>
<td>specifies the number of samples to read for the Measurement task. For single-sample Poly VI types, MC DAQ Read.vi always returns one sample, so this input is ignored.</td>
</tr>
<tr>
<td>Error in</td>
<td>is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td>status</td>
<td>is TRUE if an error occurred. This VI will not execute when status is TRUE.</td>
</tr>
<tr>
<td>code</td>
<td>is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.</td>
</tr>
<tr>
<td>source</td>
<td>identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>

© National Instruments
Output

**Output**

- **DAQ ref out** is the same as **DAQ ref in**. Wire the task reference to subsequent VIs for this task.

- **Number of samples returned** indicates the number of samples returned in the samples output.

- **Value** is a poly output that returns the samples read from the received CAN messages of the DAQ list. The type of the poly output is determined by the poly VI selection. For information on the different poly VI types provided by **MC DAQ Read.vi**, refer to the *Poly VI Types* section.

To select the data type, right-click the VI, go to **Select Type**, and select the type by name.

- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

  - **status** is **TRUE** if an error occurred.
  - **code** is the error code number identifying an error. A value of **0** means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

Description

**Poly VI Types**

The name of each Poly VI type uses the following conventions:

- The first term is either **1Chan** or **NChan**. This indicates whether the type returns data for a single channel or multiple channels. **NChan** types return an array of analogous **1Chan** types, one entry for each channel initialized in channel list of **MC DAQ Initialize.vi**. **1Chan** types are convenient because no array indexing is required, but you are limited to reading only one channel.

- The second term is either **1Samp** or **NSamp**. This indicates whether the type returns a single sample, or an array of multiple samples. **1Samp** types are often used for single point control applications, such as within LabVIEW RT.

- The third term indicates the data type used for each sample. The type **Dbl** indicates double-precision (64-bit) floating point. The type **Wfm** indicates the waveform data type. The types **1D** and **2D** indicate one and two-dimensional arrays, respectively.
1Chan 1Samp Dbl
Returns a single sample for the first channel initialized in channel list. If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, and then returns a single sample. This enables you to execute a control loop at a specific rate. If the initialized sample rate is zero, this poly VI immediately returns a single sample. The samples output returns a single sample from the most recent message received. If no message has been received since you started the task, the value of 0 is returned in samples. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received, the warning code 3FF60009 hex is returned in error out. If a new message has been received, the success code 0 is returned in error out. Unless an error occurs, number of samples returned is one.

NChan 1Samp 1D Dbl
Returns an array, one entry for each channel initialized in channel list. Each entry consists of a single sample. The order of channel entries in samples is the same as the order in the original channel list. If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, then returns a single sample for each channel. This enables you to execute a control loop at a specific rate. If the initialized sample rate is zero, this poly VI immediately returns a single sample for each channel. The samples output returns a single sample for each channel from the most recent message received. If no message has been received for a channel since you started the task a 0 is returned in samples. You can specify channels in channel list that span multiple messages. A sample from the most recent message is returned for all channels. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received for one or more channels, the warning code 3FF60009 hex is returned in error out. If a new message has been received for all channels, the success code 0 is returned in error out. Unless an error occurs, number of samples returned is one. The samples array is indexed by channel, and the entry for each channel contains a single sample. If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

1Chan NSamp 1D Dbl
Returns an array of samples for the first channel initialized in channel list. The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of the CAN channel at a specific point in time. In other words, the sample rate specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels. This VI waits until all samples arrive in time before returning.
If the initialized sample rate is zero, this poly VI returns an error. If the intent is simply to read the most recent sample for a task, use the 1Chan 1Samp Dbl type. If no message has been received since you started the task a 0 is returned in all entries of the samples array. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received, the warning code 3FF60009 hex is returned in error out. If a new message has been received, the success code 0 is returned in error out. Unless an error occurs, the number of samples returned is equal to number of samples to read.

NChan NSamp 2D Dbl

Returns an array, one entry for each channel initialized in channel list. Each entry consists of an array of value. The order of channel entries in value is the same as the order in the original channel list. The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of each CAN channel at a specific point in time. In other words, the sample rate specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channels over time, such as for comparison with other CAN or DAQ input channels. This VI waits until all samples arrive in time before returning.

If the initialized sample rate is zero, this poly VI returns an error. If the intent is simply to read the most recent samples for a task, use the NChan 1Samp 1D Dbl type. If no message has been received for a channel since you started the task, the Default Value of the channel is returned in value. You can specify channels in channel list that span multiple messages. At each point in time, a sample from the most recent message is returned for all channels. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received for one or more channels, the warning code 3FF60009 hex is returned in error out. If a new message has been received for all channels, the success code 0 is returned in error out. Unless an error occurs, the number of samples returned is equal to number of samples to read. If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

1Chan NSamp Wfm

Returns a single waveform for the first channel initialized in channel list. The initialized sample rate must be greater than zero for this poly VI, because each sample in the array indicates the value of the CAN channel at a specific point in time. In other words, the sample rate specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the CAN channel over time, such as for comparison with other CAN or DAQ input channels. This VI waits until all samples arrive in time before returning. The start time of a waveform indicates the time of the first CAN sample in the array. The delta time of a waveform indicates the time between each sample in the array, as determined by the original sample rate.
If the initialized sample rate is zero, this poly VI returns an error. If the intent is simply to read the most recent sample for a task, use the 1Chan 1Samp Dbl type. If no message has been received since you started the task a 0 is returned in all entries of the value waveform. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received, the warning code 3FF60009 hex is returned in error out. If a new message has been received, the success code 0 is returned in error out. Unless an error occurs, the number of samples returned is equal to number of samples to read.

**NChan NSamp 1D Wfm**

Returns an array, one entry for each channel initialized in channel list. Each entry consists of a single waveform. The order of channel entries in value is the same as the order in the original channel list. The initialized sample rate must be greater than zero for this poly VI, because each sample in the array of a waveform indicates the value of the CAN channel at a specific point in time. In other words, the sample rate specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the M&C DAQ channel over time, such as for comparison with other CAN or DAQ input channels. This VI waits until all samples arrive in time before returning. The start time for each waveform indicates the time of the first CAN sample in the array. The delta time of a waveform indicates the time between each sample in the array, as determined by the original sample rate.

If the initialized sample rate is zero, this poly VI returns an error. If the intent is simply to read the most recent samples for a task, use the NChan 1Samp 1D Dbl type. If no message has been received for a channel since you started the task a 0 is returned in value. You can specify channels in channel list that span multiple messages. At each point in time, a sample from the most recent message is returned for all channels. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Read.vi (or MC DAQ Start Stop.vi). If no message has been received for one or more channels, the warning code 3FF60009 hex is returned in error out. If a new message has been received for all channels, the success code 0 is returned in error out. Unless an error occurs, the number of samples returned is equal to number of samples to read. If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.

**MC Read Multi Chan Multi Samp 2D Time & Dbl**

Returns an array with one entry for each channel initialized in the measurement list. Each entry consists of an array of clusters. Each cluster corresponds to a received signal for the channels initialized in the measurement list. Each cluster contains the sample value and a timestamp that indicates when the measurement channel was received. The order of channel entries in samples is the same as the order in the original channel list. To use this type, you must set the initialized mode to timestamped read. The VI does not wait for messages, but instead returns samples from the messages received since the previous call to CAN Read.vi.
The number of samples returned is indicated in the **number of samples returned** output, up to a maximum of **number of samples** to read messages. If no new message has been received, the **number of samples returned** is 0, and **error out** indicates success.

Because the timing of values in samples is determined by when the message is received, the **sample rate** input is not used with this poly VI type. To determine the number of channels in the task after initialization, get the **Number of Channels** property for the task reference.
MC DAQ Start Stop.vi

Purpose

Starts or stops transmission of the DAQ lists for the specified Measurement task.

Format

Input

DAQ ref in is the task reference from the previous Measurement task VI. The task reference is originally returned from MC DAQ Initialize.vi, and then wired through subsequent Measurement task VIs.

mode indicates the type of function to be performed.

Stop DAQ List

Configures the ECU to stop transmitting a DAQ task. If stopped, properties of the DAQ task can be changed using MC Set Property.vi. This function is performed automatically before MC DAQ Clear.vi.

Start DAQ List

Configures the ECU to start sending data for a DAQ task. Ensure that the DAQ list has not yet been transferred to the ECU first. Once started, properties of the DAQ list can no longer be changed using MC Set Property.vi. This function is performed automatically before the first read of the DAQ list with MC DAQ Read.vi.

Transmit DAQ List to ECU

Transfers the DAQ list to the ECU, but does not start it. For example, use this mode if you want to change the session status before starting the DAQ list. For some ECUs this is necessary.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

DAQ ref out is the same as DAQ ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC DAQ Start Stop.vi is optional to start or stop transmission of the DAQ lists for an M&C Measurement task to use MC DAQ Read.vi. If you do not specify MC DAQ Start Stop.vi (Start DAQ list) before your first Read VI, it is implicitly performed by the first MC DAQ Read.vi call.

After you start the transmission of the DAQ lists, you can no longer change the configuration of the task with MC Set Property.vi. You must call MC DAQ Start Stop.vi (Stop DAQ list) first.
MC DAQ Write.vi

Purpose

Writes samples to an ECU DAQ list.

Format

```
| Input                  | | Description                        |
|------------------------||------------------------------------|
| DAQ ref in             | | The task reference from the previous Measurement task VI. The task reference is originally returned from MC DAQ Initialize.vi, and then wired through subsequent Measurement task VIs. |
| number of samples      | | Specifies the number of samples to write for the Measurement task. For single-sample Poly VI types, MC DAQ Write.vi always returns one sample, so this input is ignored. |
| Value                  | | A poly output that writes samples to the ECU STIM list. The type of the poly output is determined by the poly VI selection. For information on the different poly VI types provided by MC DAQ Write.vi, refer to the Poly VI Types section. |
| Error in               | | A cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out. |

status is TRUE if an error occurred. This VI will not execute when status is TRUE. |

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler. |
source identifies the VI where the error occurred.

Output

DAQ ref out is the same as DAQ ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Poly VI Types

The name of each Poly VI type uses the following conventions:

- The first term is either 1Chan or NChan. This indicates whether the type writes data to a single channel or multiple channels. NChan types write an array of analogous 1Chan types, one entry for each channel initialized in channel list of MC DAQ Initialize.vi. 1Chan types are convenient because no array indexing is required, but you are limited to writing only one channel.

- The second term is either 1Samp or NSamp. This indicates whether the type writes a single sample, or an array of multiple samples. 1Samp types are often used for single point control applications, such as within LabVIEW RT.

- The third term indicates the data type used for each sample. The type Dbl indicates double-precision (64-bit) floating point. The type Wfm indicates the waveform data type. The types 1D and 2D indicate one and two-dimensional arrays, respectively.
1Chan 1Samp Dbl
Writes a single sample for the first channel initialized in the channel list. If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, and then writes a single sample. This enables you to execute a control loop at a specific rate. If the initialized sample rate is zero, this poly VI immediately writes a single sample. If no message has been received since you started the task, the value of 0 is returned in samples. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Write.vi (or MC DAQ Start Stop.vi). If no message has been received, the warning code 3FF60009 hex is returned in error out. If a new message has been received, the success code 0 is returned in error out.

NChan 1Samp 1D Dbl
Writes an array, one entry for each channel initialized in the channel list. Each entry consists of a single sample. The order of channel entries in samples is the same as the order in the original channel list. If the initialized sample rate is greater than zero, this poly VI type waits for the next sample time, then writes a single sample for each channel. This enables you to execute a control loop at a specific rate. If the initialized sample rate is zero, this poly VI immediately writes a single sample for each channel. The samples output returns a single sample for each channel from the most recent message received. If no message has been received for a channel since you started the task a 0 is returned in samples. You can specify channels in channel list that span multiple messages. A sample from the most recent message is returned for all channels. You can use error out to determine whether a new message has been received since the previous call to MC DAQ Write.vi (or MC DAQ Start Stop.vi). If no message has been received for one or more channels, the warning code 3FF60009 hex is returned in error out. If a new message has been received for all channels, the success code 0 is returned in error out. Unless an error occurs, number of samples returned is one. The samples array is indexed by channel, and the entry for each channel contains a single sample. If you need to determine the number of channels in the task after initialization, get the Number of Channels property for the task reference.
MC Database Close.vi

Purpose
Closes a specified A2L Database.

Format

```
DB ref in
error in
error out
```

Input

- **DB reference in** is the task reference from the initial database task VI. The task reference is originally returned from MC Database Open.vi.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.
  - **status** is TRUE if an error occurred. Unlike other VIs, this VI will execute when status is TRUE.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.
  - **source** identifies the VI where the error occurred.

Output

- **Error out** describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.
  - **status** is TRUE if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Database Close.vi must always be the final M&C VI called for each communication task. If you do not use MC Database Close.vi, the remaining task configurations can cause problems in the execution of subsequent Measurement and Calibration applications.

MC Database Close.vi is an advanced function for database handling. In most cases it is sufficient to use MC ECU Close.vi instead.

Unlike other VIs, MC Database Close.vi will execute when status is TRUE in Error in. Because this VI clears the task, the task reference is not wired as an output.
MC Database Create.vi

Purpose

Creates an ASAM A2L database in memory.

Format

```
DB name  DB ref out
error in  error out
```

Input

- **DB name** is a database name associated with the database created in memory. Use the string syntax `:<myname>:` for the A2L database if using multiple databases in memory. (For example, if using two databases in memory, use `:MyDatabase1:` as the DB name for the first database and `:MyDatabase2:` for the second DB name created in memory.)

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

- **status** is TRUE if an error occurred. This VI is not executed when status is TRUE.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

- **source** identifies the VI where the error occurred.

Output

- **DB ref out** is the task reference that links to the opened database file.

- **Error out** describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

- **status** is TRUE if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Creates an A2L Database. Use MC Database Create.vi to create ECU and measurement objects in memory, if you do not have access to a valid A2L database file.

MC Database Create.vi does not start communication. After creating an A2L database in memory, you typically create an ECU object using MC ECU Create.vi, a scaling object using MC Conversion Create.vi, a measurement object using MC Measurement Create.vi, and an event using MC Event Create.vi.

Note  MC Database Create.vi does not support creating objects to access characteristics. To access a characteristic, assign a valid A2L database file with defined characteristics.
MC Database Open.vi

**Purpose**

Opens a specified A2L Database.

**Format**

```
<table>
<thead>
<tr>
<th>DB path</th>
<th>D6 ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>
```

**Input**

- **DB path** is a path to a A2L database file from which to get channel names. The file must use a `.A2L` extension. You can generate A2L database files with several 3rd party tools.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the `error in` cluster to `error out`.

  - `status` is **TRUE** if an error occurred. This VI will not execute when `status` is **TRUE**.
  
  - `code` is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the `code`, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

  - `source` identifies the VI where the error occurred.

**Output**

- **DB reference out** is the task reference which links to the opened database file.

- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

  - `status` is **TRUE** if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

Opens a specified A2L Database. **MC Database Open.vi** enables you to query all defined ECU names in the A2L Database using the **MC Get Names.vi** and selecting the property **ECU Names.** **MC Database Open.vi** does not start communication.

**MC Database Open.vi** is an advanced function for database handling. In most cases it is sufficient to use **MC ECU Open.vi** instead.
**MC Double to Text.vi**

**Purpose**
Converts a numerical value to a text string using an enumeration or range text type scaling.

**Format**

```
object name
ECU ref in
object type
value
error in

ECU ref out
TextValue
error out
```

**Input**

- **object name** indicates the object (measurement or characteristic) for which the COMPU_VTAB scaling is performed. If no COMPU_VTAB scaling is available for the object, **TextValue** is just a string representation of the value specified in value.

- **ECU ref in** is the task reference that links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **object type** is a U32 ring that indicates the type of the object named in **object name**. Valid values are:
  1. Measurement Name
  2. Characteristic Name

- **value** is the numerical value to be converted. For example, this could have been returned from **MC Characteristic Read.vi** or **MC Measurement Read.vi**.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute
the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

**Output**

**ECU ref out** is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

**TextValue** is the resulting converted text string. If the value specified is listed in a COMPU_VTAB scaling for the characteristic or measurement specified in **object name**, the respective text is returned. If no such value is available, a string representation of the double value is returned.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

**status** is TRUE if an error occurred.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

**Description**

**MC Double To Text.vi** performs text conversion for measurement or characteristic values.

Especially if the measurement or characteristic has an associated enumeration or range text type scaling, the textual representation of the value is returned. If no such value is present, either because the object does not have a text scaling or the value does not have a textual representation in the table, a string representation of the double value is returned.
MC Download.vi

Purpose
Downloads data to an ECU.

Format

Input

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **Address** is a cluster which contains the following values.
  - **Address** specifies the address part of the destination address.
  - **Extension** contains the extension part of the destination address.

- **Data** contains the information to be downloaded.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Download.vi is used to download data to an ECU. The data is stored starting at the location specified by the Address and Extension parameters.

On XCP protocol, when the slave supports the block mode, ECU sends the data in blocks using the DOWNLOAD_NEXT command.
MC ECU Close.vi

Purpose
Closes the selected ECU and the associated A2L database.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in</td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

Output

- **Status** is TRUE if an error occurred. Unlike other VIs, this VI will execute when **status** is TRUE.

- **Code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **Source** identifies the VI where the error occurred.

Output

- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **Status** is TRUE if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC ECU Close.vi is the very last VI which must be called. It deselects the ECU and closes the remaining database reference handle. MC ECU Close.vi must always be the final M&C VI. If you do not use MC ECU Close.vi, the remaining task configurations can cause problems in the execution of subsequent M&C applications. If you just want to deselect the ECU connections, call MC ECU Deselect.vi.
Chapter 5  ECU M&C API for LabVIEW

MC ECU Connect.vi

Purpose
Establishes the communication to the selected ECU through the CCP or XCP protocol. After a successful ECU Connect you can create a Measurement Task or read/write a Characteristic.

Format

```
<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>
```

Input

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error in</td>
<td>is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td>status</td>
<td>is TRUE if an error occurred. This VI will not execute when status is TRUE.</td>
</tr>
<tr>
<td>code</td>
<td>is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.</td>
</tr>
<tr>
<td>source</td>
<td>identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>

Output

<table>
<thead>
<tr>
<th>ECU ref out</th>
<th>is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error out</td>
<td>describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.</td>
</tr>
<tr>
<td>status</td>
<td>is TRUE if an error occurred.</td>
</tr>
</tbody>
</table>
**Description**

If you are using the CCP protocol, **MC ECU Connect.vi** implements the CCP CONNECT command. If you are using the XCP protocol, **MC ECU Connect.vi** implements the XCP command CONNECT. It establishes a logical connection to an ECU, using the provided ECU Reference handle. Unless a slave device (ECU) is disconnected, it must not execute or respond to any command sent by the application. Only one CCP slave can be connected to the application at a time from a set of CCP slaves sharing identical CRO and DTO identifiers.

**MC ECU Connect.vi** is an optional function and is automatically performed before **MC Characteristic Read.vi**, **MC Characteristic Write.vi**, **MC DAQ Initialize.vi**, any **MC CCP xxx** command, or any **MC XCP xxx** command is performed.

code is the error code number identifying an error. A value of 0 means success. A negative value means an error: VI did not execute the intended operation. A positive value means a warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.
MC ECU Create.vi

Purpose

Creates an ECU object in memory.

Format

```
Protocol and Interface
ECU Name
DB ref in
Communication Params
Byte Order
error in

ECU ref out
error out

Protocol and Interface
```

Input

**Protocol and Interface** selects target communication protocol CCP or XCP and the desired interface to use for this task. The interface input uses a string `xxx:yyy`, where `xxx` defines one of the two available protocols, CCP or XCP, and `yyy` defines the desired interface to use, such as CAN0 for CCP, or XCP, UDP, or TCP for XCP. The protocol and interface input is required, as this parameter is not defined in the A2L database. The default baud rate for CCP or XCP on CAN, or the IP address for XCP on UDP/TCP, may be defined in the A2L database, but you can change it by setting the Interface Baud Rate or IP Address property with **MC Set Property.vi**.

**NI-CAN**

The special CAN interface values 256 and 257 refer to virtual interfaces. For more information about using virtual interfaces, refer to the *Frame to Channel Conversion* section of Chapter 6, *Using The Channel API*, in the *NI-CAN Hardware and Software User Manual*.

**NI-XNET**

By default, the ECU Measurement and Calibration Toolkit uses NI-CAN for CAN communication. This means you must define an NI-CAN interface for your NI-XNET hardware (NI-CAN compatibility mode) to use your XNET hardware for CAN communication. However, to use your NI-XNET interface in the native NI-XNET mode (meaning it does not use the NI-XNET Compatibility Layer), you must define your interface under **NI-XNET Devices** in MAX and pass the NI-XNET interface name that the ECU Measurement and Calibration Toolkit will use. To do this, add
nixnet to the **Protocol and Interface** string (for example, CCP:CAN1@nixnet). The interface name is related to the NI-XNET hardware naming under **Devices and Interfaces** in MAX.

**Note**  By selecting nixnet as **Protocol and Interface** string, the ECU Measurement and Calibration Toolkit uses the Frame Input and Output Queued sessions. To force the ECU Measurement and Calibration Toolkit to use Frame Input and Output Stream sessions instead, select ni_genie_nixnet as **Protocol and Interface** string (for example, CCP:CAN1@ni_genie_nixnet). An application instance can use only one Frame Input Stream Session and one Frame Output Stream Session at a time, so use the default name nixnet as **Protocol and Interface** string, so that multiple NI-XNET Frame Queued Sessions can coexist on a single interface, and the Frame Input and Output Stream Sessions may be used, for example, for a Frame logging/replay use case.

**CompactRIO or R Series**

If using CompactRIO or R Series hardware, you must provide a bitfile that handles the CAN communication between the host system and FPGA. To access the CAN module on the FPGA, you must specify the bitfile name after the @ (for example, CCP:CAN1@MyBitfile.lvbitx). To specify a special RIO target, you can specify that target by its name followed by the bitfile name (for example, XCP:CAN1@RIO1,MyBitfile.lvbitx). Currently, only a single CAN interface is supported. RIO1 defines the RIO target name as defined in your LabVIEW Project definition. The lvbitx filename represents the filename and location of the bitfile on the host if using RIO or on a CompactRIO target. This implies that you must download the bitfile to the CompactRIO target before you can run your application. You may specify an absolute path or a path relative to the root of your target for the bitfile.

**ECU Name** sets the ECU object name. For all related ECU functions such as **MC ECU Select.vi**, use this name as reference.

**DB ref in** is the task reference that links to the opened database file.

**Communication Params** is a cluster that contains the following values.

- **CRO ID** sets the CAN identifier for the Command Receive Object (CRO) ID, which sends commands and data from the host to the slave device.
- **DTO ID** sets the Data Transmission Object (DTO) ID, which the ECU uses to respond to CCP commands and send data and status information to the CCP master application.
### Baud rate

Sets the baud rate in use by the selected CAN interface. This property applies to all tasks initialized with the NI-CAN interface. You can specify the following basic baud rates as the numeric rate: 33333, 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000. You can specify the advanced baud rate as 8000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1).

### Station Address

Sets the slave device station address. A CCP address is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a station address that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its station address, the ECU must not react to CCP commands sent by the CCP master.

### Byte Order

Sets the byte order of the CCP slave device.

- **0—MSB_LAST**
  
  The CCP slave device uses the MSB_LAST (Intel) byte ordering.

- **1—MSB_FIRST**
  
  The CCP slave device uses the MSB_FIRST (Motorola) byte ordering.

### Error in

Is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

- **status** is TRUE if an error occurred. This VI will not execute when status is TRUE.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

- **source** identifies the VI where the error occurred.

### Output

**ECU ref out** is the task reference that links to the selected ECU.
Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Description

Use **MC ECU Create.vi** to create an ECU object in memory instead of referring to an ECU object defined in the A2L database. **MC ECU Create.vi** provides an alternative in which you create the ECU and DAQ List configuration within the application, without using an A2L database. **MC ECU Create.vi** creates an ECU reference handle linked to the selected ECU name. **MC ECU Create.vi** does not start communication. This enables you to use **MC Set Property.vi** to change the properties of an ECU task and to create an event channel object manually using **MC Event Create.vi**, create a measurement object using **MC Measurement Create.vi**, and create a conversion rule using **MC Conversion Create.vi**. After you change properties, use **MC ECU Connect.vi** to start communication for the task and logically connect to the selected ECU.
MC ECU Deselect.vi

**Purpose**
Deselects an ECU and invalidates the ECU reference handle.

**Format**

```
<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.</td>
</tr>
<tr>
<td>Error in</td>
<td>is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td>status</td>
<td>is TRUE if an error occurred. Unlike other VIs, this VI will execute when status is TRUE.</td>
</tr>
<tr>
<td>code</td>
<td>is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.</td>
</tr>
<tr>
<td>source</td>
<td>identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB ref out</td>
<td>is the task reference which links to the opened database file.</td>
</tr>
<tr>
<td>Error out</td>
<td>describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.</td>
</tr>
<tr>
<td>status</td>
<td>is TRUE if an error occurred.</td>
</tr>
</tbody>
</table>
```
**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.

**Description**

**MC ECU Deselect.vi** deselects the communication to the ECU. After calling this VI you can establish the communication to another ECU defined in the A2L database using **MC ECU Select.vi**.
MC ECU Disconnect.vi

Purpose
Disconnects the CCP or XCP communication to the selected ECU.

Format

\[
\begin{array}{c|c}
\text{Input} & \text{Output} \\
\hline
\text{ECU ref in} & \text{ECU ref out} \\
\text{error in} & \text{error out} \\
\end{array}
\]

Input

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

Output

- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC ECU Disconnect.vi implements the CCP or XCP command DISCONNECT. MC ECU Disconnect.vi permanently disconnects the specified CCP or XCP slave from the communication and ends the calibration session. When the calibration session is terminated, all DAQ lists of the device are stopped and cleared and the protection masks of the device are set to their default values.

MC ECU Disconnect.vi is an optional function and is automatically performed prior to any MC ECU Deselect.vi or MC ECU Close.vi call.
MC ECU Open.vi

Purpose
Opens a specified A2L database and selects the first ECU found in the database. If there are several ECUs stored in the A2L database use the Database Open and ECU Select VIs.

Format

```
protocol and interface  ECU ref out
DB path              error in
                           error out
interface
```

Input

**protocol and interface** selects target communication protocol CCP or XCP and the desired interface to use for this task. The interface input uses a string `xxx:yyy` where `xxx` defines one of the two available protocols “CCP” or “XCP” and `yyy` defines the desired interface to use like “CAN0” for CCP or XCP or “UDP” or “TCP” for XCP. The **protocol and interface** input is required as this parameter is not defined in the A2L database.

The default baud rate for CCP or XCP on CAN, or the IP address for XCP on UDP/TCP, may be defined in the A2L database, but you can change it by setting the **Interface Baud Rate** or **IP Address** property with **MC Set Property.vi**.

**NI-CAN**

The special CAN interface values 256 and 257 refer to virtual interfaces. For more information on usage of virtual interfaces, refer to the **Frame to Channel Conversion** section of Chapter 6, **Using The Channel API**, in the **NI-CAN Hardware and Software User Manual**.

**NI-XNET**

If you use NI-XNET hardware and select the `xxx:yyy` syntax, the ECU M&C Toolkit uses the XNET NI-CAN compatibility library (XCL) internally if the XNET interface is defined in MAX under **NI-CAN Devices**. To force use of the native XNET API, you must use the `xxx:yyy@nixnet` syntax. The interface name is related to the NI-XNET hardware naming under **Devices and Interfaces** in MAX.
Note  By selecting nixnet as Protocol and Interface string, the ECU Measurement and Calibration Toolkit uses the Frame Input and Output Queued sessions. To force the ECU Measurement and Calibration Toolkit to use Frame Input and Output Stream sessions instead, select ni_genie_nixnet as Protocol and Interface string (for example, CCP:CAN1@ni_genie_nixnet). An application instance can use only one Frame Input Stream Session and one Frame Output Stream Session at a time, so use the default name nixnet as Protocol and Interface string, so that multiple NI-XNET Frame Queued Sessions can coexist on a single interface, and the Frame Input and Output Stream Sessions may be used, for example, for a Frame logging/replay use case.

CompactRIO or R Series

If using CompactRIO or R Series hardware, you must provide a bitfile that handles the CAN communication between the host system and FPGA. To access the CAN module on the FPGA, you must specify the bitfile name after the @ (for example, CCP:CAN1@MyBitfile.lvbitx). To specify a special RIO target, you can specify that target by its name followed by the bitfile name (for example, XCP:CAN1@RIO1,MyBitfile.lvbitx). Currently, only a single CAN interface is supported. RIO1 defines the RIO target name as defined in your LabVIEW Project definition. The lvbitx filename represents the filename and location of the bitfile on the host. You may use just the filename without the folder if the bitfile is in the same folder as the LabVIEW Project (*.lvproj).

DB path is a path to a A2L database file from which to get channel names. The file must use a .A2L extension. You can generate A2L database files with several 3rd party tools.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

CAN interface specifies the CAN interface to use for this task. For compatibility reasons, if you are using the CCP protocol you can only
specify the CAN interface to use for this CCP task. The interface input uses a `ring` typedef in which value 0 selects CAN0, value 1 selects CAN1, and so on. As the ECU M&C API is based on the NI-CAN Channel API, the NI-CAN Frame API cannot be used on the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

If you use NI-XNET or CompactRIO/R Series hardware, use the `protocol` and `interface` parameter instead.

**Output**

- **ECU ref out** is the task reference which links to the selected ECU.
- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.
  - **status** is **TRUE** if an error occurred.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

**Description**

**MC ECU Open.vi** opens a specified A2L database and selects the first ECU found in the database. If there are several ECUs stored in the A2L database use **MC Database Open.vi** and **MC ECU Select.vi** to select a specific ECU.

Possible selections for the `interface` and `protocol` parameter for the various hardware targets are as follows.

Using CAN hardware:
- **CCP:CAN0**—uses CCP on CAN interface 0
- **CCP:CAN1**—uses CCP on CAN interface 1, and so on with the form CANx
- **CCP:CAN256**—uses CCP on virtual CAN interface 256
- **CCP:CAN257**—uses CCP on virtual CAN interface 257
- **XCP:CAN0**—uses XCP on CAN interface 0
- **XCP:CAN1**—uses XCP on CAN interface 1, and so on with the form CANx
- **XCP:UDP**—uses XCP on UDP
- **XCP:TCP**—uses XCP on TCP

Using NI-XNET hardware with NI-XNET Frame Input/Output-based sessions:
- **CCP:CAN1@nixnet**—uses CCP on CAN interface 1
- **CCP:CAN2@nixnet**—uses CCP on CAN interface 2, and so on with the form CANx
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 1
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 2, and so on with the form CANx
- **XCP:UDP**—uses XCP on UDP
- **XCP:TCP**—uses XCP on TCP

Using NI-XNET hardware with NI-XNET Stream Input/Output-based sessions:
- **CCP:CAN1@ni_genie_nixnet**—uses CCP on CAN interface 1
- **CCP:CAN2@ni_genie_nixnet**—uses CCP on CAN interface 2, and so on with the form CANx
- **XCP:CAN1@ni_genie_nixnet**—uses XCP on CAN interface 1
- **XCP:CAN1@ni_genie_nixnet**—uses XCP on CAN interface 2, and so on with the form CANx

Using CompactRIO or R Series:
- **CCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx**—uses CCP on named target RIO1 as compiled into the bitfile at c:\temp\MyFpgaBitfile.lvbitx
- **XCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx**—uses XCP on named target RIO1 as compiled into the bitfile at c:\temp\MyFpgaBitfile.lvbitx

**Note**  You can download the ASAM MCD 2MC database configuration file to a LabVIEW RT target by the File Transfer Protocol (FTP). An FTP file transfer is possible within MAX. Refer to the *LabVIEW Real-Time Graphical File Transfer Utility* section of Chapter 2, *Installation and Configuration*, for instructions on performing an FTP transfer through MAX.
MC ECU Select.vi

**Purpose**

Selects an ECU based upon the names stored in an A2L database.

**Format**

![Diagram](image)

**Input**

- **protocol and interface** selects target communication protocol CCP or XCP and the desired interface to use for this task. The interface input uses a string `xxx:yyy` where `xxx` defines one of the two available protocols “CCP” or “XCP” and `yyy` defines the desired interface to use like “CAN0” for CCP or XCP or “UDP” or “TCP” for XCP. The **protocol and interface** input is required as this parameter is not defined in the A2L database.

The default baud rate for CCP or XCP on CAN, or the IP address for XCP on UDP/TCP, may be defined in the A2L database, but you can change it by setting the **Interface Baud Rate** or **IP Address** property with **MC Set Property.vi**.

**NI-CAN**

The special CAN interface values 256 and 257 refer to virtual interfaces. For more information about using virtual interfaces, refer to the Frame to Channel Conversion section of Chapter 6, Using The Channel API, in the NI-CAN Hardware and Software User Manual.

**NI-XNET**

If you use NI-XNET hardware and select the `xxx:yyy` syntax, the ECU M&C Toolkit uses the XNET NI-CAN compatibility library (XCL) internally if the XNET interface is defined in MAX under **NI-CAN Devices**. To force use of the native XNET API, you must use the `xxx:yyy@nixnet` syntax. The interface name is related to the NI-XNET hardware naming under **Devices and Interfaces** in MAX.
Note  By selecting nixnet as **Protocol and Interface** string, the ECU Measurement and Calibration Toolkit uses the Frame Input and Output Queued sessions. To force the ECU Measurement and Calibration Toolkit to use Frame Input and Output Stream sessions instead, select ni_genie_nixnet as **Protocol and Interface** string (for example, CCP:CAN1@ni_genie_nixnet). An application instance can use only one Frame Input Stream Session and one Frame Output Stream Session at a time, so use the default name nixnet as **Protocol and Interface** string, so that multiple NI-XNET Frame Queued Sessions can coexist on a single interface, and the Frame Input and Output Stream Sessions may be used, for example, for a Frame logging/replay use case.

**CompactRIO or R Series**

If using CompactRIO or R Series hardware, you must provide a bitfile that handles the CAN communication between the host system and FPGA. To access the CAN module on the FPGA, you must specify the bitfile name after the @ (for example, CCP:CAN1@MyBitfile.lvbitx). To specify a special RIO target, you must specify that target by its name followed by the bitfile name (for example, XCP:CAN1@RIO1,MyBitfile.lvbitx). Currently, only a single CAN interface is supported. RIO1 defines the RIO target name as defined in your LabVIEW Project definition. The lvbitx filename represents the filename and location of the bitfile on the host. You may use just the filename without the folder if the bitfile is in the same folder as the LabVIEW Project (*.lvproj).

**DB reference in** is the task reference which links to the opened database file.

**ECU name** is the ECU name to select out of a A2L Database file, with which to initialize all subsequent tasks.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

**status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.
CAN interface specifies the CAN interface to use for this task. For compatibility reasons, if you are using the CCP protocol you can only specify the CAN interface to use for this CCP task. The interface input uses a `ring` typedef in which value 0 selects CAN0, value 1 selects CAN1, and so on. As the ECU M&C API is based on the NI-CAN Channel API, the NI-CAN Frame API cannot be used on the same CAN network interface simultaneously. If the CAN network interface is already initialized in the Frame API, this function returns an error.

Output

**ECU ref out** is the task reference which links to the selected ECU.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is `TRUE` if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

Description

**MC ECU Select.vi** creates an ECU reference handle linked to the selected ECU name. **MC ECU Select.vi** does not start communication. This enables you to use **MC Set Property.vi** to change the properties of an ECU task. After you change properties, use **MC ECU Connect.vi** to start communication for the task and logically connect to the selected ECU. Prior to calling **MC ECU Select.vi**, an available ECU name can be queried by calling **MC Get Property.vi** with the parameter **ECU/Name**.

Possible selections for the interface and protocol parameter for the various hardware targets are as follows.

Using NI-CAN hardware:

- **CCP:CAN0** — uses CCP on CAN interface 0
- **CCP:CAN1** — uses CCP on CAN interface 1, and so on with the form CANx
- **CCP:CAN256** — uses CCP on virtual CAN interface 256
- **CCP:CAN257** — uses CCP on virtual CAN interface 257
• XCP:CAN0—uses XCP on CAN interface 0
• XCP:CAN1—uses XCP on CAN interface 1, and so on with the form CANx
• XCP:UDP—uses XCP on UDP
• XCP:TCP—uses XCP on TCP

Using NI-XNET hardware with NI-XNET Frame Input/Output-based sessions:
• CCP:CAN1@nixnet—uses CCP on CAN interface 1
• CCP:CAN2@nixnet—uses CCP on CAN interface 2, and so on with the form CANx
• XCP:CAN1@nixnet—uses XCP on CAN interface 1
• XCP:CAN1@nixnet—uses XCP on CAN interface 2, and so on with the form CANx
• XCP:UDP—uses XCP on UDP
• XCP:TCP—uses XCP on TCP

Using NI-XNET hardware with NI-XNET Stream Input/Output-based sessions:
• CCP:CAN1@ni_genie_nixnet—uses CCP on CAN interface 1
• CCP:CAN2@ni_genie_nixnet—uses CCP on CAN interface 2, and so on with the form CANx
• XCP:CAN1@ni_genie_nixnet—uses XCP on CAN interface 1
• XCP:CAN1@ni_genie_nixnet—uses XCP on CAN interface 2, and so on with the form CANx

Using CompactRIO or R Series:
• CCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx—uses CCP on named target RIO1 as compiled into the bitfile at c:\temp\MyFpgaBitfile.lvbitx
• XCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx—uses XCP on named target RIO1 as compiled into the bitfile at c:\temp\MyFpgaBitfile.lvbitx
MC Event Create.vi

Purpose
Creates an Event object in memory.

Format

Input

- ECU ref in is the task reference that links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Create.vi.
- Event Name identifies the event channel object. Use this name as a reference in MC Measurement Create.vi to identify the event channel.
- Event Number specifies the generic signal source that effectively determines the data transmission timing. To allow a reduction of the desired transmission rate, a prescaler may be applied to the event channel. The prescaler value factor must be greater than or equal to 1 and can be set using MC Set Property.vi using the DAQ Prescaler property.
- Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.
  - status is TRUE if an error occurred. This VI will not execute when status is TRUE.
  - code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.
  - source identifies the VI where the error occurred.
Chapter 5  ECU M&C API for LabVIEW

Output

**ECU ref out** is the task reference that links to the selected ECU.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is TRUE if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

Description

Use **MC Event Create.vi** to create an Event object in memory instead of referring to a predefined measurement in the A2L database. Assign the event channel object by name to a DAQ List in **MC Measurement Create.vi**.
MC Generic.vi

Purpose
Sends a generic CCP or XCP command.

Format

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU ref in</td>
<td>The task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.</td>
</tr>
<tr>
<td>Command</td>
<td>The CCP or XCP command to be sent to the ECU.</td>
</tr>
<tr>
<td>Data</td>
<td>A 1-dimensional array of byte information to send to the ECU.</td>
</tr>
<tr>
<td>Buffer size for return value</td>
<td>Sets the maximum length of the Return value data array.</td>
</tr>
<tr>
<td>Error in</td>
<td>A cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td>status</td>
<td>TRUE if an error occurred. This VI is not executed when status is TRUE.</td>
</tr>
<tr>
<td>code</td>
<td>The error code number identifying an error.</td>
</tr>
<tr>
<td>source</td>
<td>Identifies the VI where the error occurred.</td>
</tr>
</tbody>
</table>

Timeout is the time limit, in milliseconds, during which a specified command must complete.

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Command is the CCP or XCP command to be sent to the ECU.

Data contains a 1-dimensional array of byte information to send to the ECU.

Buffer size for return value sets the maximum length of the Return value data array.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error code** describes the error returned from the ECU during the communication.

**Return value** may contain an array of bytes returned from the ECU as a response to the CCP command sent.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

**status** is **TRUE** if an error occurred.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.

Description

**MC Generic.vi** implements any generic CCP or XCP command that can be used to execute user-defined commands that are not defined in the CCP or XCP standard or not covered by the available LabVIEW CCP/XCP VIs.
MC Get Names.vi

Purpose

Gets an array of ECU names, Measurement names, Characteristic names, Event names, Calibration page names, or Group names from a specified A2L database file.

Format

```
Input
```

<table>
<thead>
<tr>
<th>Type (mode)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ECU Names returns a list of ECU names. You can write this list to MC ECU Select.vi. This is the default value.</td>
</tr>
<tr>
<td>1</td>
<td>Measurement Names returns a list of Measurement names.</td>
</tr>
<tr>
<td>2</td>
<td>Characteristic Names returns a list of Characteristic names.</td>
</tr>
<tr>
<td>3</td>
<td>Event Channel Names returns a list of Event Channel names.</td>
</tr>
<tr>
<td>4</td>
<td>Defined Pages Names returns a list of Calibration page names.</td>
</tr>
<tr>
<td>5</td>
<td>Group Names returns a list of Group names.</td>
</tr>
<tr>
<td>6</td>
<td>Group–Subgroup Names returns a list of Subgroup names of the specified Group name.</td>
</tr>
<tr>
<td>7</td>
<td>Group–Measurement Names returns a list of Measurement names within the specified Group.</td>
</tr>
<tr>
<td>8</td>
<td>Group–Characteristic Names returns a list of Characteristic names within the specified Group.</td>
</tr>
<tr>
<td>9</td>
<td>Function Names returns a list of Function names within the specified ECU.</td>
</tr>
</tbody>
</table>

The value of Type (mode) is an enumeration:

0—ECU Names returns a list of ECU names. You can write this list to MC ECU Select.vi. This is the default value.

1—Measurement Names returns a list of Measurement names.

2—Characteristic Names returns a list of Characteristic names.

3—Event Channel Names returns a list of Event Channel names.

4—Defined Pages Names returns a list of Calibration page names.

5—Group Names returns a list of Group names.

6—Group–Subgroup Names returns a list of Subgroup names of the specified Group name.

7—Group–Measurement Names returns a list of Measurement names within the specified Group.

8—Group–Characteristic Names returns a list of Characteristic names within the specified Group.

9—Function Names returns a list of Function names within the specified ECU.
10—**Function–DefCharacteristic Names** returns a list of Characteristic names referred by the DEF_CHARACTERISTIC keyword within the related Function.

11—**Function–RefCharacteristic Names** returns a list of Characteristic names referred by the REF_CHARACTERISTIC keyword within the related Function.

12—**Function–InMeasurement Names** returns a list of Measurement names referred by the IN_MEASUREMENT keyword within the related Function.

13—**Function–OutMeasurement Names** returns a list of Measurement names referred by the OUT_MEASUREMENT keyword within the related Function.

14—**Function–LocMeasurement Names** returns a list of Measurement names referred by the LOC_MEASUREMENT keyword within the related Function.

15—**Function–SubFunction Names** returns a list of Function names referred by the SUB_FUNCTION keyword within the related Function.

16—**Group–Function List Names** returns a list of Function names referred by the FUNCTION_LIST keyword within the related Group.

**Reference in** must be an ECU M&C task reference or an A2L database reference.

**ECU name** If a valid A2L database reference is passed to the reference in terminal, the ECU name terminal is used to select one of the ECUs inside the A2L database. Then, **MC Get Names.vi** will report the names of all objects of the specified type inside the ECU, based on the name provided. If you do not provide a name, the first ECU in the A2L file is selected.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

**status** is TRUE if an error occurred. This VI will not execute when **status** is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Reference out is a copy of the reference which was passed to the reference in terminal.

Names list out returns the array of names, one string entry per name. To start a Measurement task or access a Characteristic for all channels returned from MC Get Names.vi, wire channel list to MC DAQ Initialize.vi.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Get Names.vi is used to query the names contained within an A2L file.

The ECU name terminal is ignored if a valid ECU reference is connected to the reference in terminal. In that instance, MC Get Names.vi will report the names of all objects of the specified type inside the referenced ECU.

If type = 1, type = 2, or type = 3, the corresponding ECU name must be referenced in order to access ECU-specific properties.

If type = 6, type = 7, or type = 8, the corresponding Group name must be referenced in order to access the group properties.
If using MC Get Names.vi to query the list of supported event channels on an ECU, the event channels might be stored inside the ECU instead of the A2L file. To query these event channel names from the ECU directly, connect to the ECU using MC ECU Connect.vi before using MC Get Names.vi.

If using MC Get Names.vi to query the Group names and related hierarchy to build, for example, a tree user control to query these event channels, use MC Get Property.vi with the Group–Is Root? parameter.

Note For more details on how to query the Group information out of an A2L file, refer to the installed advanced example (Read A2L Group.vi) in the Example Finder.
MC Get Property.vi

Purpose

 Gets a property for the object referenced by the reference in terminal. The poly VI selection determines the property to get.

Format

Input

Name specifies an individual channel within the task defined by reference in. The default (unwired) value of name is empty, which means the property applies to the entire task, not a specific channel. If a property relates to Measurement or Characteristic channels and does not apply to the entire task, but an individual channel or message within the task, you must wire the name of a Measurement or Characteristic channel from channel list into the name input. For other properties you must leave name unwired (empty).

Reference in is the reference to any opened A2L database, a selected ECU, or an ECU which is already connected (with MC Database Open.vi, MC ECU Select.vi, MC ECU Open.vi, or MC ECU Connect.vi). The type of this reference depends on the property you want to get.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.
source identifies the VI where the error occurred.

**Output**

**Reference out** contains an ECU M&C task reference which can be wired through subsequent ECU M&C VIs.

**Value** is a poly output value that returns the property value. You select the property returned in value by selecting the poly VI type. The data type of value is also determined by the poly VI selection. For information about the different properties provided by **MC Get Property.vi**, refer to the **Poly VI Types** section. To select the property, right-click the VI, go to **Select Type**, and select the property by name.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

**status** is **TRUE** if an error occurred.

**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.
### Description

#### Poly VI Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![abc]</td>
<td>—</td>
<td>—</td>
<td>DB File Name</td>
<td>Returns the A2L database file name with which the task has been opened. The value of this property cannot be changed using <a href="#">MC Set Property.vi</a>.</td>
</tr>
<tr>
<td>![pus2]</td>
<td>ECU</td>
<td>—</td>
<td>Byte Order</td>
<td>Returns the byte order of the CCP slave device.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0 — MSB_LAST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The CCP Slave device uses the MSB_LAST (Intel) byte ordering.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 — MSB_FIRST</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The CCP Slave device uses the MSB_FIRST (Motorola) byte ordering.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output  (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>—</td>
<td>—</td>
<td>Seedkey/Checksum DLL Path</td>
<td>This property determines the directory where the ECU M&amp;C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&amp;C Toolkit expects the DLLs in the same directory as the A2L file. If your DLLs are in a different directory, set this property pointing to this directory.</td>
</tr>
<tr>
<td>ECU</td>
<td>—</td>
<td>—</td>
<td>Checksum DLL Name</td>
<td>Returns the file name of the Checksum DLL used for verifying the checksum.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td></td>
<td></td>
<td>LogFile Path</td>
<td>Returns the filename (full path) where the CCP or XCP protocol traffic is logged in ASCII format for debugging purposes. An empty path indicates no logging (default). Note that on RT and cRIO systems, the logfile is created on the target system and must be transferred to the host after logging has been completed. Note that no additional CAN port is used for the logging, which makes this method superior to any other method such as running a bus monitor parallel.</td>
</tr>
<tr>
<td>ECU</td>
<td></td>
<td></td>
<td>Command Byte Order</td>
<td>Returns the byte order for the defined Measurement or Characteristic: 0—MSB_LAST The CCP Slave device uses the MSB_LAST (Intel) byte ordering. 1—MSB_FIRST The CCP Slave device uses the MSB_FIRST (Motorola) byte ordering.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="abc" /></td>
<td>ECU</td>
<td>—</td>
<td>—</td>
<td>Name</td>
</tr>
<tr>
<td><img src="image2" alt="u16" /></td>
<td>ECU</td>
<td>—</td>
<td>—</td>
<td>DAQ List Number</td>
</tr>
<tr>
<td><img src="image3" alt="u32" /></td>
<td>ECU</td>
<td>—</td>
<td>—</td>
<td>Event Channel</td>
</tr>
<tr>
<td><img src="image4" alt="u32" /></td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Baud Rate</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>CRO ID</td>
<td>Returns the CRO ID (Command Receive Object) which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>CRO Task</td>
<td>Returns the NI-CAN task reference for the CRO (Command Receive Object, the CAN task writing frames to the slave device). For example, you might use this to set CAN properties for this task. Handle with extreme care, as those properties are usually set correctly by the ECU M&amp;C Toolkit itself.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>DTO ID</td>
<td>Returns the DTO ID (Data Transmission Object) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.</td>
</tr>
</tbody>
</table>
### ECU M&C API for LabVIEW

#### Chapter 5

**Table 5-5. Poly Values for Value Output (Continued)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![U32]</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td><strong>DTO Task</strong>&lt;br&gt; Returns the NI-CAN task reference for the DTO ID (Data Transmission Object, the CAN task reading frames from the slave device). For example, you might use this to set CAN properties for this task. Handle with extreme care, as those properties are usually set correctly by the ECU M&amp;C Toolkit itself.</td>
</tr>
<tr>
<td>![U8]</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td><strong>ID</strong>&lt;br&gt; Returns the slave device identifier. This ID information is optional and specific to the ECU implementation. For more information about the CCP slave ID, refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>![U32]</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td><strong>ID Data Byte</strong>&lt;br&gt; Returns a data type qualifier of the slave device identifier. This ID information is optional and specific to the ECU implementation. For more information about the CCP slave ID, refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>Type</td>
<td>Hierarchy</td>
<td>Sub-Hierarchy</td>
<td>Param</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>-----------</td>
<td>---------------</td>
<td>-------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>{152}</td>
<td>ECU</td>
<td>CCP</td>
<td>Interface</td>
<td>Returns the interface initialized for the task, such as with MC DAQ Initialize.vi.</td>
</tr>
<tr>
<td>{u8}</td>
<td>ECU</td>
<td>CCP</td>
<td>Master ID</td>
<td>Returns CCP Master ID information. This ID information is optional and specific to the ECU implementation. For more information about the CCP master ID, refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>abc</td>
<td>ECU</td>
<td>CCP</td>
<td>SeedKey Cal Name</td>
<td>Returns the file name of the SeedKey DLL used for Calibration purposes.</td>
</tr>
<tr>
<td>abc</td>
<td>ECU</td>
<td>CCP</td>
<td>SeedKey DAQ Name</td>
<td>Returns the file name of the SeedKey DLL used for DAQ purposes.</td>
</tr>
<tr>
<td>abc</td>
<td>ECU</td>
<td>CCP</td>
<td>SeedKey Prog Name</td>
<td>Returns the file name of the SeedKey DLL used for programming purposes.</td>
</tr>
<tr>
<td>{1f}</td>
<td>ECU</td>
<td>CCP</td>
<td>Single Byte DAQ List?</td>
<td>Determines if an ECU supports single-byte or multi-byte DAQ list entries.</td>
</tr>
</tbody>
</table>
For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN or LIN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details.

(This property is supported for NI-XNET devices only.)
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td>For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN or LIN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN or LIN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)</td>
</tr>
</tbody>
</table>
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Station Address</td>
<td>Returns the <strong>Station Address</strong> of the slave device. CCP is based on the idea, that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts CCP defines a <strong>Station Address</strong> that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its Station Address, the ECU must not react to CCP commands sent by the CCP master.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Misc</td>
<td>Skip EXCHANGE ID</td>
<td>Returns a Boolean value that indicates whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>ACTION SERVICE</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command ACTION_SERVICE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>BUILD_CHKSUM</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command BUILD_CHKSUM.</td>
</tr>
<tr>
<td>Type</td>
<td>Hierarchy</td>
<td>Sub-Hierarchy</td>
<td>Param</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>-----------</td>
<td>----------------------------</td>
<td>---------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>CLEAR MEMORY</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>CLEAR MEMORY</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command CLEAR_MEMORY.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>DIAG SERVICE</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command DIAG_SERVICE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>DNLOAD 6</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command DNLOAD_6.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>GET ACTIVE CAL PAGE</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_PAGE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>GET S STATUS</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_S_STATUS.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Sub 1</th>
<th>Sub 2</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>GET SEED</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_SEED.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>MOVE</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command MOVE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>PROGRAM</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command PROGRAM.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>PROGRAM 6</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command PROGRAM_6.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>SELECT CAL PAGE</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>SET S STATUS</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>SHORT UP</td>
<td></td>
<td></td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command SHORT_UP.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Sub 1</th>
<th>Sub 2</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>START STOP_ALL</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command START_STOP_ALL.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>TEST</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command TEST.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>UNLOCK</td>
<td>Returns a Boolean value that indicates whether the ECU supports the optional CCP Command UNLOCK.</td>
</tr>
<tr>
<td>ECU</td>
<td>Misc</td>
<td>—</td>
<td></td>
<td></td>
<td>Timing Factor</td>
<td>Returns the used timing factor, which you can use to increase CCP or XCP command timeout values. For details on the default Command Timeout values, refer to the CCP or XCP Protocol Specification.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>—</td>
<td></td>
<td></td>
<td>Compression Method</td>
<td>Returns the selected compression method used for MC Program.vi.</td>
</tr>
<tr>
<td>U8</td>
<td>XCP</td>
<td>—</td>
<td></td>
<td></td>
<td>Compression Method</td>
<td>0—data is uncompressed.</td>
</tr>
<tr>
<td>U8</td>
<td>XCP</td>
<td>—</td>
<td></td>
<td></td>
<td>Compression Method</td>
<td>0x80...0xFF—User defined.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
</table>
| ECU  | XCP       |               | Encryption Method | Returns the selected encryption method used for MC Program.vi.  
0x00—data is not encrypted  
0x80...0xFF—User defined |
| ECU  | XCP       |               | Access Method | Returns the selected access mode:  
0x00—**Absolute Access Mode** (default). The MTA uses physical addresses  
0x01—**Functional Access Mode**. The MTA functions as a block sequence number of the new flash content file.  
0x80...0xFF—**User defined**. It is possible to use different access modes for clearing and programming. |
| ECU  | XCP       |               | Programming Method | Returns the selected programming method used for MC Program.vi.  
0x00—Sequential programming,  
0x80...0xFF—User defined. |
| ECU  | XCP       |               | SeedKey DLL | Returns the file name of the SeedKey DLL. |
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Sub 1</th>
<th>Sub 2</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[P32]</td>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td></td>
<td>Baudrate</td>
<td>Returns the Baud Rate in use by the NI-CAN Interface. Basic baud rates such as 125000 and 500000 are specified as the numeric rate. Advanced baud rates are specified as 8000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1) of the CAN controller chip. For more information, refer to the <strong>Interface Properties</strong> dialog in MAX. The value of this property is originally set within MAX, but it can be changed using <strong>MC Set Property.vi</strong>.</td>
</tr>
<tr>
<td>[P32]</td>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td></td>
<td>CRO Id</td>
<td>Returns the CRO ID (Command Receive Object) which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>[P32]</td>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td></td>
<td>DTO Id</td>
<td>Returns the DTO ID (Data Transmission Object) which is used by the ECU to respond to XCP commands and send data and status information to the XCP master.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>IP Address</td>
<td>Returns the IP address of the slave device. A slave device connected by Ethernet and TCP/IP or UDP/IP protocol is addressed by its IP Address and Port number.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>IP Port</td>
<td>Returns the IP Port number of the slave device. A slave device connected by Ethernet and TCP/IP or UDP/IP protocol is addressed by its IP Address and Port number.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>Timeout</td>
<td>Returns one of the seven timeout values (in milliseconds) defined in the XCP standard for the various XCP commands. For details of which timeout applies to a specific command, refer to the XCP standard. The values typically are read from an A2L file but may be overridden manually. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
</tbody>
</table>

Table 5-5. Poly Values for Value Output (Continued)
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_std</td>
<td>Returns the timeout value (in milliseconds) for most of the CCP commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default: 40. Standard: 25. The default is chosen slightly higher to allow for slower ECUs. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_pgm</td>
<td>Returns the timeout value (in milliseconds) for the CCP programming commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default: 120. Standard: 100. The default is chosen slightly higher to allow for slower ECUs. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
<tr>
<td>Type</td>
<td>Hierarchy</td>
<td>Sub-Hierarchy</td>
<td>Param</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------</td>
<td>---------------</td>
<td>-------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_mem</td>
<td>Returns the timeout value (in milliseconds) for the CCP memory commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default and Standard: 30000. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_diag</td>
<td>Returns the timeout value (in milliseconds) for the CCP DIAG_SERVICE command. Default and Standard: 500. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_act</td>
<td>Returns the timeout value (in milliseconds) for the CCP ACTION_SERVICE command. Default: 500. Standard: 5000. Note that the <strong>Timing Factor</strong> property might modify this value.</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>——</td>
<td>——</td>
<td>Address</td>
</tr>
</tbody>
</table>

*Table 5-5. Poly Values for Value Output (Continued)*
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Characteristic</td>
<td>---</td>
<td>--</td>
<td><strong>Byte Order</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>Returns the specified byte order:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0—<strong>Intel format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bytes are in little-endian order, with least-significant bit first.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1—<strong>Motorola format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Bytes are in big-endian order, with most-significant bit first.</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>---</td>
<td>--</td>
<td><strong>Comment</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the Comment string of the selected Characteristic.</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>---</td>
<td>--</td>
<td><strong>Data Type</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the data type of the Characteristic.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Dimension</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Extension</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Maximum</td>
</tr>
<tr>
<td></td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Minimum</td>
</tr>
</tbody>
</table>
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Character</td>
<td>—</td>
<td>Read Only?</td>
<td>Returns if a Characteristic is set to Read Only. In this case it is not allowed to call <code>MC Characteristic Write.vi</code> for this Characteristic.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>—</td>
<td>Sizes</td>
<td>Returns the array <code>Sizes</code> for X and Y direction of the Characteristic.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>—</td>
<td>Unit</td>
<td>Returns the unit string defined for this Characteristic in the A2L database.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>—</td>
<td>X Axis</td>
<td>Returns X-axis values on which the Characteristic is defined. It is valid if the dimension of the selected Characteristic is 1 or 2.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>—</td>
<td>Y Axis</td>
<td>Returns Y-axis values on which the Characteristic is defined. It is valid if the dimension of the selected Characteristic is 2.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>Scaling</td>
<td>Factor</td>
<td>Returns the scaling factor defined for this Characteristic in the A2L database.</td>
</tr>
<tr>
<td></td>
<td>Character</td>
<td>Scaling</td>
<td>Offset</td>
<td>Returns the scaling offset defined for this Characteristic in the A2L database.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic</td>
<td>Scaling</td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>Type</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0: Unknown</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1: Rational Function</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2: Enumeration Text</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Sub 1</th>
<th>Sub 2</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
</table>

- It is possible to use **MC Double to Text.vi** and **MC Text To Double.vi** to convert between enumeration text values and double values.

3: Range Text

The related scaling is based on the **COMPU_VTAB_RANG**GE keyword within the A2L file.

- Read VIs return nonscaled, numeric values.
- Write VIs accept nonscaled, numeric values.
- It is possible to use **MC Double to Text.vi** and **MC Text To Double.vi** to convert between range text values and double values.

4: Formula

The related scaling is based on the **FORMULA** keyword within the A2L file, using a free formula to calculate the values.
The related scaling is based on the TAB_INTP keyword within the A2L file, using interpolation between x-y pairs.

### 6: Table (Without Interpolation)

The related scaling is based on the TAB_NOINTP keyword within the A2L file, using x-y pairs without interpolation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Characteristic</td>
<td>Scaling</td>
<td>—</td>
<td>Text Values</td>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the list of text values that can be converted into raw values.</td>
</tr>
<tr>
<td>Group</td>
<td>—</td>
<td>—</td>
<td>Is Root?</td>
<td>Returns whether the selected Group is a root-level Group entity.</td>
</tr>
<tr>
<td>DAQ</td>
<td>—</td>
<td>—</td>
<td>Event Channel Name</td>
<td>Returns the selected event channel name to which the Measurement task is assigned.</td>
</tr>
</tbody>
</table>
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
</table>
| DAQ  | —         | —             | Mode  | Returns the selected I/O mode for the M&C Measurement task. 0—DAQ List  
The data is transmitted by the ECU based on an event channel, which can be equidistant in time or sporadic. The data can be read back with the **MC DAQ Read.vi** as Single point data using sample rate = 0, or as a waveform using a sample rate > 0. Input channel data is received from the DAQ messages. Use **MC DAQ Read.vi** to obtain input samples as single-point, array, or waveform. 1—Polling  
In this mode the data from the Measurement task is uploaded from the ECU whenever **MC DAQ Read.vi** is called. |
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAQ</td>
<td>Sub 1 Sub 2</td>
<td># Channels</td>
<td>Returns the number of channels initialized in a DAQ channel list of a M&amp;C Measurement task. This is the number of array entries required when using MC DAQ Read.vi.</td>
</tr>
<tr>
<td>U32</td>
<td>DAQ</td>
<td>Sub 1 Sub 2</td>
<td>Prescaler</td>
<td>Returns the prescaling factor which is used to reduce the desired transmission frequency of the associated DAQ list.</td>
</tr>
<tr>
<td>U16</td>
<td>DAQ</td>
<td>Sub 1 Sub 2</td>
<td>Sample Rate</td>
<td>Returns the selected Sample Rate in Hz for the M&amp;C Measurement task, which may be obtained with MC DAQ Initialize.vi.</td>
</tr>
</tbody>
</table>
## Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAQ</td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>Samples Pending</td>
</tr>
<tr>
<td></td>
<td>DAQ</td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>Time Since Last Frame</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAQ</td>
<td>CCP</td>
<td>—</td>
<td>DTO ID</td>
<td>Returns the DTO ID (Data Transmission Object) which is used by the ECU to send DAQ list data to the CCP master.</td>
</tr>
<tr>
<td>DAQ</td>
<td>CCP</td>
<td>—</td>
<td>DTO Task</td>
<td>Returns the NI-CAN task reference for the DTO ID (Data Transmission Object, the CAN task reading frames from the slave device). For example, you might use this to set CAN properties for this task. Handle with extreme care, as those properties are usually set correctly by the ECU M&amp;C Toolkit itself.</td>
</tr>
<tr>
<td>DAQ List</td>
<td>—</td>
<td>—</td>
<td>CAN ID</td>
<td>Returns the CAN ID for the specified DAQ list if mcPropDAQList_CANIdSelectMode == CAN_ID_FIXED.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output  (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAQ List</td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>CAN ID Select Mode</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the condition for selecting the CAN ID for the specified DAQ list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0—CAN_ID_FIXED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The CAN Identifier is a predefined fixed number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1—CAN_ID_VARIABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The CAN Identifier is a variable number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2—CAN_ID_DTO_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The CAN Identifier is the same as the DTO identifier.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Excluded DAQ Lists</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns an array containing the numbers of DAQ lists not working together with the current DAQ list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>First PID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the first Packet ID for the specified DAQ list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MAX Length</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the maximal length of the DAQ list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Reduction Allowed</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns whether or not the specified DAQ list allows reduction.</td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAQ List</td>
<td>—</td>
<td>Name</td>
<td>Name of the DAQ list (measurement source). Pass the DAQ list number converted to a string in the Name parameter of Get Property. The available DAQ list number can be obtained by the ECU DAQ List Numbers property.</td>
</tr>
<tr>
<td></td>
<td>DAQ List</td>
<td>—</td>
<td>Display Identifier</td>
<td>Optional property you can use as a display name as an alternative to the DAQ List Name property.</td>
</tr>
<tr>
<td>Measurement</td>
<td>—</td>
<td>—</td>
<td>Address</td>
<td>Returns the address part of the address of the selected Measurement in the memory of the control unit.</td>
</tr>
</tbody>
</table>
| Measurement | —     | —             | Byte Order | Returns the specified byte order:  
0—Intel format  
Bytes are in little-endian order, with least-significant bit first.  
1—Motorola format  
Bytes are in big-endian order, with most-significant bit first. |
| Measurement | —     | —             | Comment | Returns the Comment string of the selected Measurement. |
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![U8]</td>
<td>Measurement</td>
<td></td>
<td>Data Type</td>
<td>Returns the data type of the Measurement task.</td>
</tr>
<tr>
<td>![U8]</td>
<td>Measurement</td>
<td></td>
<td>Extension</td>
<td>Returns the extension part of the address. This optional parameter may contain additional address information defined in the A2L database. For instance, it can be used to distinguish different address spaces of an ECU (multi-microcontroller devices).</td>
</tr>
<tr>
<td>![TF]</td>
<td>Measurement</td>
<td></td>
<td>Is Virtual?</td>
<td>Indicates whether the Measurement is virtual. Virtual Measurements are not transmitted by the ECU but are calculated in the application. They return an error when opened in a DAQ list.</td>
</tr>
<tr>
<td>![DBL]</td>
<td>Measurement</td>
<td></td>
<td>Maximum</td>
<td>Returns the maximum value of the Measurement.</td>
</tr>
<tr>
<td>![DBL]</td>
<td>Measurement</td>
<td></td>
<td>Minimum</td>
<td>Returns the minimum value of the Measurement.</td>
</tr>
</tbody>
</table>
**Table 5-5. Poly Values for Value Output (Continued)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![TF]</td>
<td>Measurement</td>
<td>—</td>
<td>—</td>
<td><strong>Read Only?</strong></td>
</tr>
<tr>
<td>![abc]</td>
<td>Measurement</td>
<td>—</td>
<td>—</td>
<td><strong>Unit</strong></td>
</tr>
<tr>
<td>![DBL]</td>
<td>Measurement</td>
<td>Scaling</td>
<td>—</td>
<td><strong>Factor</strong></td>
</tr>
<tr>
<td>![DBL]</td>
<td>Measurement</td>
<td>Scaling</td>
<td>—</td>
<td><strong>Offset</strong></td>
</tr>
</tbody>
</table>
### Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measurement</td>
<td>Scaling</td>
<td></td>
<td>Type</td>
<td>Returns the scaling type defined for this Measurement in the A2L database.</td>
</tr>
</tbody>
</table>

**0: Unknown**

The type of the scaling could not be derived from the A2L file content.

**1: Rational Function**

The related scaling is based on a rational function of second order. This also covers the linear scaling, given by factor and offset.

**2: Enumeration Text**

The related scaling is based on the COMPU_VTAB keyword within the A2L file.

- Read VIs return nonscaled, numeric values.
- Write VIs accept nonscaled, numeric values.
### 3: Range Text

The related scaling is based on the `COMPU_VTAB_RANGEO` keyword within the A2L file.

- Read VIs return nonscaled, numeric values.
- Write VIs accept nonscaled, numeric values.
- It is possible to use `MC Double to Text.vi` and `MC Text To Double.vi` to convert between range text values and double values.

### 4: Formula

The related scaling is based on the `FORMULA` keyword within the A2L file, using a free formula to calculate the values.

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>It is possible to use <code>MC Double to Text.vi</code> and <code>MC Text To Double.vi</code> to convert between enumeration text values and double values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3: Range Text</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The related scaling is based on the <code>COMPU_VTAB_RANGE</code> keyword within the A2L file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Read VIs return nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- Write VIs accept nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>- It is possible to use <code>MC Double to Text.vi</code> and <code>MC Text To Double.vi</code> to convert between range text values and double values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4: Formula</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>The related scaling is based on the <code>FORMULA</code> keyword within the A2L file, using a free formula to calculate the values.</td>
</tr>
</tbody>
</table>
**Table 5-5. Poly Values for Value Output (Continued)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sub 1</td>
<td>Sub 2</td>
</tr>
<tr>
<td>5: Table (Using Interpolation)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>The related scaling is based on the TAB_INTP keyword within the A2L file, using interpolation between x-y pairs.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6: Table (Without Interpolation) | | | | | |
| The related scaling is based on the TAB_NOINTP keyword within the A2L file, using x-y pairs without interpolation. |

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Scaling</th>
<th>—</th>
<th>Text Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the list of text values that can be converted into raw values.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Version</th>
<th>—</th>
<th>—</th>
<th>Build</th>
</tr>
</thead>
<tbody>
<tr>
<td>Returns the build number of the ECU M&amp;C software. This number applies to the Development, Alpha, and Beta phases only, and should be ignored for the Release phase.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 5-5. Poly Values for Value Output (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Version</td>
<td>Sub 1</td>
<td>Sub 2</td>
<td>Comment</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns a comment string for the ECU M&amp;C software. If you received a custom release of ECU M&amp;C from National Instruments, this comment often describes special features of the release.</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td></td>
<td></td>
<td>Major</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the major version of the ECU M&amp;C software, such as the 1 in version 1.2.5.</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td></td>
<td></td>
<td>Minor</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the minor version of the ECU M&amp;C software, such as the 2 in version 1.2.5.</td>
</tr>
<tr>
<td></td>
<td>Version</td>
<td></td>
<td></td>
<td>Update</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Returns the update version of the ECU M&amp;C software, such as the 5 in version 1.1.5.</td>
</tr>
</tbody>
</table>
MC Measurement Create.vi

Purpose
Creates a Measurement object in memory.

Format

Conversion Name identifies the referred conversion object defined by MC Conversion Create.vi.

Measurement Name sets the measurement object name.

ECU_ref in is the task reference that links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Create.vi.

Address is a cluster that contains the following values:

   Address specifies the address part of the source address.

   Extension contains the extension part of the source address.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
**Data Type** sets the measurement task data type. **Data Type** can contain the following values:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unsigned byte</td>
</tr>
<tr>
<td>1</td>
<td>Signed byte</td>
</tr>
<tr>
<td>2</td>
<td>Unsigned word</td>
</tr>
<tr>
<td>3</td>
<td>Signed word</td>
</tr>
<tr>
<td>4</td>
<td>Unsigned long</td>
</tr>
<tr>
<td>5</td>
<td>Signed long</td>
</tr>
<tr>
<td>6</td>
<td>Float 32</td>
</tr>
</tbody>
</table>

**Output**

**ECU ref out** is the task reference that links to the selected ECU.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **true** if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

**Description**

Use **MC Measurement Create.vi** to create a measurement object in memory instead of referring to a predefined measurement in the A2L database.
**MC Measurement Read.vi**

**Purpose**

Reads a single Measurement value from the ECU.

**Format**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>measurement name</strong></td>
<td><strong>ECU ref out</strong></td>
</tr>
<tr>
<td><strong>ECU ref in</strong></td>
<td><strong>Value</strong></td>
</tr>
<tr>
<td><strong>error in</strong></td>
<td><strong>error out</strong></td>
</tr>
</tbody>
</table>

- **Measurement name** is the name of a measurement channel stored in the A2L database file you want to read.
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.
- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.
- **Value** returns a single sample for the Measurement channel initialized in **measurement name**.
**Description**

**MC Measurement Read.vi** performs a single point read of a single Measurement from the selected ECU without opening a Measurement task.
MC Measurement Write.vi

Purpose

Writes a single Measurement value to the ECU.

Format

<table>
<thead>
<tr>
<th>measurement name</th>
<th>ECU ref in</th>
<th>value</th>
<th>status</th>
<th>code</th>
<th>source</th>
</tr>
</thead>
</table>

Input

- **measurement name** is the name of a Measurement channel stored in the A2L database file to which to write a Measurement value.
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **Value** writes a single sample for the Measurement channel initialized in **measurement name**.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
- **status** is TRUE if an error occurred. This VI will not execute when **status** is TRUE.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.
Chapter 5  ECU M&C API for LabVIEW

Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Measurement Write.vi performs a single point write of a Measurement into the selected ECU without opening a Measurement task. MC Measurement Write.vi can only be performed if the Measurement is not set to read only. To query if an ECU Measurement channel can be accessed by MC Measurement Write.vi, first call MC Get Property.vi with the parameter Measurement/Read Only?.
MC Program.vi

Purpose
Programs a memory block on the ECU.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>Address</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data</td>
<td>ECU ref out</td>
</tr>
<tr>
<td></td>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Address** is a cluster which contains the following values.

- **Address** specifies the address part of the destination address.
- **Extension** contains the extension part of the destination address.

**Data** contains the byte array to be transmitted to the ECU.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI is not executed when **status** is **TRUE**.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.
Output

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

Description

If you are using the CCP protocol, **MC Program.vi** implements the CCP command PROGRAM. The command is used to program the specified data into nonvolatile ECU memory (Flash, EEPROM, etc.). Programming starts at the selected MTA0 address and extension defined in the **Address** cluster.

If you are using the XCP protocol, **MC Program.vi** implements the XCP command PROGRAM. The command is used to program a non-volatile memory segment in the ECU slave. The end of the programming sequence is indicated by using the **MC Program Reset.vi** command which executes the XCP command PROGRAM_RESET. The slave device will move into a disconnected state. Usually a hardware reset of the slave device is executed. This command may support block transfer similar to the commands DOWNLOAD and DOWNLOAD_NEXT.

For further information on how to use the **MC Program.vi** and details on block mode transfers, refer to the ASAM **XCP Part 2 Protocol Layer Specification**.
MC Program Reset.vi

Purpose
Indicates the end of a programming sequence.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>

Input
ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output
ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

If you are using the XCP protocol, MC Program Reset.vi implements the XCP command PROGRAM_RESET. This optional command indicates the end of a non-volatile memory programming sequence and may or may not have a response from the ECU. In either case, the slave device will go into a disconnected state.

MC Program Reset.vi may be used to reset a slave device for other purposes. For further information on how to use program ECU memory and to use the MC Program Reset.vi command refer to the ASAM XCP Part 2 Protocol Layer Specification.
MC Program Start.vi

Purpose
Indicates the start of a programming sequence.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

  - **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.

  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

  - **source** identifies the VI where the error occurred.

Output

- **ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

  - **status** is TRUE if an error occurred.
**code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

**source** identifies the VI where the error occurred.

**Description**

If you are using the XCP protocol, **MC Program Start.vi** implements the XCP command PROGRAM_START. This optional command indicates the beginning of a programming sequence into a non-volatile memory area. If the slave device is not in a state which permits programming, an error is returned. The memory programming commands The end of a non-volatile memory programming sequence is indicated by using the **MC Program Start.vi** function.

For further information on how to use program ECU memory and to use the **MC Program Start.vi** command refer to the ASAM XCP Part 2 Protocol Layer Specification.
MC Set Property.vi

Purpose
Sets a property for the specified A2L database file, Measurement Task or Characteristic referenced by the reference in terminal. The poly VI selection determines the property to set.

Format

<table>
<thead>
<tr>
<th>Name</th>
<th>Reference in</th>
<th>Value</th>
<th>Reference out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Value</th>
<th>Reference in</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input

Name is not used, and can be left unwired. This parameter may be used for further extensions.

Reference in specifies a valid task handle depending on the information which must be set. If a generic property must be set, a DB ref handle is needed. If a Measurement property must be set, a valid DAQ ref handle must be wired into reference in. If an ECU property must be set, a valid ECU ref handle must be wired into reference in.

Value is a poly input that specifies the property value. You select the property to set as value by selecting the poly VI type. The data type of value is also determined by the poly VI selection. For information on the different properties provided by MC Set Property.vi, refer to the Poly VI Types section. To select the property, right-click the VI, go to Select Type and select the property by name.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Reference out is a copy of the reference in terminal which can be wired through subsequent ECU M&C VIs.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

There are four types of properties which can be modified in the poly input value: ECU-specific properties, DAQ-specific properties, Characteristic-specific properties, and Measurement-specific properties.

ECU-Specific Properties

ECU-specific properties relate to the setting of the ECU. If you need to change a property of the ECU you need a valid ECU reference, but the ECU should not be connected. First, call MC ECU Open.vi, followed by MC Set Property.vi and then MC ECU Connect.vi. If you have already connected to the ECU, you can change an ECU property by calling MC ECU Disconnect.vi, followed by MC Set Property.vi, and then MC ECU Connect.vi again. Refer to Table 5-6 for a list of ECU-specific properties that can be used to define the poly input value.
DAQ-Specific Properties

You cannot set a property while the task is running. If you need to change a property prior to starting the task, call `MC DAQ Initialize.vi`, followed by `MC Set Property.vi` and then `MC DAQ Start Stop.vi`. After you start the task, you also can change a property by calling `MC DAQ Start Stop.vi`, followed by `MC Set Property.vi`, and then restart the task with `MC DAQ Start Stop.vi`. Refer to Table 5-7 for a list of DAQ-specific properties that can be used to define the poly input value.

Poly VI Types

**Table 5-6.** ECU-Specific Property Value Types for the POLY Input Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
</table>
| ECU  | —         | —             | Byte Order | Sets the byte order of the CCP slave device.  
|      | —         | —             |       | $0$—MSB-LAST  
|      | —         | —             |       | The CCP slave device uses the MSB_LAST (Intel) byte ordering.  
|      | —         | —             |       | $1$—MSB-FIRST  
|      | —         | —             |       | The CCP slave device uses the MSB_FIRST (Motorola) byte ordering.  
| ECU  | —         | —             | Command Byte Order | Sets the byte order of the CCP or XCP commands.  
|      | —         | —             |       | $0$—MSB-LAST  
|      | —         | —             |       | The CCP slave device uses the MSB_LAST (Intel) byte ordering.  
|      | —         | —             |       | $1$—MSB-FIRST  
|      | —         | —             |       | The CCP slave device uses the MSB_FIRST (Motorola) byte ordering.  |
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>---</td>
<td>---</td>
<td>Seedkey/Checksum DLL Path</td>
<td>Determines the directory where the ECU M&amp;C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&amp;C Toolkit expects the DLLs in the same directory as the A2L file. If your DLLs are in a different directory, set this property pointing to this directory.</td>
</tr>
<tr>
<td>ECU</td>
<td>---</td>
<td>---</td>
<td>Checksum DLL Name</td>
<td>Sets the file name of the Checksum DLL used for verifying the checksum.</td>
</tr>
<tr>
<td>ECU</td>
<td>---</td>
<td>---</td>
<td>Logfile Path</td>
<td>Sets a filename (full path) where the CCP or XCP protocol traffic is logged in ASCII format for debugging purposes. Setting this parameter to an empty path disables logging (default). Note that on RT and cRIO systems, the logfile is created on the target system and must be transferred to the host after logging has been completed. Note that no additional CAN port is used for the logging, which makes this method superior to any other method such as running a bus monitor parallel.</td>
</tr>
</tbody>
</table>
### Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U32</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Baud Rate</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sets the <strong>Baud Rate</strong> in use by the interface. This property applies to all tasks initialized with the Interface. You can specify the following basic baud rates as the numeric rate: 33333, 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000. You also can specify advanced baud rates in the form 8000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1).</td>
</tr>
<tr>
<td>U32</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>CRO ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sets the <strong>CRO ID</strong> (<strong>Command Receive Object</strong>) which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>U32</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>DTO ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sets the <strong>DTO ID</strong>, which is the CAN identifier for the <strong>Data Transmission Object</strong> (DTO). The DTO is used by the CCP slave devices to return data and status information to the application.</td>
</tr>
<tr>
<td>U8</td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Master ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Sets the CAN identifier of the CCP master that is used by the CCP command EXCHANGE_ID as a parameter.</td>
</tr>
</tbody>
</table>
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>SeedKey Cal Name</td>
<td>Sets the file name of the SeedKey DLL used for Calibration purposes.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>SeedKey DAQ Name</td>
<td>Sets the file name of the SeedKey DLL used for DAQ purposes.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>SeedKey Prog Name</td>
<td>Sets the file name of the SeedKey DLL used for programming purposes.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Single Byte DAQ List?</td>
<td>Sets the ECU to support single-byte or multi-byte DAQ list entries.</td>
</tr>
</tbody>
</table>
For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN or LIN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)

**Table 5-6.** ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Termination</td>
<td>For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN or LIN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)</td>
</tr>
</tbody>
</table>
For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN or LIN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN or LIN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>—</td>
<td>Station Address &lt;br&gt; Sets the Station Address of the slave device. CCP is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a Station Address that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its Station Address, the ECU must not react to CCP commands sent by the CCP master.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Misc</td>
<td>Skip EXCHANGE ID &lt;br&gt; Sets whether or not the CCP command EXCHANGE_ID should be suppressed during connection to the ECU.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>ACTION SERVICE &lt;br&gt; Sets whether the ECU supports the optional ASAM CCP Command ACTION_SERVICE.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>BUILD CHECKSUM &lt;br&gt; Sets whether the ECU supports the optional ASAM CCP Command BUILD_CHKSUM.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>CLEAR MEMORY &lt;br&gt; Sets whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.</td>
</tr>
</tbody>
</table>
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command DIAG_SERVICE.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command DNLOAD_6.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command GET_ACTIVE_CAL_PAGE.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command GET_S_STATUS.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command GET_SEED.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command MOVE.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command PROGRAM.</td>
</tr>
</tbody>
</table>
### Table 5-6. ECU-Specific Property Value Types for the POLY Input Value  (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Sub 1</th>
<th>Sub 2</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>PROGRAM 6</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command PROGRAM_6.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>SELECT CAL PAGE</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command SELECT_CAL_PAGE.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>SET_S STATUS</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command SET_S_STATUS.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>SHORT UP</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command SHORT_UP.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>START_STOP_ALL</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command START_STOP_ALL.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>TEST</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command TEST.</td>
</tr>
<tr>
<td>ECU</td>
<td>CCP</td>
<td>Optional Commands</td>
<td></td>
<td></td>
<td>UNLOCK</td>
<td>Sets whether the ECU supports the optional ASAM CCP Command UNLOCK.</td>
</tr>
<tr>
<td>ECU</td>
<td>Misc</td>
<td>—</td>
<td></td>
<td></td>
<td>Timing Factor</td>
<td>Sets the timing factor to increase the XCP or CCP Command timeouts by this value. For details on the default Command Timeout values, refer to the CCP or XCP Protocol Specification.</td>
</tr>
</tbody>
</table>
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECU</td>
<td>XCP</td>
<td>SeedKey DLL</td>
<td>Sets the file name of the XCP SeedKey DLL.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>XCP</td>
<td>Access Method</td>
<td>Sets the selected access mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x00—Absolute Access Mode</strong> (default). The MTA uses physical addresses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x01—Functional Access Mode</strong>. The MTA functions as a block sequence number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>of the new flash content file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x80...0xFF—User defined.</strong> It is possible to use different access modes for</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>clearing and programming.</td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>XCP</td>
<td>Compression Method</td>
<td>Sets the selected compression method used for <strong>MC Program.vi</strong>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0</strong>—data is uncompressed.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x80...0xFF—User defined.</strong></td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>XCP</td>
<td>Encryption Method</td>
<td>Sets the selected encryption method used for <strong>MC Program.vi</strong>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x00—data is not encrypted</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x80...0xFF—User defined</strong></td>
</tr>
<tr>
<td></td>
<td>ECU</td>
<td>XCP</td>
<td>Programming Method</td>
<td>Sets the selected programming method used for <strong>MC Program.vi</strong>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x00—Sequential programming.</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>0x80...0xFF—User defined.</strong></td>
</tr>
</tbody>
</table>
Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td>Baudrate</td>
<td>Sets the Baud Rate in use by the NI-CAN Interface. Basic baud rates such as 125000 and 500000 are specified as the numeric rate. Advanced baud rates are specified as 80000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1) of the CAN controller chip.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td>CRO Id</td>
<td>Sets the CRO ID (Command Receive Object) which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>CAN</td>
<td>DTO Id</td>
<td>Sets the DTO ID (Data Transmission Object) which is used by the ECU to respond to XCP commands and send data and status information to the XCP master.</td>
</tr>
<tr>
<td>ECU</td>
<td>XCP</td>
<td>Ethernet</td>
<td>IP Address</td>
<td>Sets the IP address of the slave device. A slave device connected by Ethernet and TCP/IP or UDP/IP protocol is addressed by its IP Address and Port number.</td>
</tr>
</tbody>
</table>
### Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="U16" /></td>
<td>ECU</td>
<td>XCP</td>
<td>IP Port</td>
<td>Sets the IP Port number of the slave device. A slave device connected by Ethernet and TCP/IP or UDP/IP protocol is addressed by its IP Address and Port number.</td>
</tr>
<tr>
<td><img src="image" alt="U32" /></td>
<td>ECU</td>
<td>XCP</td>
<td>Timeout</td>
<td>T1, T2, T3, T4, T5, T6, T7</td>
</tr>
<tr>
<td><img src="image" alt="U32" /></td>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_std</td>
</tr>
</tbody>
</table>
### Table 5-6. ECU-Specific Property Value Types for the POLY Input Value  (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="U32" /></td>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_pgm</td>
</tr>
<tr>
<td><img src="image" alt="U32" /></td>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td>T_mem</td>
</tr>
</tbody>
</table>
### Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>U32t</td>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td><strong>T_diag</strong></td>
</tr>
<tr>
<td>U32t</td>
<td>ECU</td>
<td>CCP</td>
<td>Timeout</td>
<td><strong>T_act</strong></td>
</tr>
</tbody>
</table>
Table 5-7. DAQ-Specific Property Value Types for the POLY Input Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Sub-Hierarchy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Event Channel Name</td>
<td>Sets the event channel name to which the Measurement task is assigned. If there is no event channel name defined in the A2L file, you can set the Event Channel Number manually by passing a decimal number as a string.</td>
</tr>
<tr>
<td></td>
<td>Mode</td>
<td>Sets the selected I/O mode for the M&amp;C Measurement task.</td>
</tr>
<tr>
<td></td>
<td>Prescaler</td>
<td>Sets the prescaling factor, which reduces the desired transmission frequency of the associated DAQ list.</td>
</tr>
<tr>
<td></td>
<td>Sub 1</td>
<td>Sub 2</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
**Table 5-7.** DAQ-Specific Property Value Types for the POLY Input Value (Continued)

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>![DBL]</td>
<td>DAQ</td>
<td></td>
<td>Sample Rate</td>
<td>SampleRate specifies the timing to use for the samples of the (NI-CAN) task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately. For a DAQMode of mcDAQModeDAQList, SampleRate of zero means that MC DAQ Read.vi returns a single sample from the most recent messages received, and greater than zero means that MC DAQ Read.vi returns samples timed at the specified rate. For DAQMode of mcDAQModePolling, SampleRate is ignored.</td>
</tr>
<tr>
<td>![U32]</td>
<td>DAQ</td>
<td>CCP</td>
<td>DTO ID</td>
<td>Sets the DTO ID (Data Transmission Object) which is used by the ECU to send DAQ list data to the CCP master.</td>
</tr>
</tbody>
</table>
### Characteristic-Specific Properties

Table 5-8. Characteristic-Specific Property Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DBL]</td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>X Axis</td>
</tr>
<tr>
<td>[DBL]</td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Y Axis</td>
</tr>
<tr>
<td>[U32]</td>
<td>Characteristic</td>
<td>—</td>
<td>—</td>
<td>Byte Order</td>
</tr>
</tbody>
</table>

### Measurement-Specific Properties

Table 5-9. Measurement-Specific Property Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Hierarchy</th>
<th>Sub-Hierarchy</th>
<th>Param</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U32]</td>
<td>Measurement</td>
<td>—</td>
<td>—</td>
<td>Byte Order</td>
</tr>
</tbody>
</table>
MC Text To Double.vi

Purpose
Converts a text value into the numeric representation using an enumeration or range text type scaling.

Format

Input
- **object name** indicates the object (measurement or characteristic) for which the enumeration or range text scaling is performed.
- **ECU ref in** is the task reference that links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi and then wired through subsequent VIs.
- **object type** is a U32 ring that indicates the type of the object named in **object name**. Valid values are:
  1 Measurement Name
  2 Characteristic Name
- **TextValue** is the text value you want to convert. Use MC Get Property.vi (Scaling—Text Values) to request the available values.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is TRUE if an error occurred. This VI will not execute when **status** is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

value returns the numeric value to be transferred to the ECU in subsequent Write requests.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Text To Double.vi performs the conversion from the text input value into a double value.

You can use this especially if the measurement or characteristic has an associated enumeration or range text type scaling before writing the double values to the ECU, using the regular Write VIs (MC DAQ Write.vi, MC Measurement Write.vi, MC Characteristic Write.vi).
Chapter 5  ECU M&C API for LabVIEW

MC Upload.vi

Purpose

Uploads data from an ECU.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>address</th>
<th>ECU ref out</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>input</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Input

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Address is a cluster which contains the following values.

Address specifies the address part of the source address in the ECU from which the memory block is copied.

Extension specifies the extension part of the source address.

Block size is the size of the data block, in bytes, to be uploaded.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI is not executed when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Data is a byte array which receives the uploaded data from the ECU.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC Upload.vi implements the UPLOAD command. A data block of the specified length, starting at the specified address, is uploaded from the ECU. MC Upload.vi will set the Memory Transfer Address pointer MTA0 to the appropriate value as defined in the Address cluster.

If you are using the CCP protocol, MC Upload.vi implements the UPLOAD command. A data block of the specified length, starting at the specified address, is uploaded from the ECU. MC Upload.vi will set the Memory Transfer Address pointer MTA0 to the appropriate value as defined in the Address cluster.

If you are using the XCP protocol, MC Upload.vi implements the XCP command UPLOAD. A data block of the specified length starting at the specified address is uploaded from the ECU. The Memory Transfer Address pointer MTA0 is post-incremented by the given number of data elements. If the slave device does not support block transfer mode, all uploaded data is transferred in a single response packet. If block transfer mode is supported, the uploaded data is transferred in multiple responses on the same request packet. For the master there are no limitations allowed concerning the maximum block size.

For further information on how to upload data and to use the MC Upload.vi command refer to the ASAM XCP Part 2 Protocol Layer Specification.
MC XCP Copy Cal Page.vi

Purpose
Forces a copy transaction of one calibration page to another.

Format

```
source page
source segment
ECU ref in
destination segment
destination page
error in
```

Input

- **Source page** specifies the logical page number of the source data page.
- **Source segment** specifies the logical segment number of the source data page.
- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.
- **Destination page** specifies the logical segment number of the destination data page.
- **Destination segment** specifies logical page number of the destination data page.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.
- **status** is TRUE if an error occurred. This VI will not execute when status is TRUE.
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC XCP Copy Cal Page.vi implements the XCP command COPY_CAL_PAGE and forces the slave to copy one calibration page to another. This command is only available if more than one calibration page is defined. In principal, any page of any segment can be copied to any page of any other segment but there may be restrictions.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
MC XCP Get Cal Page.vi

Purpose
Queries a calibration page setting.

Format

<table>
<thead>
<tr>
<th>Input</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mode</strong></td>
<td>specifies the access mode:</td>
</tr>
<tr>
<td><strong>Mode = 1</strong></td>
<td>The given page is used by the slave device application.</td>
</tr>
<tr>
<td><strong>Mode = 2</strong></td>
<td>The slave device XCP driver will access the given page.</td>
</tr>
<tr>
<td><strong>ECU ref in</strong></td>
<td>is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.</td>
</tr>
<tr>
<td><strong>Segment</strong></td>
<td>specifies the selected logical data segment number.</td>
</tr>
<tr>
<td><strong>Error in</strong></td>
<td>is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.</td>
</tr>
<tr>
<td><strong>status</strong></td>
<td>is TRUE if an error occurred. This VI will not execute when status is TRUE.</td>
</tr>
</tbody>
</table>
code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.

**Output**

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Page returns the logical data page number.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.

**Description**

MC XCP Get Cal Page.vi implements the XCP command GET_CAL_PAGE and queries the logical number for the calibration data page that is currently activated for the specified access mode and data segment.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
MC XCP Get ID.vi

Purpose
Queries session configuration or slave device identification.

Format

<table>
<thead>
<tr>
<th>reference in</th>
<th>reference out</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>id string</td>
</tr>
<tr>
<td>error in</td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

Reference in is the reference to any opened A2L database, a selected ECU, or an ECU which is already connected (with MC Database Open.vi, MC ECU Select.vi, MC ECU Open.vi, or MC ECU Connect.vi). The type of this reference depends on the property you want to get.

Type specifies the type of the requested identification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ASCII text</td>
</tr>
<tr>
<td>1</td>
<td>ASAM-MC2 filename without path and extension</td>
</tr>
<tr>
<td>2</td>
<td>ASAM-MC2 filename with path and extension</td>
</tr>
<tr>
<td>3</td>
<td>URL where the ASAM-MC2 file can be found</td>
</tr>
<tr>
<td>4</td>
<td>ASAM-MC2 file to upload</td>
</tr>
<tr>
<td>128..255</td>
<td>User defined</td>
</tr>
</tbody>
</table>

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.
**Chapter 5  ECU M&C API for LabVIEW**

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.

**Output**

Reference out is a copy of the reference in terminal which can be wired through subsequent ECU M&C VIs.

Id contains the queried identification string.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

source identifies the VI where the error occurred.

**Description**

MC XCP Get ID.vi implements the XCP command GET_ID and returns session configuration or slave device identification information of the selected ECU slave device. The supported types are implementation specific of the ECU slave device. The identification string is ASCII text format.
MC XCP Get Status.vi

Purpose
Queries the current session status from an ECU slave device.

Format

<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td>error in</td>
<td>session id</td>
</tr>
<tr>
<td></td>
<td>ECU ref out</td>
</tr>
<tr>
<td></td>
<td>session status</td>
</tr>
<tr>
<td></td>
<td>resource mask</td>
</tr>
<tr>
<td></td>
<td>error out</td>
</tr>
</tbody>
</table>

Input

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

Session Id returns the defined session configuration ID.

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Session status returns the current status of the selected ECU.
Resource mask is the current resource protection status of the selected ECU.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC XCP Get Status.vi implements the XCP command GET_STATUS and returns all current status information of the selected ECU slave device, including the status of the resource protection, pending store requests and the general status of data acquisition and stimulation.

Current Session Status

Session status contains a bit mask which is described below:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STORE_CAL_REQ</td>
<td>REQuest to STORE CALibration data:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—STORE_CAL_REQ mode is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—STORE_CAL_REQ mode is set.</td>
</tr>
<tr>
<td>1</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>STORE_DAQ_REQ</td>
<td>REQuest to STORE DAQ list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—STORE_DAQ_REQ mode is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—STORE_DAQ_REQ mode is set.</td>
</tr>
</tbody>
</table>
The STORE_CAL_REQ flag indicates a pending request to save the calibration data into non-volatile memory. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_STORE_CAL event packet.

The STORE_DAQ_REQ flag indicates a pending request to save the DAQ list setup in non-volatile memory. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_STORE_DAQ event packet.

The CLEAR_DAQ_REQ flag indicates a pending request to clear all DAQ lists in non-volatile memory. All ODT entries are reset to address = 0, extension = 0, size = 0 and bit_offset = FF. Session configuration ID is reset to 0. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_CLEAR_DAQ event packet. If the slave device does not support the requested mode, an ERR_OUT_OF_RANGE is returned.

The DAQ_RUNNING flag indicates that at least one DAQ list has been started and is in RUNNING mode.

The RESUME flag indicates that the slave is in RESUME mode.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>CLEAR_DAQ_REQ</td>
<td>REQuest to CLEAR DAQ configuration:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—CLEAR_DAQ_REQ is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—CLEAR_DAQ_REQ is set.</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>DAQ_RUNNING</td>
<td>Data Transfer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—The data transfer is not running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—The data transfer is running.</td>
</tr>
<tr>
<td>7</td>
<td>RESUME</td>
<td>RESUME Mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—The slave device is not in RESUME mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—The slave device is in RESUME mode.</td>
</tr>
</tbody>
</table>
Resource mask contains the current resource protection status as a bit mask described below:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAL/PAG</td>
<td>REQuest to STORE CALibration data:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—STORE_CAL_REQ mode is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—STORE_CAL_REQ mode is set.</td>
</tr>
<tr>
<td>1</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>DAQ</td>
<td>DAQ list commands (DIRECTION = DAQ):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—DAQ list commands are not protected with SEED &amp; Key mechanism.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—DAQ list commands are protected with SEED &amp; Key mechanism.</td>
</tr>
<tr>
<td>3</td>
<td>STIM</td>
<td>DAQ list commands (DIRECTION = STIM):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—DAQ list commands are not protected with SEED &amp; Key mechanism.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—DAQ list commands are protected with SEED &amp; Key mechanism.</td>
</tr>
<tr>
<td>4</td>
<td>PGM</td>
<td>ProGraMming commands:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—ProGraMming commands are not protected with SEED &amp; Key mechanism.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—ProGraMming commands are protected with SEED &amp; Key mechanism</td>
</tr>
<tr>
<td>5</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>Unused</td>
<td>—</td>
</tr>
</tbody>
</table>

The CAL/PAG flags indicates that all commands of the CALibration/PAGing group are protected and will return an ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.

The PGM flags indicates that all the commands of the ProGraMming group are protected and will return a ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.
The parameter **Session Id** contains the Session configuration ID. The session configuration ID must be set by a prior **MC XCP Set Request.vi** call with STORE_DAQ_REQ set. This allows the master device to verify that automatically started DAQ lists contain the expected data transfer configuration.
MC XCP Program Prepare.vi

Purpose
Prepares the programming of non volatile memory.

Format

```
ECU ref in

address

code size

error in

ECU ref out

ECU

ECU ref out

error out
```

Input

**ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

**Address** is a cluster which contains the following values.

- **Address** specifies the address part of the source address.
- **Extension** contains the extension part of the source address.

**Code size** determines the size of data code to be downloaded by the subsequent memory programming.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is TRUE if an error occurred. This VI is not executed when **status** is TRUE.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.
**Output**

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.
- **code** is the error code number identifying an error. A value of **0** means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

**Description**

**MC XCP Program Prepare.vi** may be used to indicate a data download as a pre-condition for non-volatile memory reprogramming. The Memory Transfer address (MTA) pointer is set to the volatile memory location specified by the parameters **Address** and **Extension**. The download itself is done by using subsequent standard commands like **MC Download.vi**. The slave device must ensure that the target memory area is available and it is in an operational state which permits the download of code. If not, an error will be returned.

**MC XCP Program Prepare.vi** implements the optional XCP PROGRAM_PREPARE command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM XCP Part 2 Protocol Layer Specification.
MC XCP Program Verify.vi

Purpose
Performs a non-volatile memory certification task on the ECU device.

Format

<table>
<thead>
<tr>
<th>verification mode</th>
<th>ECU ref in</th>
<th>verification type</th>
<th>verification value</th>
<th>ECU ref out</th>
<th>error in</th>
<th>error out</th>
</tr>
</thead>
</table>

Input

- **Verification mode** specifies the type of the requested identification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Request to start internal routine.</td>
</tr>
<tr>
<td>1</td>
<td>Send a Verification Value stored in <strong>Verification value</strong>.</td>
</tr>
</tbody>
</table>

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.

- **Verification type** specifies the Verification Type of the requested program verification. The Verification Type is a bit mask described below:

<table>
<thead>
<tr>
<th>Verification Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>Calibration area(s) of the flash.</td>
</tr>
<tr>
<td>0x0002</td>
<td>Code area(s) of the flash.</td>
</tr>
<tr>
<td>0x0004</td>
<td>Complete flash content.</td>
</tr>
<tr>
<td>0x0008 ... 0x0080</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x0100 ... 0xFF00</td>
<td>User defined.</td>
</tr>
</tbody>
</table>
**Verification value** contains the selected verification value if Mode=1.

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

**Output**

**ECU ref out** is the same as **ECU ref in**. Wire the task reference to subsequent VIs for this task.

**Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.

- **status** is **TRUE** if an error occurred.

- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.

- **source** identifies the VI where the error occurred.

**Description**

**MC XCP Program Verify.vi** may be used to verify the success of non-volatile memory reprogramming.

With **Verification mode** set to 00 the master may request the slave to begin internal test routines to check whether the new flash contents fits to the rest of the flash. Only the result is of interest. With **Verification mode** set to 01, the master may tell the slave that it will be sending a **Verification value** to the slave. The definition of the **Verification mode** is
The master receives the **Verification mode** from the project-specific programming flow control and passes it to the slave.

**MC XCP Program Verify.vi** implements the optional XCP PROGRAM_VERIFY command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM *XCP Part 2 Protocol Layer Specification*. 
Chapter 5  ECU M&C API for LabVIEW

MC XCP Set Cal Page.vi

Purpose
Sets a calibration page.

Format

<table>
<thead>
<tr>
<th>mode</th>
<th>ECU ref out</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ECU ref in</td>
</tr>
<tr>
<td></td>
<td>segment</td>
</tr>
<tr>
<td></td>
<td>page</td>
</tr>
<tr>
<td></td>
<td>error in</td>
</tr>
</tbody>
</table>

Input

Mode is a bit mask described below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The given page is used by the slave device application.</td>
</tr>
<tr>
<td>1</td>
<td>The slave device XCP driver will access the given page.</td>
</tr>
<tr>
<td>2</td>
<td>Unused.</td>
</tr>
<tr>
<td>3</td>
<td>Unused.</td>
</tr>
<tr>
<td>4</td>
<td>Unused.</td>
</tr>
<tr>
<td>5</td>
<td>Unused.</td>
</tr>
<tr>
<td>6</td>
<td>Unused.</td>
</tr>
<tr>
<td>7</td>
<td>The logical segment number is ignored. The command applies to all segments.</td>
</tr>
</tbody>
</table>

ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Segment specifies the selected logical data segment number.

Page specifies the logical data page number.
**Chapter 5  ECU M&C API for LabVIEW**

**Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

- **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.
- **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
- **source** identifies the VI where the error occurred.

**Output**

- **ECU ref out** is the same as **ECU ref**. Wire the task reference to subsequent VIs for this task.
- **Error out** describes error conditions. If the **Error in** cluster indicated an error, the **Error out** cluster contains the same information. Otherwise, **Error out** describes the error status of this VI.
  - **status** is **TRUE** if an error occurred.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.

**Description**

**MC XCP Set Cal Page.vi** implements the XCP command SET_CAL_PAGE and sets the access mode for a calibration data segment, if the slave device supports calibration data page switching. A calibration data segment and its pages are specified by logical numbers.

Refer to the ASAM XCP *Part 2 Protocol Layer Specification* for more information on how to set up a request.
MC XCP Set Request.vi

Purpose
Performs a request to save session and device information to non-volatile memory.

Format

Input

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Request to store calibration data in non-volatile memory.</td>
<td></td>
</tr>
<tr>
<td>1 Unused.</td>
<td></td>
</tr>
<tr>
<td>2 Request to save all DAQ lists, which have been selected with START_STOP_DAQ_LIST(Select) into non-volatile memory. The slave also must store the session configuration ID in non-volatile memory. Upon saving, the slave first must clear any DAQ list configuration that might already be stored in non-volatile memory.</td>
<td></td>
</tr>
<tr>
<td>3 Request to clear all DAQ lists in non-volatile memory. All ODT entries reset to address = 0, extension = 0, size = 0 and bit_offset = FF. Session configuration ID reset to 0.</td>
<td></td>
</tr>
<tr>
<td>4 Unused.</td>
<td></td>
</tr>
<tr>
<td>5 Unused.</td>
<td></td>
</tr>
<tr>
<td>6 Unused.</td>
<td></td>
</tr>
<tr>
<td>7 Unused.</td>
<td></td>
</tr>
</tbody>
</table>
ECU ref in is the task reference which links to the selected ECU. This reference is originally returned from MC ECU Open.vi or MC ECU Select.vi, and then wired through subsequent VIs.

Session ID is a session configuration ID that is stored in non-volatile memory together with the information requested by the Mode parameter.

Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the error in cluster to error out.

status is TRUE if an error occurred. This VI will not execute when status is TRUE.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.
Description

**MC XCP Set Request.vi** implements the XCP command SET_REQUEST and is used to save session configuration information into non-volatile memory in the ECU.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
MC XCP Set Segment Mode.vi

Purpose
Sets the mode of a specified segment.

Format

```
<table>
<thead>
<tr>
<th>ECU ref in</th>
<th>Segment</th>
<th>Mode</th>
<th>Error in</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>ECU ref out</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ECU ref out</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>error out</td>
</tr>
</tbody>
</table>
```

Input

- **ECU ref in** is the task reference which links to the selected ECU. This reference is originally returned from **MC ECU Open.vi** or **MC ECU Select.vi**, and then wired through subsequent VIs.
- **Segment** specifies the logical data segment number.
- **Mode** specifies the mode for the segment.
- **Error in** is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.
  - **status** is **TRUE** if an error occurred. This VI will not execute when **status** is **TRUE**.
  - **code** is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire the error cluster to a LabVIEW error-handling VI, such as the **Simple Error Handler**.
  - **source** identifies the VI where the error occurred.
Output

ECU ref out is the same as ECU ref in. Wire the task reference to subsequent VIs for this task.

Error out describes error conditions. If the Error in cluster indicated an error, the Error out cluster contains the same information. Otherwise, Error out describes the error status of this VI.

status is TRUE if an error occurred.

code is the error code number identifying an error. A value of 0 means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the code, wire the error cluster to a LabVIEW error-handling VI, such as the Simple Error Handler.

source identifies the VI where the error occurred.

Description

MC XCP Set Segment Mode.vi implements the XCP command SET_SEGMENT_MODE and sets the selected segment into the specified mode. If Mode = 0 the segment disables the FREEZE mode, if Mode = 1 the segment is set to FREEZE mode through an XCP STORE_CAL_REQ operation.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
ECU M&C API for C

This chapter lists the ECU M&C functions and describes the format, purpose, and parameters. Unless otherwise stated, each ECU M&C function suspends execution of the calling thread until it completes. The functions in this chapter are listed alphabetically.

Section Headings

The following are section headings found in the ECU M&C API for C functions.

Purpose

Each function description includes a brief statement of the purpose of the function.

Format

The format section describes the format of each function for the C programming language.

Input and Output

The input and output parameters for each function are listed.

Description

The description section gives details about the purpose and effect of each function.

List of Data Types

The following data types are used with functions of the ECU M&C API for C.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>i8</td>
<td>8-bit signed integer</td>
</tr>
<tr>
<td>i16</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>i32</td>
<td>32-bit signed integer</td>
</tr>
</tbody>
</table>
Table 6-1. Data Types for the ECU M&C API for C (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>u16</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>u32</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>f32</td>
<td>32-bit floating-point number</td>
</tr>
<tr>
<td>f64</td>
<td>64-bit floating-point number</td>
</tr>
<tr>
<td>str</td>
<td>ASCII string represented as an array of characters terminated by null character ('\0'). This type is used with output strings. str is typically used in the ECU M&amp;C API as a pointer to a string, as char*.</td>
</tr>
<tr>
<td>cstr</td>
<td>ASCII string represented as an array of characters terminated by null character ('\0'). This type is used with input strings. cstr is typically used in the ECU M&amp;C API as a pointer to a string, as const char*.</td>
</tr>
<tr>
<td>mcTypeTaskRef</td>
<td>Reference to an initialized database task, ECU task, or Measurement task.</td>
</tr>
<tr>
<td>mcAddress</td>
<td>C struct which represents the target address for a specific CCP operation in the ECU.</td>
</tr>
</tbody>
</table>

List of Functions

The following table contains an alphabetical list of the ECU M&C Toolkit API functions.

Table 6-2. Functions for the ECU M&C API for C

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcBuildChecksum</td>
<td>Calculates a checksum over a defined memory range within the ECU.</td>
</tr>
<tr>
<td>mcCalculateChecksum</td>
<td>Calculates the checksum of a data block in memory.</td>
</tr>
<tr>
<td>mcCCPActionService</td>
<td>Calls an implementation-specific action service on the ECU.</td>
</tr>
<tr>
<td>mcCCPDiagService</td>
<td>Calls an implementation-specific diagnostic service on the ECU.</td>
</tr>
<tr>
<td>mcCCPGetActiveCalPage</td>
<td>Retrieves the ECU Memory Transfer Address pointer to the calibration data page.</td>
</tr>
<tr>
<td>mcCCPGetResult</td>
<td>Uploads data from the ECU when the Memory Transfer Address pointer 0 (MTA0) has been set.</td>
</tr>
</tbody>
</table>
Table 6-2. Functions for the ECU M&C API for C (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcCCPGetSessionStatus</td>
<td>Retrieves the current status of the Calibration Session.</td>
</tr>
<tr>
<td>mcCCPGetVersion</td>
<td>Retrieves CCP version implemented in the ECU.</td>
</tr>
<tr>
<td>mcCCPMoveMemory</td>
<td>Moves a memory block on the ECU.</td>
</tr>
<tr>
<td>mcCCPSelectCalPage</td>
<td>Sets the specified address to be the start address of the calibration data page.</td>
</tr>
<tr>
<td>mcCCPSetSessionStatus</td>
<td>Updates the ECU with the current state of the calibration session.</td>
</tr>
<tr>
<td>mcCharacteristicRead</td>
<td>Reads all data from a named Characteristic on the ECU which is identified by the ECU Reference handle.</td>
</tr>
<tr>
<td>mcCharacteristicReadSingleValue</td>
<td>Reads a single value from a named Characteristic on the ECU which is identified by the ECU Reference handle.</td>
</tr>
<tr>
<td>mcCharacteristicWrite</td>
<td>Downloads data to a Characteristic for a selected ECU.</td>
</tr>
<tr>
<td>mcCharacteristicWriteSingleValue</td>
<td>Writes a single value to a named Characteristic on the ECU.</td>
</tr>
<tr>
<td>mcClearMemory</td>
<td>Clears the contents of the specified ECU memory.</td>
</tr>
<tr>
<td>mcConversionCreate</td>
<td>Creates a signal conversion object in memory.</td>
</tr>
<tr>
<td>mcDAQClear</td>
<td>Stops communication for the Measurement task and clears the task.</td>
</tr>
<tr>
<td>mcDAQInitialize</td>
<td>Initializes a Measurement task for the specified Measurement channel list.</td>
</tr>
<tr>
<td>mcDAQListInitialize</td>
<td>Defines a DAQ list on a specific DAQ list number and initializes the Measurement task for the specified Measurement channel list. Initializes a Measurement task for the specified Measurement channel list.</td>
</tr>
<tr>
<td>mcDAQRead</td>
<td>Reads samples from a Measurement task. Samples are obtained from received CAN messages.</td>
</tr>
<tr>
<td>mcDAQReadTimestamped</td>
<td>Reads timestamped samples from a DAQ task initialized with the selected mode of mcDAQModeDAQListTimeStamped.</td>
</tr>
<tr>
<td>mcDAQStartStop</td>
<td>Starts or stops the transmission of the DAQ lists for the specified Measurement task.</td>
</tr>
<tr>
<td>mcDAQWrite</td>
<td>Writes samples to an ECU DAQ list.</td>
</tr>
</tbody>
</table>
Table 6-2. Functions for the ECU M&C API for C (Continued)

<table>
<thead>
<tr>
<th>Function</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>mcDatabaseClose</td>
<td>Stops transmission of the DAQ lists for the specified Measurement task.</td>
</tr>
<tr>
<td>mcDatabaseClose</td>
<td>Closes a specified A2L Database reference.</td>
</tr>
<tr>
<td>mcDatabaseOpen</td>
<td>Opens a specified A2L Database.</td>
</tr>
<tr>
<td>mcDatabaseOpenEx</td>
<td>Creates an A2L database in memory, for using the ECU M&amp;C Toolkit functions without access to a valid ASAM A2L file.</td>
</tr>
<tr>
<td>mcDoubleToText</td>
<td>Converts a numerical value to a text string using an enumeration or range text type scaling.</td>
</tr>
<tr>
<td>mcDownload</td>
<td>Downloads data to an ECU.</td>
</tr>
<tr>
<td>mcECUConnect</td>
<td>Establishes communication to the selected ECU through CCP. After a successful ECU Connect you can create a Measurement Task or read/write a Characteristic.</td>
</tr>
<tr>
<td>mcECUCreate</td>
<td>Creates an ECU object in memory.</td>
</tr>
<tr>
<td>mcECUDeselect</td>
<td>Deselects an ECU and invalidates the ECU reference handle.</td>
</tr>
<tr>
<td>mcECUDisconnect</td>
<td>Disconnects CCP communication to the selected ECU.</td>
</tr>
<tr>
<td>mcECUSelectEx</td>
<td>Selects an ECU from the names stored in an A2L database.</td>
</tr>
<tr>
<td>mcEventCreate</td>
<td>Creates an Event object in memory.</td>
</tr>
<tr>
<td>mcGeneric</td>
<td>Sends a generic CCP command.</td>
</tr>
<tr>
<td>mcGetNames</td>
<td>Retrieves a comma-separated list of ECU names, Measurement names, Characteristic names, Event names, Characteristic pages, or Group names from a specified A2L database.</td>
</tr>
<tr>
<td>mcGetNamesLength</td>
<td>Retrieves the amount of memory required to store the names returned by mcGetNames.</td>
</tr>
<tr>
<td>mcGetProperty</td>
<td>Retrieves a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.</td>
</tr>
<tr>
<td>mcMeasurementCreate</td>
<td>Creates a Measurement object in memory.</td>
</tr>
<tr>
<td>mcMeasurementRead</td>
<td>Reads a single Measurement value from the ECU.</td>
</tr>
<tr>
<td>mcMeasurementWrite</td>
<td>Writes a single Measurement value to the ECU.</td>
</tr>
<tr>
<td>mcProgram</td>
<td>Programs a memory block on the ECU.</td>
</tr>
<tr>
<td>Function</td>
<td>Purpose</td>
</tr>
<tr>
<td>---------------------------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>mcProgramReset</td>
<td>Indicates the end of a programming sequence.</td>
</tr>
<tr>
<td>mcProgramStart</td>
<td>Indicates the start of a programming sequence.</td>
</tr>
<tr>
<td>mcSetProperty</td>
<td>Sets a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.</td>
</tr>
<tr>
<td>mcStatusToString</td>
<td>Converts a status code into a descriptive string.</td>
</tr>
<tr>
<td>mcTextToDouble</td>
<td>Converts a text string to a numerical value using an enumeration or range text scaling.</td>
</tr>
<tr>
<td>mcUpload</td>
<td>Uploads data from an ECU.</td>
</tr>
<tr>
<td>mcXCPCopyCalPage</td>
<td>Forces a copy transaction of one calibration page to another.</td>
</tr>
<tr>
<td>mcXCPGetCalPage</td>
<td>Queries a calibration page setting.</td>
</tr>
<tr>
<td>mcXCPGetId</td>
<td>Queries session configuration or slave device identification.</td>
</tr>
<tr>
<td>mcXCPGetStatus</td>
<td>Queries the current session status from an ECU slave device.</td>
</tr>
<tr>
<td>mcXCPProgramPrepare</td>
<td>Prepares the programming of non volatile memory.</td>
</tr>
<tr>
<td>mcXCPProgramVerify</td>
<td>Performs a non-volatile memory certification task on the ECU device.</td>
</tr>
<tr>
<td>mcXCPSetCalPage</td>
<td>Sets a calibration page.</td>
</tr>
<tr>
<td>mcXCPSetRequest</td>
<td>Performs a request to save session and device information to non-volatile memory.</td>
</tr>
<tr>
<td>mcXCPSetSegmentMode</td>
<td>Sets the mode of a specified segment.</td>
</tr>
</tbody>
</table>

Table 6-2. Functions for the ECU M&C API for C (Continued)
**mcBuildChecksum**

**Purpose**
Calculates a checksum over a defined memory range within the ECU.

**Format**

```c
mcTypeStatus mcBuildChecksum(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u32 BlockSize,
    u8 *ChecksumType,
    u8 *SizeOfChecksum
    u32 *Checksum);
```

**Input**

- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.
- **Address**
  Configures the target address for the checksum operation in the ECU. `mcAddress` is a C struct consisting of:
  - **Address**
    Specifies the address part of the target address.
  - **Extension**
    Extension contains the extension part of the target address.
  - **BlockSize**
    BlockSize determines the size of the block on which the checksum must be calculated.
  - **ChecksumType**
    ChecksumType returns the type of the calculated checksum. For CCP, ChecksumType is 0xFF. For XCP, refer to the Description section.

**Output**

- **SizeOfChecksum**
  SizeOfChecksum returns the size in bytes of the calculated checksum.
- **Checksum**
  Checksum is the calculated checksum.

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.
Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

`mcBuildChecksum` is used to calculate the checksum of a specified memory block inside the ECU starting at the selected Address.

If you are using the CCP protocol, `mcBuildChecksum` implements the CCP BUILD_CHKSUM command. The checksum algorithm is not specified by CCP and the checksum algorithm may be different on different devices.

If you are using the XCP protocol, `mcBuildChecksum` implements the BUILD_CHECKSUM command of the XCP specification. The result of the checksum calculation is returned in `Checksum` regardless of the checksum type. The following values for `ChecksumType` are defined in the XCP specification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>XCP_ADD_11</td>
<td>Add BYTE into a BYTE checksum, ignore overflows</td>
</tr>
<tr>
<td>0x02</td>
<td>XCP_ADD_12</td>
<td>Add BYTE into a WORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x03</td>
<td>XCP_ADD_14</td>
<td>Add BYTE into a DWORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x04</td>
<td>XCP_ADD_22</td>
<td>Add WORD into a WORD checksum, ignore overflows, <code>blocksize</code> must be modulo 2</td>
</tr>
<tr>
<td>0x05</td>
<td>XCP_ADD_24</td>
<td>Add WORD into a DWORD checksum, ignore overflows, <code>blocksize</code> must be modulo 2</td>
</tr>
<tr>
<td>0x06</td>
<td>XCP_ADD_44</td>
<td>Add DWORD into DWORD, ignore overflows, <code>blocksize</code> must be modulo 4</td>
</tr>
<tr>
<td>0x07</td>
<td>XCP_CRC_16</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x08</td>
<td>XCP_CRC_16_CITT</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x09</td>
<td>XCP_CRC_32</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0xFF</td>
<td>XCP_USER_DEFINED</td>
<td>User defined algorithm, in externally calculated function</td>
</tr>
</tbody>
</table>
With `ChecksumType XCP_USER_DEFINED`, the Slave may indicate that the Master which calculates the checksum must use a user-defined algorithm implemented in an externally calculated function (for instance, Win32 DLL, UNIX shared object file, etc.). The master retrieves the name of the external function file to be used for this slave from the ASAM MCD 2MC description file.

The CRC algorithms are specified by the following parameters:

<table>
<thead>
<tr>
<th>Name</th>
<th>Width</th>
<th>Poly</th>
<th>Init</th>
<th>Refin</th>
<th>Refout</th>
<th>XORout</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCP_CRC_16</td>
<td>16</td>
<td>0x8005</td>
<td>0x0000</td>
<td>TRUE</td>
<td>TRUE</td>
<td>0x0000</td>
</tr>
<tr>
<td>XCP_CRC16_CITT</td>
<td>16</td>
<td>0x1021</td>
<td>0xFFFF</td>
<td>FALSE</td>
<td>FALSE</td>
<td>0x0000</td>
</tr>
<tr>
<td>XCP_CRC_32</td>
<td>32</td>
<td>0x04C11DB7</td>
<td>0xFFFFFFFF</td>
<td>TRUE</td>
<td>TRUE</td>
<td>0xFFFFFFFF</td>
</tr>
</tbody>
</table>

**Name**

The name of the algorithm. A string value starting with “XCP_”.

**Width**

The width of the algorithm expressed in bits. This is one less than the width of the Poly.

**Poly**

The polynomial. This is a binary value specified as a hexadecimal number. The top bit of the Poly should be omitted. For example, if the Poly is 0x10110, you should specify 0x06. An important aspect of this parameter is that it represents the unreflected polynomial. The bottom of this parameter is always the least significant bit (LSB) of the divisor during the division, regardless of whether the algorithm is reflected.

**Init**

This parameter specifies the initial value of the register when the algorithm starts. This is the value to be assigned to the register in the direct table algorithm. In the table algorithm, we may think of the register always commencing with the value zero, and this value being XORed into the register after the \( n \)th bit iteration. This parameter should be specified as a hexadecimal number.

**Refin**

A Boolean parameter. If it is FALSE, input bytes are processed with bit 7 being treated as the most significant bit (MSB) and bit 0 being treated as the least significant bit. If this parameter is TRUE, each byte is reflected before being processed.
Refout
A Boolean parameter. If it is set to FALSE, the final value in the register is fed into the XORout stage directly. If this parameter is TRUE, the final register value is reflected first.

XORout
This is a width-bit value that should be specified as hexadecimal number. It is XORed to the final register value (after the Refout stage) before the value is returned as the official checksum.

For more detailed information about CRC algorithms, refer to:
http://www.repairfaq.org/filipg/LINK/F_crc_v34.html
mcCalculateChecksum

Purpose
Calculates the checksum of a data block in memory.

Format
```c
mcTypeStatus mcCalculateChecksum(
    mcTypeTaskRef ECURefNum,
    u32 BlockSize,
    u8 *Data,
    u8 TypeOfChecksum,
    u8 *SizeOfChecksum,
    u32 *Checksum);
```

Input
- **ECURefNum**
  - ECURefNum is the task reference which links to the selected ECU.
  - This reference is originally returned from `mcECUSelectEx`.
- **BlockSize**
  - BlockSize determines the size of the block on which the checksum must be calculated.
- **TypeOfChecksum**
  - TypeOfChecksum specifies the type of the calculated checksum.

Output
- **Data**
  - Data is a byte array upon which the checksum calculation is performed.
- **SizeofChecksum**
  - SizeofChecksum returns the size in bytes of the calculated checksum.
- **Checksum**
  - Checksum is the calculated checksum.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

mcCalculateChecksum implements a checksum calculation over a given data block. The checksum algorithm is performed over a dedicated checksum function provided by a specific DLL. The name of the Checksum DLL is defined in the A2L data base and can be changed by the application by the mcSetProperty function using the mcPropECU_Checksum property.

If you are using the CCP protocol, TypeOfChecksum must be set to 0xFFh, since CCP only supports an external checksum DLL. If you are using XCP, the following values for TypeOfChecksum are defined in the XCP specification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>XCP_ADD_11</td>
<td>Add BYTE into a BYTE checksum, ignore overflows</td>
</tr>
<tr>
<td>0x02</td>
<td>XCP_ADD_12</td>
<td>Add BYTE into a WORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x03</td>
<td>XCP_ADD_14</td>
<td>Add BYTE into a DWORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x04</td>
<td>XCP_ADD_22</td>
<td>Add WORD into a WORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x05</td>
<td>XCP_ADD_24</td>
<td>Add WORD into a DWORD checksum, ignore overflows</td>
</tr>
<tr>
<td>0x06</td>
<td>XCP_ADD_44</td>
<td>Add DWORD into DWORD, ignore overflows, blocksize must be modulo 4</td>
</tr>
<tr>
<td>0x07</td>
<td>XCP_CRC_16</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x08</td>
<td>XCP_CRC_16_CITT</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0x09</td>
<td>XCP_CRC_32</td>
<td>Refer to CRC error detection algorithms</td>
</tr>
<tr>
<td>0xFF</td>
<td>XCP_USER_DEFINED</td>
<td>User defined algorithm, in externally calculated function</td>
</tr>
</tbody>
</table>

For a detailed description of the checksum algorithm refer to the mcBuildChecksum command or the XCP Part 2 Protocol Layer Specification.

For more detailed information about CRC algorithms, please refer to:
http://www.repairfaq.org/filipg/LINK/F_crc_v34.html
mcCCPActionService

Purpose
Calls an implementation-specific action service on the ECU (CCP only).

Format
```
mcTypeStatus mcCCPActionService(
    mcTypeTaskRef ECURefNum,
    u16 ServiceNo,
    u8 Params[4],
    u8 *ResultLength,
    u8 *DataType);
```

Input
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

- **ServiceNo**
  ServiceNo determines the service that is executed inside the ECU. For more information about the services that are implemented in the ECU refer to the documentation for the ECU.

- **Params**
  Params passes the parameters of the service function as an array of bytes to the ECU. Since the parameters and their data types are specific to the ECU implementation, you are responsible of providing the required parameters in the correct byte ordering.

Output
- **ResultLength**
  ResultLength indicates the amount of data that can be uploaded from the ECU as a result of the execution of the service. The result of this service can be accessed by calling the function `mcCCPGetResult` right after `mcCCPActionService`.

- **DataType**
  DataType is a data type qualifier that determines the data format of the result.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
**Description**

`mcCCPActionService` implements the CCP command ACTION_SERVICE. The ECU carries out the requested service and automatically sets the Memory Transfer Address MTA0 to the location from which the CCP master may upload the requested action service return information (if applicable).

The result of this service can be accessed by calling the function `mcCCPGetResult` right after `mcCCPActionService`. 
mcCCPDiagService

Purpose
Calls an implementation-specific diagnostic service on the ECU (CCP only).

Format
```
mcTypeStatus mcCCPDiagService(
    mcTypeTaskRef ECURefNum,
    u16 ServiceNo,
    u8 Params[4],
    u8 *ResultLength,
    u8 *DataType);
```

Input
- **ECURefNum** is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **ServiceNo** determines the diagnostic service that is executed inside the ECU. For more information about the services that are implemented in the ECU refer to the documentation for the ECU.
- **Params** passes an array of bytes to the ECU that might be needed by the ECU to run the diagnostic service. Since the definition of the parameters is specific to the implementation of the ECU, the parameters can only be passed as an array of bytes. It is your responsibility to pass the correct number of parameters in the correct byte ordering to this function.

Output
- **ResultLength** returns the number of bytes that can be uploaded from the ECU as a result of the execution of the service. The result of this service can be accessed by calling the function `mcCCPGetResult` right after `mcCCPDiagService`.
- **DataType** is a data type qualifier which provides information about the data type of the result of the diagnostic service.
Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcCCPDiagService implements the CCP command DIAG_SERVICE which calls a diagnostic service on the ECU and waits until it is finished. The selected ServiceNo specifies the diagnostic service that must be executed inside the ECU. For more information about the available services that are implemented in the ECU refer to the documentation for the ECU.

The result of this service can be accessed by calling the function mcCCPGetResult right after mcCCPDiagService.
mcCCPGetActiveCalPage

Purpose
Retrieves the ECU Memory Transfer Address pointer to the calibration data page (CCP only).

Format
```c
mcTypeStatus mcCCPGetActiveCalPage(
    mcTypeTaskRef ECURefNum,
    mcAddress *Address);
```

Input
- `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

Output
- `Address` returns the address for the active calibration page in the ECU.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcCCPGetActiveCalPage` retrieves the start address of the active calibration data page in the ECU memory.
**mcCCPGetResult**

**Purpose**
Uploads data from the ECU when the Memory Transfer Address pointer 0 (MTA0) has been set (CCP only).

**Format**
```
mcTypeStatus mcCCPGetResult(
    mcTypeTaskRef ECURefNum,
    u32 BlockSize,
    u8 *Data);
```

**Input**
- **ECURefNum**
  - ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **BlockSize**
  - BlockSize determines the size of the data block to be uploaded from the ECU.

**Output**
- **Data**
  - Data contains the data uploaded from the ECU memory.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**
This function uploads data from the ECU. It is assumed that the Memory Transfer Address 0 (MTA0) has already been set to the start address of the data to be uploaded. Functions like `mcCCPActionService` or `mcCCPDiragService` implicitly set the Memory Transfer Address 0 (MTA0) to the beginning of their result. To upload data from a specified address, use `mcUpload` instead.
mcCCPGetSessionStatus

Purpose
Retrieves the current status of the Calibration Session (CCP only).

Format
```c
mcTypeStatus mcCCPGetSessionStatus(
    mcTypeTaskRef ECURefNum,
    u8 *SessionStatus,
    u8 *StatusQualifier,
    u8 *AdditionalStatus);
```

Input
- **ECURefNum**: ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.

Output
- **SessionStatus**: The current SessionStatus which is returned from the ECU.
- **StatusQualifier**: The additional StatusQualifier is manufacturer and/or project specific and is not part of the CCP protocol specification.
- **AdditionalStatus**: If the StatusQualifier does not contain additional status information, AdditionalStatus must be set to FALSE. If AdditionalStatus is not FALSE, it may be used to determine the type of the additional status information.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcCCPGetSessionStatus retrieves the current calibration session status of the ECU. The return value SessionStatus is a bit mask that represents several session states inside the ECU. StatusQualifier and AdditionalStatus contain additional status information. The content of these parameters is ECU specific and not defined by CCP. For more information about the parameter SessionStatus, refer to the description of mcCCPSetSessionStatus.
**mcCCPGetVersion**

**Purpose**
Retrieves CCP version implemented in the ECU (CCP only).

**Format**

```c
mcTypeStatus mcCCPGetVersion(
    mcTypeTaskRef ECURefNum,
    u8 *MajorVersion,
    u8 *MinorVersion);
```

**Input**

- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

**Output**

- **MajorVersion**
  MajorVersion returns the major version number of the CCP implementation.

- **MinorVersion**
  MinorVersion returns the minor version number of the CCP implementation.

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

`mcCCPGetVersion` can be used to query the CCP version implemented in the ECU. This command performs a mutual identification of the protocol version in the slave device to agree on a common protocol version.

`mcCCPGetVersion` implements the CCP command GET_CCP_VERSION defined by the CCP specification.
mcCCPMoveMemory

Purpose
Moves a memory block on the ECU (CCP only).

Format
mcTypeStatus mcCCPMoveMemory(
    mcTypeTaskRef ECURefNum,
    mcAddress Source,
    mcAddress Destination,
    u32 BlockSize);

Input
ECURefNum ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.
Source Configures the source address for the memory move operation in the ECU. mcAddress is a C struct consisting of:
    Address Specifies the address part of the source address.
    Extension Extension contains the extension part of the source address.
Destination Configures the destination address for the memory move operation in the ECU. mcAddress is a C struct consisting of:
    Address Specifies the address part of the destination address.
    Extension Extension contains the extension part of the destination address.
BlockSize BlockSize determines the size of memory block in bytes which should be moved from the source address to the destination address.

Output
Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.
Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

`mcCCPMoveMemory` is used to move the memory contents of an ECU from one memory location to another. Before calling the CCP MOVE command this function sets the Memory Transfer Address pointers MTA0 as defined in the source struct and MTA1 as defined in the destination struct to appropriate values.

`mcCCPMoveMemory` implements the CCP command MOVE defined by the CCP specification.
**mcCCPSelectCalPage**

**Purpose**
Sets the specified address to be the start address of the calibration data page (CCP only).

**Format**
```c
mcTypeStatus mcCCPSelectCalPage(
    mcTypeTaskRef ECURefNum, 
    mcAddress Address); 
```

**Input**
- **ECURefNum**
  
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

- **Address**
  
  Configures the target address for the programming operation in the ECU. `mcAddress` is a C struct consisting of:
  
  - **Address**
    
    Specifies the address part of the target address.
  
  - **Extension**
    
    Extension contains the extension part of the address.

**Output**

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

`mcCCPSelectCalPage` implements the CCP command `SELECT_CAL_PAGE`. This command sets the beginning of the calibration data page to the specified address within the ECU.
mcCCPSetSessionStatus

Purpose
Updates the ECU with the current state of the calibration session (CCP only).

Format
```c
mcTypeStatus mcCCPSetSessionStatus(
    mcTypeTaskRef ECURefNum,
    u8 SessionStatus);
```

Input
- `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`
- `SessionStatus` contains the new status to be set in the ECU.

Output

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcCCPSetSessionStatus` implements the CCP command SET_S_STATUS and is used to keep the ECU informed about the current state of the calibration session. The session status bits of an ECU can be read and written. Possible conditions are: reset on power-up, session log-off, and in applicable error conditions. The calibration session status is organized as a bit mask with the following assignment.

Table 6-3. Bit Mask Assignments for Calibration Session Status

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CAL</td>
<td>Calibration data initialized.</td>
</tr>
<tr>
<td>1</td>
<td>DAQ</td>
<td>DAQ list(s) initialized.</td>
</tr>
<tr>
<td>2</td>
<td>RESUME</td>
<td>Request to save DAQ set-up during shutdown in CCP slave. CCP slave automatically restarts DAQ after start-up.</td>
</tr>
</tbody>
</table>
### Table 6-3. Bit Mask Assignments for Calibration Session Status (Continued)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>STORE</td>
<td>Request to save calibration data during shut-down in CCP slave.</td>
</tr>
<tr>
<td>7</td>
<td>RUN</td>
<td>Session in progress.</td>
</tr>
</tbody>
</table>
mcCharacteristicRead

Purpose
Reads all data from a named Characteristic on the ECU which is identified by the ECU Reference handle.

Format
```
mcTypeStatus mcCharacteristicRead(
    mcTypeTaskRef ECURefNum,
    char *CharacteristicName,
    f64 *Values,
    u32 NumberOfValues);
```

Input
- **ECURefNum**: ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **CharacteristicName**: CharacteristicName is the name of the Characteristic defined in the A2L database file.
- **NumberOfValues**: Specifies the number of values to read. To determine the dimension of the Characteristic use the `mcGetProperty` function upfront using the parameter `mcPropChar_Dimension`. To determine the size of each dimension use the `mcGetProperty` function with the parameter `mcPropChar_Sizes`.

Output
- **Values**: Returns a single value, a 1-dimensional array, or a 2-dimensional array of values for the selected Characteristic.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcCharacteristicRead` reads values from a named Characteristic on the ECU which is identified by the ECU Reference handle. The function returns a double, 1D, or 2D array.
**mcCharacteristicReadSingleValue**

**Purpose**
Reads a single value from a named Characteristic on the ECU which is identified by the ECU Reference handle.

**Format**
```c
mcTypeStatus mcCharacteristicReadSingleValue(
    mcTypeTaskRef ECURefNum,
    char *CharacteristicName,
    f64 *Value,
    u32 X,
    u32 Y);
```

**Input**
- **ECURefNum**
  - ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **CharacteristicName**
  - CharacteristicName is the name of the Characteristic defined in the A2L database file.
- **X**
  - X is the horizontal index if the Characteristic consists of 1 or 2 dimensions.
- **Y**
  - Y is the vertical index if the Characteristic consists of 2 dimensions.

**Output**
- **Value**
  - Returns a single value from the selected Characteristic.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
**Description**

`mcCharacteristicReadSingleValue` reads a value from a named Characteristic on the ECU which is identified by the ECU Reference handle. The value to be read is identified by the \( X \) and \( Y \) indices.

If the Characteristic array is 0-dimensional, \( X \) and \( Y \) can be set to 0.

If the Characteristic array is 1-dimensional, \( Y \) can be set to 0.
Chapter 6 ECU M&C API for C

mcCharacteristicWrite

Purpose
Downloads data to a Characteristic for a selected ECU.

Format

mcTypeStatus mcCharacteristicWrite(
    mcTypeTaskRef ECURefNum,
    char *CharacteristicName,
    f64 Values,
    u32 NumberOfValues);

Input

ECURefNum ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.

CharacteristicName CharacteristicName is the name of the Characteristic defined in the A2L database file.

Values Values contains a pointer to a double, a double 1D, or 2D array which is sent to the ECU.

NumberOfValues Specifies the number of values to write for the task.

Output

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcCharacteristicWrite writes the value(s) of a named Characteristic to an ECU identified by the ECU reference handle ECURefNum.
**mcCharacteristicWriteSingleValue**

**Purpose**
Writes a single value to a named Characteristic on the ECU.

**Format**

```c
mcTypeStatus mcCharacteristicWriteSingleValue(
    mcTypeTaskRef ECURefNum,
    char *CharacteristicName,
    f64 Value,
    u32 X,
    u32 Y);
```

**Input**

- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

- **CharacteristicName**
  CharacteristicName is the name of the Characteristic defined in the A2L database file, to which the values are written.

- **Value**
  Value contains the value which is sent to the ECU.

- **X**
  X refers to the array offset of the Characteristic defined in the A2L database file as 1- or 2-dimensional. If the Characteristic is defined as 0-dimensional you can set X to 0.

- **Y**
  Y refers to the array offset of the Characteristic defined in the A2L database file as 2-dimensional. If the Characteristic is defined as 0- or 1-dimensional you can set Y to 0.

**Output**

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

`mcCharacteristicWriteSingleValue` writes a value to a named Characteristic on the ECU which is identified by the ECU Reference handle `ECURefNum`. The location to which the value is written is identified by the `X` and `Y` indices. If the Characteristic array is 0- or 1-dimensional, `Y` and/or `X` can be set to 0.
mcClearMemory

Purpose
Clears the contents of the specified ECU memory.

Format
```c
mcTypeStatus mcClearMemory(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u32 BlockSize);
```

Input
- **ECURefNum**: ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Address**: Configures the target address to be cleared in the ECU.
  - **Address**: Specifies the address part of the target address.
  - **Extension**: Extension contains the extension part of the target address.
- **BlockSize**: BlockSize determines the size of the block on which the checksum must be calculated.

Output
- **Return Value**: The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.
  
  Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcClearMemory can be used to clear the contents of the non-volatile memory prior to reprogramming it. The Memory Transfer Address 0 (MTA 0) is set to the start of the memory block automatically by this function. The size parameter is the size of the block to be erased.

If you are using the XCP protocol, mcClearMemory implements the PROGRAM_CLEAR command. Refer to the ASAM XCP specification for further information on how to clear parts of non-volatile memory in the ECU.
**mcConversionCreate**

**Purpose**

Creates a signal conversion object in memory.

**Format**

```c
mcTypeStatus mcConversionCreate(
    mcTypeTaskRef ECURefNum,
    char *ConversionName,
    f64 Factor,
    f64 Offset,
    char *Unit);
```

**Input**

- **ECURefNum**
  
  ECURefNum is the task reference that links to the selected ECU. This reference is originally returned from `mcECUCreate`.

- **ConversionName**
  
  ConversionName identifies the conversion object that handles measurement scaling. Use this name as a reference in `mcConversionCreate`.

- **Factor**
  
  Factor configures the scaling factor used to convert raw measurement data in the message to/from scaled floating-point units. The factor is the $A$ in the linear scaling formula $AX + B$, where $X$ is the raw data, and $B$ is the scaling offset.

- **Offset**
  
  Offset configures the scaling offset used to convert raw data in the measurement message to/from scaled floating-point units. The scaling offset is the $B$ in the linear scaling formula $AX + B$, where $X$ is the raw data, and $A$ is the scaling factor.

- **Unit**
  
  Configures the measurement channel unit string. You can use this value to display units (such as volts or RPM) along with the channel samples.

**Output**

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

Use `mcConversionCreate` to create a conversion object in memory instead of referring to measurement properties defined in the A2L database.
mcDAQClear

Purpose
Stops communication for the Measurement task and clears the task.

Format
```
mctypeStatus mcDAQClear(
    mctypeTaskRef *DAQRefNum);
```

Input
DAQRefNum
DAQRefNum is the task reference which links to the selected Measurement task. This reference is originally returned from mcDAQInitialize.

Output
Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcDAQClear must always be the final function called for a Measurement task. If you do not use mcDAQClear, the remaining Measurement task configuration can cause problems in the execution of subsequent applications. Because this function clears the Measurement task, the Measurement task reference is not given as an output but is transferred into an ECU reference task handle. To change properties of a running Measurement task, use mcDAQStartStop to stop the task, mc SetProperty to change the desired DAQ property, then mcDAQStartStop to restart the Measurement task.
mcDAQInitialize

**Purpose**
Initializes a Measurement task for the specified Measurement channel list.

**Format**
```
mcTypeStatus mcDAQInitialize(
    cstr MeasurementNames,
    mcTypeTaskRef ECURefNum,
    i32 DAQMode,
    u32 DTO_ID,
    f64 SampleRate,
    mcTypeTaskRef *DAQRefNum);
```

**Input**
- **MeasurementNames**
  Comma-separated list of Measurement names to initialize as a task. You can type in the channel list as a string constant or you can obtain the list from an A2L database file by using the `mcGetNames` function.
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **DAQMode**
  DAQMode specifies the I/O mode for the task. For an overview of the I/O modes, including figures, refer to the *Basic Programming Model* section of Chapter 4, *Using the ECU M&C API*.

**mcDAQModeDAQList**
Data is transmitted automatically by the ECU using DAQ lists. The data can be read back with the `mcDAQRead` as Single point data using sample rate = 0 or as waveform using a sample rate > 0. Input channel data is received from the DAQ messages.

**mcDAQModePolling**
In this mode the data from the Measurement task are uploaded from the ECU whenever `mcDAQRead` is called.
### mcDAQModeSTIMList

For XCP, this defines a DAQ list for data stimulation (STIM). Within a DAQ task initialized with this parameter, you can call `mcDAQWrite` to write stimulation data to the ECU. Calling `mcDAQRead` is not allowed. For CCP, an error is returned.

### mcDAQModeDAQListTimeStamped

The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with `mcDAQReadTimestamped` as a timestamped data array. Input channel data are received from the DAQ messages. Use `mcDAQReadTimestamped` to obtain input samples as an array of sample/timestamp pairs. Use this input mode to read samples with timestamps that indicate when each channel is received from the network.

- **DTO_ID**
  - DTO_ID is the CAN identifier for the Data Transmission Object (DTO) used to transmit the data from the DAQ lists to the host. The default value is -1 which means that the DTO ID used to transmit the DAQ list data is the same that is used for the rest of the CCP communication.

- **SampleRate**
  - SampleRate specifies the timing to use for samples of the (NI-CAN) task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

  For a DAQMode of `mcDAQModeDAQList`, SampleRate of zero means that `mcDAQRead` returns a single sample from the most recent messages received, and greater than zero means that `mcDAQRead` returns samples timed at the specified rate. For DAQMode of `mcDAQModePolling`, SampleRate is ignored.

### Output

- **DAQRefNum**
  - DAQRefNum is the reference handle for the Measurement task. Use this Measurement task reference in subsequent M&C DAQ functions for this task.

### Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

`mcDAQInitialize` does not start the transmission of the DAQ lists on the ECU. This enables you to use `mcSetProperty` to change the properties of a Measurement task. After you change properties, use `mcDAQStartStop` to start the transmission of the Measurement task.
mcDAQListInitialize

**Purpose**
Defines a DAQ list on a specific DAQ list number and initializes the Measurement task for the specified Measurement channel list. Initializes a Measurement task for the specified Measurement channel list.

**Format**
```c
mcTypeStatus mcDAQListInitialize(
    cstr MeasurementNames,
    mcTypeTaskRef ECURefNum,
    i16 DAQListNo,
    i32 DAQMode,
    u32 DTO_ID,
    f64 SampleRate,
    mcTypeTaskRef *DAQRefNum);
```

**Input**
- **MeasurementNames**
  Comma-separated list of Measurement names to initialize as a task. You can type in the channel list as a string constant or you can obtain the list from an A2L database file by using the `mcGetNames` function.
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **DAQListNo**
  DAQListNo specifies which DAQ list entry number should be used for the defined Measurement channel list for the selected ECU. To query the available amount of DAQ List numbers on the ECU use `mcPropECU_NumberOfDefinedDAQLists` with the function `mcGetProperty`. To query the defined DAQ list numbers use `mcPropECU_DAQListNumbers` with `mcGetProperty`.
- **DAQMode**
  DAQMode specifies the I/O mode for the task. For an overview of the I/O modes, including figures, refer to the Basic Programming Model section of Chapter 4, Using the ECU M&C API.

**mcDAQModeDAQList**
Data is transmitted automatically by the ECU using DAQ lists. The data can be read back with the `mcDAQRead` as Single point data using sample rate = 0 or as waveform using a sample rate > 0. Input channel data is received from the DAQ messages.
**mcDAQModePolling**

In this mode the data from the Measurement task are uploaded from the ECU whenever `mcDAQRead` is called.

**DTO_ID**

`DTO_ID` is the CAN identifier for the Data Transmission Object (DTO) used to transmit the data from the DAQ lists to the host. The default value is -1 which means that the DTO ID used to transmit the DAQ list data is the same that is used for the rest of the CCP communication.

**SampleRate**

`SampleRate` specifies the timing to use for samples of the (NI-CAN) task. The sample rate is specified in Hertz (samples per second). A sample rate of zero means to sample immediately.

For a `DAQMode` of `mcDAQModeDAQList`, `SampleRate` of zero means that `mcDAQRead` returns a single sample from the most recent messages received, and greater than zero means that `mcDAQRead` returns samples timed at the specified rate. For `DAQMode` of `mcDAQModePolling`, `SampleRate` is ignored.

**Output**

`DAQRefNum` is the reference handle for the Measurement task. Use this Measurement task reference in subsequent M&C DAQ functions for this task.

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

If an ECU offers a reduced and specific range of DAQ list entry numbers use the `mcDAQListInitialize` function to setup your Measurement list. `mcDAQListInitialize` does not start the transmission of the DAQ lists from the ECU to the application or vice versa through CCP or XCP. This enables you to use `mcSetProperty` to change the properties of a Measurement task. After you change properties use `mcDAQStartStop` to start the communication for the Measurement task. To query the available DAQ list entry numbers use `mcGetProperty` with the property `mcPropECU_DAQListNumbers`. 
mcDAQRead

**Purpose**
Reads samples from a Measurement task. Samples are obtained from received CAN messages.

**Format**
```
mcTypeStatus mcDAQRead(
    mcTypeTaskRef DAQRefNum,
    u32 NumberOfSamplesToRead,
    mcTypeTimestamp *StartTime,
    mcTypeTimestamp *DeltaTime,
    f64 *SampleArray,
    u32 *NumberOfSamplesReturned);
```

**Input**
- **DAQRefNum**
  DAQRefNum is the task reference from the previous Measurement task function. The task reference is originally returned from `mcDAQInitialize`, and then reused by subsequent Measurement task functions.
- **NumberOfSamplesToRead**
  Specifies the number of samples to read for the task. For single-sample input, pass 1 to this parameter.
  - If the initialized sample rate is zero, you must pass `NumberOfSamplesToRead` no greater than 1. `SampleRate` of zero means `mcDAQRead` immediately returns a single sample from the most recent message(s) received.

**Output**
- **StartTime**
  Returns the time of the first CAN sample in `SampleArray`. This parameter is optional. If you pass NULL for the `StartTime` parameter, the `mcDAQRead` function proceeds normally. If the initialized sample rate is greater than zero, the `StartTime` is determined by the sample timing. If the initialized `SampleRate` is zero, the `StartTime` is zero, because the most recent sample is returned regardless of timing.
  - `StartTime` uses the `mcTypeTimestamp` data type. The `mcTypeTimestamp` data type is a 64-bit unsigned integer compatible with the Microsoft Win32 FILETIME type. This absolute time is kept in a Coordinated Universal Time (UTC)
UTC time is loosely defined as the current date and time of day in Greenwich, England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Because mcTypeTimestamp is compatible with Win32 FILETIME, you can pass it into the Win32 FileTimeToLocalFileTime function to convert it to the local time zone, and then pass the resulting local time to the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. SYSTEMTIME is a struct with fields for year, month, day, and so on. For more information on Win32 time types and functions, refer to the Microsoft Win32 documentation.

DeltaTime

Returns the time between each sample in SampleArray. This parameter is optional. If you pass NULL for the DeltaTime parameter, the mcDAQRead function proceeds normally. If the initialized sample rate is greater than zero, the DeltaTime is determined by the sample timing. If the initialized sample rate is zero, the DeltaTime is zero, because the most recent sample is returned regardless of timing. DeltaTime uses the mcTypeTimestamp data type. The delta time is a relative 64-bit counter of 100 ns intervals, not an absolute UTC time. Nevertheless, you can use functions like the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. In addition, you can use the 32-bit LowPart of DeltaTime to obtain a simple 100 ns count, because values for SampleRate as slow as 0.4 Hz are still limited to a 32-bit 100 ns count.

SampleArray

Returns a 2D array, one array for each channel initialized in the task. The array of each channel must have NumberOfSamplesToRead entries allocated. The order of channel entries in SampleArray is the same as the order in the original ChannelList. If you need to determine the number of channels in the task after initialization, get the mcPropDAQ_NumChannels property for the task reference. If no message has been received since you started the task, 0 is returned as default value for of the channel in all entries of SampleArray.

NumberOfSamplesReturned

NumberOfSamplesReturned indicates the number of samples returned for each channel in SampleArray. The remaining entries are left unchanged (zero).
Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

If the initialized SampleRate is greater than zero, this function returns an array of samples, each of which indicates the value of the CAN channel at a specific point in time. The mcDAQRead function waits for these samples to arrive in time before returning. In other words, the SampleRate specifies a virtual clock that copies the most recent value from CAN messages for each sample time. The changes in sample values from message to message enable you to view the channel over time, such as for comparison with other CAN or DAQ input channels. To avoid internal waiting, you can use mcGetProperty to obtain nctPropSamplesPending property, and pass that as the NumberOfSamplesToRead parameter to mcDAQRead.

If the initialized SampleRate is zero, mcDAQRead immediately returns a single sample from the most recent message(s) received. For this single-point read, you must pass the NumberOfSamplesToRead parameter as 1. You can use the return value of mcDAQRead to determine whether a new message has been received since the previous call to mcDAQRead (or mcDAQStartStop). If no message has been received, the warning code CanWarnOldData is returned. If a new message has been received, the success code 0 is returned. If no message has been received since you started the task, the default value of the channel (nctPropChanDefaultValue) is returned in all entries of SampleArray.
mcDAQReadTimestamped

**Purpose**
Reads timestamped samples from a DAQ task initialized with the selected mode of mcDAQModeDAQListTimeStamped.

**Format**
```
mcTypeStatus mcDAQReadTimestamped(
    mcTypeTaskRef DAQRefNum,
    u32 NumberOfSamplesToRead,
    __int64 *TimestampArray,
    double *SampleArray,
    u32 *NumberOfSamplesReturned);
```

**Input**
- **DAQRefNum**
  DAQRefNum is the task reference that links to the selected measurement task. This reference is originally returned from mcDAQInitialize or mcDAQListInitialize.
- **NumberOfSamplesToRead**
  Specifies the number of samples to read for the task.

**Output**
- **TimestampArray**
  Returns the time at which each corresponding sample in SampleArray was received in a CAN message. The timestamps are returned as an array of arrays (2D array), one array for each channel initialized in the task. The array of each channel must have NumberOfSamplesToRead entries allocated. For example, if you call mcDAQInitialize with ChannelList of myDAQ1, myDAQ2, then call mcDAQReadTimestamped with NumberOfSamplesToRead of 20, both TimestampArray and SampleArray must be allocated as:

```c
__int64 mcTypeTimestamp TimestampArray[2][20];
double SampleArray[2][20];
```

The order of channel entries in TimestampArray is the same as the order in the original DAQ channel list. To determine the number of channels in the DAQ task after initialization, get the mcPropDAQ_NumChannels property for the DAQ task reference. Each timestamp in TimestampArray uses the __int64 data type compatible with the Microsoft Win32 FILETIME type. This absolute time is kept in a Coordinated Universal Time (UTC).
UTC time is loosely defined as the current date and time of day in Greenwich, England. Microsoft defines its UTC time (FILETIME) as a 64-bit counter of 100 ns intervals that have elapsed since 12:00 a.m., January 1, 1601. Because the timestamp is compatible with Win32 FILETIME, you can pass it into the Win32 FileTimeToLocalFileTime function to convert it to the local timezone, and then pass the resulting local time to the Win32 FileTimeToSystemTime function to convert to the Win32 SYSTEMTIME type. SYSTEMTIME is a struct with fields for year, month, day, and so on. For more information about Win32 time types and functions, refer to the Microsoft Win32 documentation.

**SampleArray**  
SampleArray returns the sample value(s) for each received CAN message. The samples are returned as an array of arrays (a 2D array), one array for each channel initialized in the DAQ task. The array of each channel must have NumberOfSamplesToRead entries allocated. You must allocate SampleArray exactly as TimestampArray, and the order of channel entries is the same for both.

**NumberOfSamplesReturned**  
Indicates the number of samples returned for each channel in SampleArray, and the number of timestamps returned for each channel in TimestampArray. The remaining entries are left unchanged (zero).

**Return Value**  
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**  
Each returned sample corresponds to a received CAN message for the measurement channels initialized in the DAQ channel list. For each sample, mcDAQReadTimestamped returns the sample value and a timestamp that indicates when the message was received. Because the timing of samples returned by mcDAQReadTimestamped is determined by when the message is received, the initialized sample rate is not used.

The function does not wait for messages, but instead returns samples from the messages received since the previous call to mcDAQReadTimestamped. The number of samples
returned is indicated in the `NumberOfSamplesReturned` output, up to a maximum of `NumberOfSamplesToRead` messages. If no new message has been received, `NumberOfSamplesReturned` is 0, and the return value indicates success.
mcDAQStartStop

**Purpose**

Starts the transmission of the DAQ lists assigned to the Measurement task on the ECU.

**Format**

```c
mcTypeStatus mcDAQStartStop(
    mcTypeTaskRef DAQRefNum,
    u32 StartStopMode);
```

**Input**

- **DAQRefNum**: DAQRefNum is the task reference from the previous Measurement task function. The task reference is originally returned from `mcDAQInitialize`, and then reused by subsequent Measurement task functions.
- **StartStopMode**: StartStopMode indicates the type of function to be performed:
  - **0—mcStartStopModeStop**
    Configures the ECU to stop transmitting a DAQ task. If stopped, properties of the DAQ task can be changed using `mcSetProperty`. This function is performed automatically before `mcDAQClear`.
  - **1—mcStartStopModeStart**
    Configures the ECU to start sending data for a Measurement task. Ensure that the DAQ list has not yet been transferred to the ECU first. Once started, properties of the DAQ list can no longer be changed using `mcSetProperty`. This function is performed automatically before the first read of the DAQ list with `mcDAQRead`.
  - **2—mcStartStopModeTransmitDAQ**
    Transfers the DAQ list to the ECU, but does not start it. For example, use this mode if you want to change the session status before starting the DAQ list. For some ECUs, this is necessary.
Output

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcDAQStartStop` is an optional command to start or stop communication for an M&C Measurement task. If you do not perform `mcDAQStartStop` (with the parameter `mcStartStopModeStart`) before using `mcDAQRead` the Measurement task is started by the first call of `mcDAQRead`. After you start the transmission of the DAQ lists, you can no longer change the configuration of the Measurement task with `mcSetProperty`. You must call `mcDAQStartStop` (with the parameter `mcStartStopModeStop`) first.
mcDAQWrite

Purpose

Writes samples to an ECU DAQ list.

Format

\[
\text{mcTypeStatus mcDAQWrite(}
\begin{align*}
&\text{mcTypeTaskRef DAQRefNum}, \\
&\text{u32 NumberOfSamplesToWrite}, \\
&\text{f64 *SampleArray});
\end{align*}
\]

Input

- **DAQRefNum**
  
  DAQRefNum is the task reference from the previous Measurement task function. The task reference is originally returned from mcDAQInitialize, and then reused by subsequent Measurement task functions.

- **NumberOfSamplesToWrite**
  
  NumberOfSamplesToWrite specifies the number of samples to write for the ECU MC DAQ task to the ECU DAQ list. For single-sample output, pass 1 to this parameter. The initialized DAQ sample rate must be set to zero. A SampleRate of zero means mcDAQWrite immediately writes a single sample to the ECU when calling the mcDAQWrite function. You must pass NumberOfSamplesToWrite no greater than 1.

- **SampleArray**
  
  SampleArray specifies a 2D array, one array for each channel initialized in the task. The array of each channel must have NumberOfSamplesToWrite entries allocated. The order of channel entries in SampleArray is the same as the order in the original ChannelList. If you must determine the number of channels in the task after initialization, get the mcPropDAQ_NumChannels property for the task reference.

Output

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.
Description

For XCP STIM lists (refer to mcDAQInitialize), mcDAQWrite transfers an array of samples to the ECU. These samples are called data stimulation packets (STIM). On the ECU side the STIM processor buffers incoming data stimulation packets. When an event occurs which triggers a DAQ list in data stimulation mode, the buffered data is transferred to the memory on the slave device.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to configure data stimulation.
mcDatabaseClose

Purpose
Closes a specified A2L Database.

Format
```c
mcTypeStatus mcDatabaseClose(
    mcTypeTaskRef *DBRefNum);
```

Input
`DBRefNum` is the task reference from the initial database task function. The database task reference is originally returned from `mcDatabaseOpen`.

Output

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcDatabaseClose` must always be the final ECU M&C function called for each database task. If you do not use the `mcDatabaseClose` function, the remaining task configurations can cause problems in the execution of subsequent Measurement and Calibration applications.
mcDatabaseOpen

Purpose
Opens a specified A2L Database.

Format
```
mcTypeStatus mcDatabaseOpen(
    cstr Database,
    mcTypeTaskRef *DBRefNum);
```

Input
Database Database is a path to an A2L database file from which to get Measurement or calibration channel names. The file must use a .A2L extension. You can generate A2L database files with several 3rd party tools.

Output
DBRefNum DBRefNum is the task reference from the initial database task function. The database task reference is originally returned from mcDatabaseOpen.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
The mcDatabaseOpen function does not start communication. This enables you to query all defined ECU names in the A2L Database using the mcGetNames function and selecting the property value ECU Names.

To use the ECU M&C Toolkit on a LabVIEW RT system, you must download your ASAM MCD 2MC database (*.A2L) file to the RT target.
mcDatabaseOpenEx

Purpose

Creates a specified A2L database by a name in memory.

Format

mcTypeStatus mcDatabaseOpenEx(
  cstr DatabaseName,
  mcTypeTaskRef *DBRefNum);

Input

DatabaseName

DatabaseName is a database name associated with the database created in memory. Use the string syntax :<myname>: for the A2L database if using multiple databases in memory. (For example, if using two databases in memory, use :MyDatabase1: as DatabaseName for the first database, and :MyDatabase2: for the second DatabaseName created in memory.)

Output

DBRefNum

DBRefNum is the task reference from the initial database task function. The database task reference is originally returned from mcDatabaseOpen.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcDatabaseOpenEx does not start communication. Use it to create all needed objects in memory. After creating an A2L database in memory, you typically create an ECU object using mcECUCreate, a scaling object using mcConversionCreate, a measurement object using mcMeasurementCreate, and an event using mcEventCreate.

Note

mcDatabaseOpenEx does not support creating objects to access characteristics. To access a characteristic, assign a valid A2L database file with defined characteristics.
**mcDoubleToText**

**Purpose**
Converts a numerical value to a text string using an enumeration or range text type scaling.

**Format**

```c
mcTypeStatus mcDoubleToText(
    mcTypeTaskRef ECURefNum,
    u32 ObjectType,
    cstr ObjectName,
    double Value,
    u32 SizeOfTextValue,
    str TextValue);
```

**Input**

- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

- **ObjectType**
  Indicates the type of the object named in `ObjectName`. Valid values are:
  
  1. Measurement Name
  2. Characteristic Name

- **ObjectName**
  Indicates the object (measurement or characteristic) for which the COMPU_VTAB scaling is performed. If no COMPU_VTAB scaling is available for the object, `TextValue` is just a string representation of the value specified in `Value`.

- **Value**
  The numerical value to be converted. For example, this could have been returned from `mcCharacteristicRead` or `mcMeasurementRead`.

- **SizeOfTextValue**
  Must contain the number of bytes in the buffer passed to `TextValue`. Note that there is no way of requesting the necessary size of this buffer. If you do not know up front how long your text could become, specify a buffer of 256 bytes. This is the maximum the ASAM standard allows.
Output

TextValue The buffer for the resulting converted text string. If the Value specified is listed in a COMPU_VTAB scaling for the characteristic or measurement specified in ObjectName, the respective text is returned. If no such value is available, a string representation of the double value is returned.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcDoubleToText performs text conversion for measurement or characteristic values. Especially if the measurement or characteristic has an associated COMPU_VTAB type scaling, the textual representation of the value is returned. If no such value is present, either because the object does not have a text scaling or the value does not have a textual representation in the table, a string representation of the double value is returned.
**mcDownload**

**Purpose**
Downloads data to an ECU.

**Format**
```c
mcTypeStatus mcDownload(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u32 BlockSize,
    u8 *Data);
```

**Input**
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Address**
  Configures the target address for the download operation in the ECU. `mcAddress` is a C struct consisting of:
  - **Address**
    Specifies the address part of the target address.
  - **Extension**
    Extension contains the extension part of the target address.
- **BlockSize**
  BlockSize determines the size of the data block to be downloaded.

**Output**
- **Data**
  Data pointer to the information to be downloaded.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
**Description**

`mcDownload` downloads data to an ECU. The data is stored starting at the selected **Address** and **Extension** in the ECU memory. The function can download more than 5 data bytes to the ECU.

If you are using the CCP protocol and the selected **BlockSize** is higher than 5 bytes, `mcDownload` performs several CCP DNLOAD commands until all data bytes are downloaded to the ECU. `mcDownload` implements the CCP DNLOAD command defined by the CCP specification.

If you are using the XCP protocol, the **Data** block of the specified **BlockSize** is copied into the ECU memory, starting at the MTA. The MTA is post-incremented by the number of downloaded data bytes. If the slave device does not support Block Transfer Mode, all downloaded data is transferred in a single command packet. If Block Transfer Mode is supported, the downloaded data is transferred in multiple command packets. For the slave however, there might be limitations concerning the maximum number of consecutive command packets, so the number of data elements may be within a limited range. The master device has two additional consecutive DOWNLOAD_NEXT command packets. The slave device will acknowledge only the last DOWNLOAD_NEXT command packet. The separation time between the command packets and the maximum number of packets are specified in the response for the CONNECT command.
mcECUConnect

**Purpose**
Establishes communication to the selected ECU through CCP or XCP. After a successful ECU Connect you can create a Measurement task or read/write a Characteristic.

**Format**
```
mcTypeStatus mcECUConnect(
    mcTypeTaskRef ECURefNum);
```

**Input**
- **ECURefNum**

  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

**Output**

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**
mcECUConnect implements the CCP or XCP CONNECT command. It establishes a logical connection to an ECU, using the provided ECU Reference handle ECURefNum. Unless a slave device (ECU) is unconnected, it must not execute or respond to any command sent by the application. The only exception to this rule is the Test command, to which the CCP or XCP slave with the specific address may return an acknowledgement. Only a single CCP or XCP slave can be connected to the application at a time.
mcECUCreate

Purpose
Creates an ECU object in memory.

Format
```
mcTypeStatus mcECUCreate(
    mcTypeTaskRef DBRefNum,
    cstr ECUName,
    char *Interface,
    i32 ByteOrder,
    u32 CRO_ID,
    u32 DTO_ID,
    u16 StationAddress,
    u32 BaudRate,
    mcTypeTaskRef *ECURefNum);
```

Input
- **DBRefNum**
  - DBRefNum is the task reference from the initial database task function. The database task reference is originally returned from `mcDatabaseOpen`.

- **ECUname**
  - Identifies the ECU object. Use this name as reference in `mcMeasurementCreate` to create a DAQ list on the ECU.

- **Interface**
  - Specifies the protocol and optional interface to use for this task.

- **ByteOrder**
  - Sets the byte order of the CCP slave device:
    - **0—MSB_LAST**
      - The CCP Slave device uses the MSB_LAST (Intel) byte ordering.
    - **1—MSB_FIRST**
      - The CCP Slave device uses the MSB_FIRST (Motorola) byte ordering.

- **CRO_ID**
  - Sets the Command Receive Object (CRO) CAN Identifier for CCP, or XCP on CAN, which is used to send commands and data from the host to the slave device.

- **DTO_ID**
  - Sets the Data Transfer Object (DTO) CAN Identifier for CCP, or XCP on CAN, which is used to send commands and data from the slave device to the host.

- **StationAddress**
  - Sets the slave device station address. CCP is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a station address that must be unique for all ECUs sharing the
same CAN Arbitration IDs. Unless an ECU has been addressed by its station address, the ECU must not react to CCP commands sent by the CCP master.

**BaudRate**
Sets the CAN baud rate in use by the selected interface. This property applies to all tasks initialized with the NI-CAN or NI-XNET interface. You can specify the following basic baud rates as the numeric rate: 33333, 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000. You can specify advanced baud rates as 8000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1). For more information, refer to the Interface Properties dialog in MAX.

**Output**

**ECURefNum**
ECURefNum is the task reference that links to the selected ECU.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**
The function `mcECUCreate` is used to create an ECU object in memory instead of referring to a predefined ECU of an A2L database.

**Interface** is the name of the protocol and interface the selected ECU task will use. This string uses the syntax `XXX:YYY`, where X defines the selected protocol. The following strings may be used as Y:

- String CCP refers using the CAN Calibration Protocol (CCP)
- String XCP refers using the Universal Measurement and Calibration Protocol (XCP)

**Using NI-CAN**
If you are using the CCP protocol with NI-CAN hardware, `YYY` can be associated with a defined NI-CAN interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX). For example, if you are using the CCP protocol on NI-CAN interface CAN1, the value passed to `Interface` is `CCP:CAN1`. The special string values “CAN256” and
“CAN257” refer to virtual interfaces. Refer to the *NI-CAN Hardware and Software User Manual* for detailed information on how to use virtual NI-CAN ports.

If you are using the XCP protocol, YYY can be associated with a XCP transport layer. The transport layers may defined as follows:

- CANxx
- TCP
- UDP

If you select CAN as the transport layer you must specify the NI-CAN interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX). For example, if you are using the XCP on NI-CAN interface CAN2 the value passed to Interface is XCP:CAN2. If you are using the XCP on UDP the value passed to Interface is XCP:UDP. If you are using the XCP on TCP the value passed to Interface is XCP:TCP. The special string values “CAN256” and “CAN257” refer to virtual interfaces. Refer to the *NI-CAN Hardware and Software User Manual* for detailed information on how to use virtual NI-CAN ports.

**Using NI-XNET**

If you are using NI-XNET hardware and select the xxx:yyy syntax, the ECU M&C Toolkit uses the XNET NI-CAN compatibility library (XCL) internally if the XNET interface is defined in MAX under **NI-CAN Devices**. To force use of the native XNET API, you must define a unique interface name that differs from the NI-CAN-compatible interface name under **XNET Devices**, or use the xxx:yyy@nixnet syntax. The interface name is related to the NI-XNET hardware naming under **Devices and Interfaces** in MAX. The extension nixnet directs the ECU M&C Toolkit to use the native NI-XNET API.

**Note** By selecting nixnet as **Protocol and Interface** string, the ECU Measurement and Calibration Toolkit uses the Frame Input and Output Queued sessions. To force the ECU Measurement and Calibration Toolkit to use Frame Input and Output Stream sessions instead, select ni_genie_nixnet as **Protocol and Interface** string (for example, CCP:CAN1@ni_genie_nixnet). An application instance can use only one Frame Input Stream Session and one Frame Output Stream Session at a time, so use the default name nixnet as **Protocol and Interface** string, so that multiple NI-XNET Frame Queued Sessions can coexist on a single interface, and the Frame Input and Output Stream Sessions may be used, for example, for a Frame logging/replay use case.

**CompactRIO or R Series**

If you are using CompactRIO or R Series hardware, you must provide a bitfile that handles the CAN communication between the host system and the FPGA. To access the CAN module on the FPGA, you must specify the bitfile name after the @ (for example, CCP:CAN1@MyBitfile.lvbitx). To specify a special RIO target, you can specify that target by
its name followed by the bitfile name (for example, \texttt{XCP\_CAN1@RIO1.MyBitfile.lvbitx}). Currently, only a single CAN interface is supported. RIO1 defines the RIO target name as defined in your LabVIEW Project definition. The \texttt{lvbitx} filename represents the filename and location of the bitfile on the host. You may use just the filename without the folder if the bitfile is in the same folder as the LabVIEW Project (\texttt{*lvproj}).
mcECUDeselect

Purpose
Deselects an ECU and invalidates the ECU reference handle.

Format
mcTypeStatus mcECUDeselect(
    mcTypeTaskRef *ECURefNum);

Input
ECURefNum
ECURefNum is the task reference which links to the selected ECU.
This reference is originally returned from mcECUSelectEx.

Output
Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means
the function executed successfully. A negative value specifies an error, which means the
function did not perform the expected behavior. A positive value specifies a warning, which
means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string
for the return value.

Description
mcECUDeselect deselects the ECU and clears all internal driver data stored for this ECU.
After calling this function it is no longer possible to communicate with the specified ECU.
The task reference ECURefNum is transferred into a database handle DBRefNum.
mcECUDisconnect

Purpose
Disconnects CCP or XCP communication to the selected ECU.

Format
```c
mcTypeStatus mcECUDisconnect(
    mcTypeTaskRef ECURefNum);
```

Input
ECURefNum ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.

Output
Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcECUDisconnect implements the CCP or XCP command DISCONNECT. mcECUDisconnect disconnects the specified CCP or XCP slave from the actual communication and ends the calibration session. When the calibration session is terminated, all CCP or XCP DAQ lists of the device are stopped and cleared and the protection masks of the device are set to their default values.

mcECUDisconnect is an optional command as disconnecting from the ECU is performed by the function mcECUDeselect.
mcECUSelectEx

Purpose
Selects an ECU from the names stored in an A2L database.

Format
```c
mcTypeStatus mcECUSelectEx(
    mcTypeTaskRef DBRefNum,
    cstr ECUName,
    cstr Interface,
    mcTypeTaskRef *ECURefNum);
```

Input
- **DBRefNum**: DBRefNum is the task reference from the initial database task function. The database task reference is originally returned from mcDatabaseOpen.
- **ECUName**: ECUName is the selected ECU name out of an A2L Database file with which to initialize all subsequent tasks.
- **Interface**: Specifies the protocol and optional interface to use for this task.

Output
- **ECURefNum**: ECURefNum is the task reference which links to the selected ECU.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcECUSelectEx creates an ECU reference handle to the selected ECU name. The mcECUSelectEx function does not start communication. This enables you to use mcSetProperty to change the properties of an ECU task. After you change properties use mcECUConnect to start communication for the task and logically connect to the selected ECU.
**Interface** is the name of the protocol and interface the selected ECU task will use. This string uses the syntax XXX:YYY, where X defines the selected protocol. The following strings may be used:

- **String CCP** refers using the CAN Calibration Protocol (CCP)
- **String XCP** refers using the Universal Measurement and Calibration Protocol (XCP)

If you are using the CCP protocol, **YYY** can be associated with a defined NI-CAN interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX). For example, if you are using the CCP protocol on NI-CAN interface CAN1, the value passed to **Interface** is **CCP:CAN1**. The special string values “CAN256” and “CAN257” refer to virtual interfaces. Refer to the [NI-CAN Hardware and Software User Manual](#) for detailed information on how to use virtual NI-CAN ports.

If you are using the XCP protocol, **YYY** can be associated with a XCP transport layer. The transport layers may defined as follows:

- **CAN**
- **TCP**
- **UDP**

**Using NI-CAN**

If you select CAN as the transport layer you must specify the NI-CAN interface (CAN0, CAN1, up to CAN63). CAN network interface names are associated with physical CAN ports using the Measurement and Automation Explorer (MAX). For example, if you are using the XCP on NI-CAN interface CAN2 the value passed to **Interface** is **XCP:CAN2**. If you are using the XCP on UDP the value passed to **Interface** is **XCP:UDP**. If you are using the XCP on TCP the value passed to **Interface** is **XCP:TCP**. The special string values “CAN256” and “CAN257” refer to virtual interfaces. Refer to the [NI-CAN Hardware and Software User Manual](#) for detailed information on how to use virtual NI-CAN ports.

**Using NI-XNET**

If you are using NI-XNET hardware and select the **xxx:yyy** syntax, the ECU M&C Toolkit uses the XNET NI-CAN compatibility library (XCL) internally if the XNET interface is defined in MAX under **NI-CAN Devices**. To force use of the native XNET API, you must define a unique interface name that differs from the NI-CAN-compatible interface name under **XNET Devices**, or use the **xxx:yyy@nixnet** syntax. The interface name is related to the NI-XNET hardware naming under **Devices and Interfaces** in MAX. The extension **nixnet** directs the ECU M&C Toolkit to use the native NI-XNET API.

**Note**  By selecting **nixnet** as **Protocol and Interface** string, the ECU Measurement and Calibration Toolkit uses the Frame Input and Output Queued sessions. To force the ECU
Measurement and Calibration Toolkit to use Frame Input and Output Stream sessions instead, select `ni_genie_nixnet` as **Protocol and Interface** string (for example, `CCP:CAN1@ni_genie_nixnet`). An application instance can use only one Frame Input Stream Session and one Frame Output Stream Session at a time, so use the default name `nixnet` as **Protocol and Interface** string, so that multiple NI-XNET Frame Queued Sessions can coexist on a single interface, and the Frame Input and Output Stream Sessions may be used, for example, for a Frame logging/replay use case.

**CompactRIO or R Series**

If you are using CompactRIO or R Series hardware, you must provide a bitfile that handles the CAN communication between the host system and the FPGA. To access the CAN module on the FPGA, you must specify the bitfile name after the `@` (for example, `CCP:CAN1@MyBitfile.lvbitx`). To specify a special RIO target, you can specify that target by its name followed by the bitfile name (for example, `XCP:CAN1@RIO1,MyBitfile.lvbitx`). Currently, only a single CAN interface is supported. RIO1 defines the RIO target name as defined in your LabVIEW Project definition. The `lvbitx` filename represents the filename and location of the bitfile on the host. You may use just the filename without the folder if the bitfile is in the same folder as the LabVIEW Project (`*.lvproj`).
mcEventCreate

**Purpose**

Creates an Event object in memory.

**Format**

```c
mcTypeStatus mcEventCreate(
    mcTypeTaskRef ECURefNum,
    cstr EventChannelName,
    u8 EventChannelNumber);
```

**Input**

- **ECURefNum**
  - `ECURefNum` is the task reference that links to the selected ECU. This reference is originally returned from `mcECUCreate`.
- **EventChannelName**
  - `EventChannelName` identifies the Event Channel object.
- **EventChannelNumber**
  - `EventChannelNumber` identifies the number of the Event Channel. The event channel number specifies the generic signal source that effectively determines the data transmission timing. To allow a reduction of the desired transmission rate, a prescaler may be applied to the Event Channel. The prescaler value factor must be greater than or equal to 1 to use `mcSetProperty` using `mcPropDAQ_Prescaler`.

**Output**

**Return Value**

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**

Use the function `mcEventCreate` to create an Event object in memory instead of referring to a predefined Event Channel in the A2L database. Assign the event channel object by name to a DAQ List in `mcMeasurementCreate`. 
mcGeneric

Purpose
Sends a generic command.

Format
```
mcTypeStatus mcGeneric(
    mcTypeTaskRef ECURefNum,
    u8 Command,
    u8 *Data,
    u32 DataSize,
    u32 Timeout,
    u8 *ErrorCode,
    u8 *ReturnValue,
    u32 *ReturnValueSize);
```

Input
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Command**
  Command is the CCP command code to send to the ECU.
- **Data**
  Data contains the parameters of the command as an array of bytes. For more information about the parameters of the (user defined) commands implemented in the ECU, refer to the documentation for the ECU.
- **DataSize**
  DataSize defines the number of bytes (the array size) passed in the input parameter Data.
- **Timeout**
  Timeout specifies the maximum number of milliseconds to wait for a response from the ECU. If the Timeout expires before an ECU response occurs, the error `mcErrorTimeout` is returned.

Output
- **ErrorCode**
  ErrorCode describes the error returned from the ECU during the communication.
- **ReturnValue**
 ReturnValue may contain an array of bytes returned from the ECU as a response to the command sent to the ECU.
- **ReturnValueSize**
  ReturnValueSize contains the number of bytes returned from the ECU passed to ReturnValue.
Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description

`mcGeneric` can be used to send commands to the ECU that are not defined by the CCP or XCP specification. The command code in `Command` and the parameters of this command defined in `Data` are sent to the ECU, and the data returned by the ECU is passed to the parameter `ReturnValue`. Since the ECU M&C driver has no knowledge of the parameters of the command and their data types, all parameters and return values are passed as an array of bytes. Therefore you are responsible for the correct type casting of all parameters and return values of this command. Make sure that all parameters are passed in the correct byte ordering for this function. For more information about the (user defined) commands and their parameters refer to the documentation for the ECU.
mcGetNames

Purpose
Retrieves a comma-separated list of ECU names, Measurement names, Characteristic names, Event names, Characteristic pages, or Group names from a specified A2L database.

Format
```c
mcTypeStatus mcGetNames(
    mcTypeTaskRef RefNum,
    u32 Type,
    cstr ECUName,
    u32 SizeOfNamesList,
    str NameList);
```

Input
- **RefNum**: RefNum is any ECU M&C task reference which consists of a valid link to the opened A2L database (DBRefNum), a selected ECU (ECURefNum) or a Measurement task (DAQRefNum). RefNum must be valid for the related Type.
- **Type**: Specifies the Type of names to return.
  - 0—mcTypeECUNames
    Returns a list of ECU names. You can pass one of the returned names to mcECUSelectEx.
  - 1—mcTypeMeasurementNames
    Returns a list of Measurement names. You can pass the returned NamesList to mcDAQInitialize.
  - 2—mcTypeCharacteristicNames
    Returns a list of Characteristic names. You can pass a single name out of the NamesList to mcCharacteristicWrite or mcCharacteristicRead.
  - 3—mcTypeEventChannelNames
    Returns a list of Event Channel names.
  - 4—mcTypeDefinedPagesNames
    Returns a list of Calibration page names.
  - 5—mcTypeGroupNames
    Returns a list of Group names.
6—mcTypeGroup_SubGroupNames
   Returns a list of Subgroup names of the specified Group name.

7—mcTypeGroup_MeasurementNames
   Returns a list of Measurement names within the specified Group.

8—mcTypeGroup_CharacteristicNames
   Returns a list of Characteristic names within the specified Group.

9—mcTypeFuncNames
   Returns a list of Function names within the specified ECU.

10—mcTypeFunc_DefCharacNames
    Returns a list of Characteristic names referred by the DEF_CHARACTERISTIC keyword within the related Function.

11—mcTypeFunc_RefCharacNames
    Returns a list of Characteristic names referred by the REF_CHARACTERISTIC keyword within the related Function.

12—mcTypeFunc_InMeasNames
    Returns a list of Measurement names referred by the IN_MEASUREMENT keyword within the related Function.

13—mcTypeFunc_OutMeasNames
    Returns a list of Measurement names referred by the OUT_MEASUREMENT keyword within the related Function.

14—mcTypeFunc_LocMeasNames
    Returns a list of Measurement names referred by the LOC_MEASUREMENT keyword within the related Function.

15—mcTypeFunc_SubFuncNames
    Returns a list of Function names referred by the SUB_FUNCTION keyword within the related Function.

16—mcTypeGroup_FunctionListNames
    Returns a list of Function names referred by the FUNCTION_LIST keyword within the related Group.
ECUName

If the Type = `mcTypeMeasurementNames` or `mcTypeCharacteristicNames` and RefNum contains a DBRefNum, the corresponding ECU name must be referenced in order to access ECU specific properties. If RefNum contains an ECURefNum or DAQRefNum the parameter ECUName is ignored and can be set to NULL.

SizeOfNamesList

Size of the buffer provided to take the names list. After calling `mcGetNamesLength`, you can allocate an array of size SizeofNamesList, and then pass that array to `mcGetNames` using the same input parameters. This ensures that `mcGetNames` will return all names without error.

Output

NameList

Returns the comma-separated list of names specified by Type.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description

Get a comma-separated list of ECU, Measurement, Characteristic, or Event Channel names from a specified A2L database file.

If using `mcGetNames` to query the list of supported event channels on an ECU, the event channels might be stored inside the ECU instead of the A2L file. To query these event channel names from the ECU directly, connect to the ECU using `mcECUConnect` before calling `mcGetNames`.
mcGetNamesLength

Purpose
Retrieves the amount of memory required to store the names returned by mcGetNames.

Format
```c
mcTypeStatus mcGetNamesLength(
    mcTypeTaskRef RefNum,
    u32 Type,
    cstr ECUName,
    u32 *SizeOfNamesList);
```

Input
- **RefNum**: is any ECU M&C task reference which consists of a valid link to the opened A2L database (DBRefNum), a selected ECU (ECURefNum) or a Measurement task (DAQRefNum). RefNum must be valid for the related Type.
- **Type**: Specifies the Type of names to return.
  - 0—`mcTypeECUNames` Returns a list of ECU names.
  - 1—`mcTypeMeasurementNames` Returns a list of Measurement names.
  - 2—`mcTypeCharacteristicNames` Returns a list of Characteristic names.
  - 3—`mcTypeEventChannelNames` Returns a list of Event Channel names.
  - 4—`mcTypeDefinedPagesNames` Returns a list of Calibration page names.
  - 5—`mcTypeGroupNames` Returns a list of Group names.
  - 6—`mcTypeGroup_SubGroupNames` Returns a list of Subgroup names of the specified Group name.
  - 7—`mcTypeGroup_MeasurementNames` Returns a list of Measurement names within the specified Group.
8—**mcTypeGroup_CharacteristicNames**

   Returns a list of Characteristic names within the specified Group.

9—**mcTypeFuncNames**

   Returns a list of Function names within the specified ECU.

10—**mcTypeFunc_DefCharacNames**

   Returns a list of Characteristic names referred by the DEF_CHARACTERISTIC keyword within the related Function.

11—**mcTypeFunc_RefCharacNames**

   Returns a list of Characteristic names referred by the REF_CHARACTERISTIC keyword within the related Function.

12—**mcTypeFunc_InMeasNames**

   Returns a list of Measurement names referred by the IN_MEASUREMENT keyword within the related Function.

13—**mcTypeFunc_OutMeasNames**

   Returns a list of Measurement names referred by the OUT_MEASUREMENT keyword within the related Function.

14—**mcTypeFunc_LocMeasNames**

   Returns a list of Measurement names referred by the LOC_MEASUREMENT keyword within the related Function.

15—**mcTypeFunc_SubFuncNames**

   Returns a list of Function names referred by the SUB_FUNCTION keyword within the related Function.

16—**mcTypeGroup_FunctionListNames**

   Returns a list of Function names referred by the FUNCTION_LIST keyword within the related Group.

---

**ECUName**

If the Type = **mcTypeMeasurementNames** or Type = **mcTypeCharacteristicNames** and RefNum contains a **DBRefNum**, the corresponding ECU name must be referenced in order to access ECU specific properties. If RefNum contains an **ECURefNum** or **DAQRefNum** the parameter ECUName is ignored and can be set to NULL.
Output

SizeOfNamesList  Number of bytes required for mcGetNames to return all names for the specified ECUName and Type. After calling mcGetNamesLength, you can allocate an array of size SizeOfNamesList, then pass that array to mcGetNames using the same input parameters. This ensures that mcGetNames will return all names without error.

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

After calling mcGetNamesLength, you can allocate an array of size SizeOfNamesList, then pass that array to mcGetNames using the same input parameters. This ensures that mcGetNames will return all names without error.

If using mcGetNamesLength to query the length of the list of supported event channels on an ECU, the event channels might be stored inside the ECU instead of the A2L file. To query these event channel names from the ECU directly, connect to the ECU using mcECUConnect before calling mcGetNamesLength.
mcGetProperty

**Purpose**
Retrieves a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.

**Format**
```c
mcTypeStatus mcGetProperty(
    mcTypeTaskRef RefNum,
    cstr Name,
    u32 PropertyID,
    u32 SizeOfValue,
    void *Value);
```

**Input**
- **RefNum**
  RefNum is any ECU M&C task reference which consists of a valid link to the opened A2L database (DBRefNum), a selected ECU (ECURefNum) or a Measurement task (DAQRefNum). RefNum must be valid for the related PropertyID type.
- **Name**
  Specifies an individual name (ECU name, Measurement channel name, or Characteristic name) within the task.
- **PropertyID**
  Selects the property to get.
  For a description of each property, including its data type and PropertyId, refer to the Properties section.
- **SizeOfValue**
  Number of bytes allocated for the Value output. This size normally depends on the data type listed in the description of the property.

**Output**
- **Value**
  Returns the property value. PropertyId determines the data type of the returned value.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.
### Properties

**Table 6-4. Values for PropertyID**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropCANBaudRate</td>
<td>Returns the CAN Baud rate for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropCANTermination</td>
<td>For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_Address</td>
<td>Returns the address of the selected Characteristic in the memory of the ECU.</td>
</tr>
</tbody>
</table>
| u32       | mcPropChar_ByteOrder        | Returns the specified byte order:  
0—**Intel format**  
Bytes are in little-endian order, with least-significant bit first.  
1—**Motorola format**  
Bytes are in big-endian order, with most-significant bit first. |
| u8        | mcPropChar_Datatype         | Returns the data type of the Characteristic.                                                                                                 |
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropChar_Dimension</td>
<td>Returns the dimension of the Characteristic: 0—0-dimensional: The Characteristic can be accessed (read/write) through a double value. 1—1-dimensional: The Characteristic can be accessed (read/write) through a one-dimensional array of double value. 2—2-dimensional: The Characteristic can be accessed (read/write) through a two-dimensional array of double value.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropChar_Extension</td>
<td>Returns additional address information. For instance it can be used to distinguish different address spaces of an ECU (multi-microcontroller devices).</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropChar_Maximum</td>
<td>Returns the Maximum value of the Characteristic.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropChar_Minimum</td>
<td>Returns the Minimum value of the Characteristic.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_ReadOnly</td>
<td>Returns if a Characteristic is set to read only. In this case it is not allowed to call <code>mcCharacteristicWrite</code> for this Characteristic.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_Sizes</td>
<td>Returns the Array Sizes for the X and Y directions of the Characteristic.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropChar_Unit</td>
<td>Returns the unit string defined for this Characteristic in the A2L database.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_Unit_Size</td>
<td>Returns the number of bytes to be allocated if you call <code>mcGetProperty</code> with the parameter <code>mcPropChar_Unit</code>.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropChar_X_Axis</td>
<td>Returns X-axis values on which the Characteristic is defined. Valid if the selected Characteristic is 1- or 2-dimensional.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropChar_Y_Axis</td>
<td>Returns Y-axis values on which the Characteristic is defined. Valid if the selected Characteristic is 2-dimensional.</td>
</tr>
</tbody>
</table>
### Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>f64</td>
<td>mcPropChar_Scale_Factor</td>
<td>Returns the scaling factor defined for this Characteristic in the A2L database.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropChar_Scale_Offset</td>
<td>Returns the scaling offset defined for this Characteristic in the A2L database.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_Scale_Type</td>
<td>Returns the scaling type defined for this Characteristic in the A2L database.</td>
</tr>
</tbody>
</table>

- **0: Unknown**
  The type of the scaling could not be derived from the A2L file content.

- **1: Rational Function**
  The related scaling is based on a rational function of second order. This covers also the linear scaling, given by factor and offset.
### Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| 2: Enumeration Text |      | The related scaling is based on the COMPU_VTAB keyword within the A2L file.  
  - Read functions return nonscaled, numeric values.  
  - Write functions accept nonscaled, numeric values.  
  - It is possible to use `mcDoubleToText` and `mcTextToDouble` to convert between enumeration text values and double values. |
| 3: Range Text |      | The related scaling is based on the COMPU_VTAB_RANGE keyword within the A2L file.  
  - Read functions return nonscaled, numeric values.  
  - Write functions accept nonscaled, numeric values.  
  - It is possible to use `mcDoubleToText` and `mcTextToDouble` to convert between range text values and double values. |
| 4: Formula |      | The related scaling is based on the FORMULA keyword within the A2L file, using a free formula to calculate the values. |
| 5: Table (Using Interpolation) |      | The related scaling is based on the TAB_INTP keyword within the A2L file, using interpolation between x-y pairs. |
| 6: Table (Without Interpolation) |      | The related scaling is based on the TAB_NOINTP keyword within the A2L file, using x-y pairs without interpolation. |
### Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropChar_Scale_TextValues_Size</td>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the length needed to store the comma-separated list of text values that can be converted into raw values (refer to the mcPropChar_Scale_TextValues property).</td>
</tr>
<tr>
<td>str</td>
<td>mcPropChar_Scale_TextValues</td>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the comma-separated list of text values that can be converted into raw values.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropCmd_EXCHANGE_ID</td>
<td>Returns whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropCROID</td>
<td>Returns the CRO CAN Identifier (Command Receive Object) for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQ_DTO_ID</td>
<td>Returns the DTO ID (Data Transmission Object) which is used by the ECU to respond to send data from the DAQ lists to the CCP master.</td>
</tr>
<tr>
<td>nctType Taskref</td>
<td>mcPropDAQ_DTO_Task</td>
<td>NI-CAN task reference to the CAN Task assigned to the DTO ID of the Measurement task.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQ_EventChannelName</td>
<td>Returns the selected event channel name to which the Measurement task is assigned.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQ_EventChannelName_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropDAQ_EventChannelName.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQ_Mode</td>
<td>Returns the selected mode of an M&amp;C Measurement task.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>0—DAQ List</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with mcDAQRead as Single point data using sample rate = 0, or as waveform using a sample rate &gt; 0. Input channel data is received from the DAQ messages. Use mcDAQRead to obtain input samples as single-point, array, or waveform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>1—Polling</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In this mode the data from the Measurement task is uploaded from the ECU whenever mcDAQRead is called.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQ_NumChannels</td>
<td>Returns the number of channels initialized in a DAQ channel list of a M&amp;C Measurement task. This is the number of array entries required when using mcDAQRead.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropDAQ_Prescaler</td>
<td>Prescaler for the Measurement task on the ECU.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropDAQ_SampleRate</td>
<td>Returns the selected Sample Rate in Hz for the M&amp;C Measurement task.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQ_Samples Pending</td>
<td>Returns the number of samples available for read in DAQ tasks defined with sample rate &gt; 0. If this property is queried before the DAQ list is started, it always returns 0. Start the DAQ list first with mcDAQStartStop before you query this property.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropDAQ_TimeSinceLastFrame</td>
<td>Indicates how much time has passed (in seconds) since the measurement session received the last DAQ frame. You can reuse this property to restart the measurement when the value increases a threshold (for example, 0.5 seconds), assuming the ECU stopped sending DAQ messages and must be restarted.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>mcPropDB_Filename</td>
<td>Returns the A2L Database file name with which the task has been opened. The value of this property cannot be changed using mcSetProperty.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDB_Filename_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropDB_Filename.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDTOID</td>
<td>Returns the DTO CAN Identifier (Data Transfer Object) for CCP or XCP on CAN which is used to send commands and data from the slave device to the host.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_BaudRate</td>
<td>Returns the baud rate in use.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_ByteOrder</td>
<td>Returns the byte order of the slave device.</td>
</tr>
<tr>
<td></td>
<td>0—MSB_LAST</td>
<td>The Slave device uses the MSB_LAST (Intel) byte ordering.</td>
</tr>
<tr>
<td></td>
<td>1—MSB_FIRST</td>
<td>The Slave device uses the MSB_FIRST (Motorola) byte ordering.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_Checksum</td>
<td>Returns the file name of the Checksum DLL used for verifying the checksum.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_Checksum_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_Checksum.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CmdByteOrder</td>
<td>Returns the byte order for multi-byte command parameters.</td>
</tr>
<tr>
<td></td>
<td>0—MSB_LAST</td>
<td>The CCP Slave device uses the MSB_LAST (Intel) byte ordering.</td>
</tr>
<tr>
<td></td>
<td>1—MSB_FIRST</td>
<td>The CCP Slave device uses the MSB_FIRST (Motorola) byte ordering.</td>
</tr>
</tbody>
</table>
### Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_CRO_ID</td>
<td>Returns the <strong>CRO ID</strong> (Command Receive Object) which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>nctType Taskref</td>
<td>mcPropECU_CRO_Task</td>
<td>NI-CAN Task reference to the CAN Task assigned to the CRO ID.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU.DTO_ID</td>
<td>Returns the <strong>DTO ID</strong> (Data Transmission Object) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.</td>
</tr>
<tr>
<td>nctType Taskref</td>
<td>mcPropECU.DTO_Task</td>
<td>NI-CAN Task reference to the CAN Task assigned to the DTO ID.</td>
</tr>
<tr>
<td>i32</td>
<td>mcPropECU_EventChannel</td>
<td>Translates the event channel name to the event channel number. Pass the event channel name in the Name parameter of GetProperty.</td>
</tr>
<tr>
<td>[u8]</td>
<td>mcPropECU_ID</td>
<td>Returns the slave device identifier. This ID information is optional and specific to the ECU implementation. For more information about the CCP slave ID information refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropECU_ID_DataType</td>
<td>Returns a data type qualifier of the slave device ID information. This ID information is optional and specific to the ECU implementation. For more information about the CCP slave ID information refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropECU_ID_Length</td>
<td>Returns the length of the slave device identifier in bytes.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_Interface</td>
<td>Returns the interface initialized for the task, such as with mcDAQInitialize.</td>
</tr>
<tr>
<td>[u8]</td>
<td>mcPropECU_MasterID</td>
<td>Returns CCP master ID information. This ID information is optional and specific to the ECU implementation. For more information about the CCP master ID information refer to the documentation for the ECU.</td>
</tr>
</tbody>
</table>
### Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>mcPropECU_Name</td>
<td>Returns the name of the selected ECU opened by <code>mcECUSelectEx</code>.</td>
</tr>
<tr>
<td>[u16]</td>
<td>mcPropECU_DAQList Numbers</td>
<td>Returns an array of DAQ list numbers for all DAQ lists defined in the A2L file.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_TimingFactor</td>
<td>Returns the used timing factor, which you can use to increase CCP or XCP command timeout values. For details on the default Command Timeout values, refer to the CCP or XCP Protocol Specification.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropDAQList_Max Length</td>
<td>Returns the maximum length of the DAQ list.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQList_CANId SelectMode</td>
<td>Returns how to select the CAN ID for the specified DAQ list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—CAN_ID_FIXED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CAN Identifier is a predefined fixed number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—CAN_ID_VARIABLE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CAN Identifier is a variable number.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2—CAN_ID_DTO_ID</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The CAN Identifier is the same as the DTO identifier.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQList_CANId</td>
<td>Returns the CAN ID for the specified DAQ list if <code>mcPropDAQList_CANIdSelectMode == CAN_ID_FIXED</code>.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropDAQList_FirstPID</td>
<td>Returns the first Packet ID for the specified DAQ list.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQList_NumberOfEventChannels</td>
<td>Returns the number of allowed event channels for the specified DAQ list.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDAQList_ReductionAllowed</td>
<td>Returns whether or not the specified DAQ list allows reduction.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropDAQList_NumberOfExcludedDAQLists</td>
<td>Returns the length of the array containing the numbers of DAQ lists not working together with the current DAQ list.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropDAQList_ExcludedDAQLists</td>
<td>Returns an array containing the numbers of DAQ lists not working together with the current DAQ list.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQList_Name</td>
<td>Name of the DAQ list (measurement source). Pass the DAQ list number converted to a string in the Name parameter of GetProperty. The available DAQ list number can be obtained by the ECU_DAQListNumbers property.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQList_Name_Size</td>
<td>Call this property before calling mcPropDAQList_Name to find the amount of storage needed to get the name value.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQList_DisplayIdentifier</td>
<td>Optional property you can use as a display name as an alternative to the DAQList_Name property.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQList_DisplayIdentifier_Size</td>
<td>Call this property before calling mcPropDAQList_DisplayIdentifier to find the amount of storage needed to get the display identifier value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_Name_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_Name.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedChkDllPath</td>
<td>Determines the directory where the ECU M&amp;C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&amp;C Toolkit expects the DLLs in the same directory as the A2L file. If your DLLs are in a different directory, set this property pointing to this directory.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedChkDllPath_Size</td>
<td>Returns the required buffer size to read the mcPropECU_SeedChkDllPath property.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_Cal</td>
<td>Returns the file name of the SeedKey DLL used for Calibration purposes.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_SeedKey_Cal_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_Cal.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_DAQ</td>
<td>Returns the file name of the SeedKey DLL used for DAQ purposes.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_SeedKey_DAQ_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_DAQ.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_Prog</td>
<td>Returns the file name of the SeedKey DLL used for programming purposes.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_SeedKey_Prog_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_Prog.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_XCP</td>
<td>Returns the file name of the SeedKey DLL for XCP.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_LogFileName</td>
<td>Returns the filename (full path) where the CCP or XCP protocol traffic is logged in ASCII format for debugging purposes. An empty path indicates no logging (default). Note that on RT and cRIO systems, the logfile is created on the target system and must be transferred to the host after logging has been completed. Note that no additional CAN port is used for the logging, which makes this method superior to any other method such as running a bus monitor parallel.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_LogFileName_Size</td>
<td>Returns the size of the buffer needed to retrieve the mcPropECU_LogFileName property.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_SeedKey_XCP_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_XCP.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropECU_Single_Byte_DAQ_Lists</td>
<td>Determines if an ECU supports single-byte or multi-byte DAQ list entries.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_StationAddress</td>
<td>Returns the station address of the slave device. CCP is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a Station Address that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its Station Address, the ECU must not react to CCP commands sent by the CCP master.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T1</td>
<td>Returns one of the seven timeout values (in milliseconds) defined in the XCP standard for the various XCP commands. For details of which timeout applies to a specific command, refer to the XCP standard. The values are typically read from an A2L file, but may be overridden manually. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T2</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T3</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T4</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T5</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T6</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_XCP_Timeout_T7</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_std</td>
<td>Returns the timeout value (in milliseconds) for most of the CCP commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default: 40. Standard: 25. The default is chosen slightly higher to allow for slower ECUs. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_pgm</td>
<td>Returns the timeout value (in milliseconds) for the CCP programming commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default: 120. Standard: 100. The default is chosen slightly higher to allow for slower ECUs. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_mem</td>
<td>Returns the timeout value (in milliseconds) for the CCP memory commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default and Standard: 30000. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_diag</td>
<td>Returns the timeout value (in milliseconds) for the CCP DIAG_SERVICE command. Default and Standard: 500. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_act</td>
<td>Returns the timeout value (in milliseconds) for the CCP ACTION_SERVICE command. Default: 500. Standard: 5000. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropGen_Version_Build</td>
<td>Returns the build number of the ECU M&amp;C software. This number applies to Development, Alpha, and Beta phase only, and should be ignored for Release phase.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropGen_Version_Comment</td>
<td>Returns a comment string for the ECU M&amp;C software. If you received a custom release of ECU M&amp;C from National Instruments, this comment often describes special features of the release.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropGen_Version_Comment_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropGen_Version_Comment.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropGen_Version_Major</td>
<td>Returns the major version of the ECU M&amp;C software, such as the 1 in version 1.2.5.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropGen_Version_Minor</td>
<td>Returns the minor version of the ECU M&amp;C software, such as the 2 in version 1.2.5.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropGen_Version_Update</td>
<td>Returns the update version of the ECU M&amp;C software, such as the 5 in version 1.2.5.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropIPAddress</td>
<td>Returns the IP address for XCP on Ethernet (TCP or UDP) as a string.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropIPAddress_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropIPAddress.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropIPPort</td>
<td>Returns the IP port for XCP on Ethernet (TCP or UDP).</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropMeas_Address</td>
<td>Returns the address of the selected Measurement in the memory of the control unit.</td>
</tr>
</tbody>
</table>
| u32       | mcPropMeas_BByteOrder             | Returns the specified byte order:  
0—Intel format  
Bytes are in little-endian order, with least-significant bit first.  
1—Motorola format  
Bytes are in big-endian order, with most-significant bit first.                                                                                           |
<p>| u8        | mcPropMeas_Datatype               | Returns the data type of the Measurement task.                                                                                                                                                            |
| u8        | mcPropMeas_Extension              | Returns the address extension of the ECU address. This optional parameter may contain additional address information defined in the A2L database. For instance it can be used, to distinguish different address spaces of an ECU (multi-microcontroller devices). |</p>
<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropMeas_IsVirtual</td>
<td>Returns whether the Measurement is virtual. Virtual Measurements are not transmitted by the ECU but are calculated in the application. They return an error when opened in a DAQ list.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropMeas_Maximum</td>
<td>Returns the maximum value of the Measurement.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropMeas_Minimum</td>
<td>Returns the minimum value of the Measurement.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropMeas_ReadOnly</td>
<td>Returns TRUE if the selected Measurement is read only and can only be accessed through mcMeasurementRead, or returns FALSE if the Measurement can be accessed through mcMeasurementWrite as well.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropMeas_Unit</td>
<td>Returns the unit string defined for this Measurement in the A2L database.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropMeas_Unit_Size</td>
<td>Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropMeas_Unit.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropMeas_Scale_Factor</td>
<td>Returns the scaling factor defined for this Measurement in the A2L database.</td>
</tr>
<tr>
<td>f64</td>
<td>mcPropMeas_Scale_Offset</td>
<td>Returns the scaling offset defined for this Measurement in the A2L database.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropMeas_Scale_Type</td>
<td>Returns the scaling type defined for this Measurement in the A2L database.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>0: Unknown</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The type of the scaling could not be derived from the A2L file content.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>1: Rational Function</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The related scaling is based on a rational function of second order. This</td>
</tr>
<tr>
<td></td>
<td></td>
<td>covers also the linear scaling, given by factor and offset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>2: Enumeration Text</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The related scaling is based on the COMPU_VTAB keyword within the A2L file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read functions return nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write functions accept nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• It is possible to use mcDoubleToText and mcTextToDouble to convert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>between enumeration text values and double values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>3: Range Text</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>The related scaling is based on the COMPU_VTAB_RANGE keyword within the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A2L file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Read functions return nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Write functions accept nonscaled, numeric values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• It is possible to use mcDoubleToText and mcTextToDouble to convert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>between range text values and double values.</td>
</tr>
</tbody>
</table>


**Chapter 6  ECU M&C API for C**

**Table 6-4. Values for PropertyID (Continued)**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropMeas_Scale_TextValues_Size</td>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the length needed to store the comma-separated list of text values that can be converted into raw values (refer to the mcPropMeas_Scale_TextValues property).</td>
</tr>
<tr>
<td>str</td>
<td>mcPropMeas_Scale_TextValues</td>
<td>If the scaling type is 2 = Enumeration Text or 3 = Range Text, you can use this property to request the comma-separated list of text values which can be converted into raw values.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_ACTION_SERVICE</td>
<td>Returns whether the ECU supports the optional CCP Command ACTION_SERVICE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_BUFF_CHKSUM</td>
<td>Returns whether the ECU supports the optional CCP Command BUILD_CHKSUM.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_CLEAR_MEMORY</td>
<td>Returns whether the ECU supports the optional CCP Command CLEAR_MEMORY.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_DIAG_SERVICE</td>
<td>Returns whether the ECU supports the optional CCP Command DIAG_SERVICE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_DNLOAD_6</td>
<td>Returns whether the ECU supports the optional CCP Command DNLOAD_6.</td>
</tr>
<tr>
<td>Data Type</td>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GET_ACTIVE_CAL_PAGE</td>
<td>Returns whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_PAGE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GET_S_STATUS</td>
<td>Returns whether the ECU supports the optional CCP Command GET_S_STATUS.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GET_SEED</td>
<td>Returns whether the ECU supports the optional CCP Command GET_SEED.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_MOVE</td>
<td>Returns whether the ECU supports the optional CCP Command MOVE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_PROGRAM</td>
<td>Returns whether the ECU supports the optional CCP Command PROGRAM.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_PROGRAM_6</td>
<td>Returns whether the ECU supports the optional CCP Command PROGRAM_6.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SELECT_CAL_PAGE</td>
<td>Returns whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SET_S_STATUS</td>
<td>Returns whether the ECU supports the optional CCP Command SET_S_STATUS.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SHORT_UP</td>
<td>Returns whether the ECU supports the optional CCP Command SHORT_UP.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_START_STOP_ALL</td>
<td>Returns whether the ECU supports the optional CCP Command START_STOP_ALL.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_TEST</td>
<td>Returns whether the ECU supports the optional CCP Command TEST.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_UNLOCK</td>
<td>Returns whether the ECU supports the optional CCP Command UNLOCK.</td>
</tr>
</tbody>
</table>
Table 6-4. Values for PropertyID (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| u8        | mcPropPGM_AccessMethod    | Returns the selected access mode for mcProgram and mcClearMemory:
|           |                           | 0x00—**Absolute Access Mode** (default). The MTA uses physical addresses.                                                                  |
|           |                           | 0x01—**Functional Access Mode**. The MTA functions as a block sequence number of the new flash content file.                                  |
|           |                           | 0x80...0xFF—**User defined**. It is possible to use different access modes for clearing and programming.                                     |
| u8        | mcPropPGM_Compression Method | Returns the selected compression method used for mcProgram.                                                                                     |
|           |                           | 0—Data is uncompressed (default).                                                                                                               |
|           |                           | 0x80...0xFF—User defined.                                                                                                                      |
| u8        | mcPropPGM_Encryption Method | Returns the selected encryption method used for mcProgram.                                                                                      |
|           |                           | 0—Data is not encrypted (default).                                                                                                              |
|           |                           | 0x80...0xFF—User defined.                                                                                                                      |
| u8        | mcPropPGM_Programming Method | Returns the selected programming method used for mcProgram.                                                                                      |
|           |                           | 0—Sequential programming (default).                                                                                                             |
|           |                           | 0x80...0xFF—User defined.                                                                                                                      |
| u32       | mcPropGroup_IsRoot        | Returns a nonzero value for Groups being root.                                                                                                  |
mcMeasurementCreate

**Purpose**

Creates a Measurement object in memory.

**Format**

```c
mcTypeStatus mcMeasurementCreate(
    mcTypeTaskRef ECURefNum,
    char *MeasurementName,
    mcAddress Address,
    i32 DataType,
    u8 DataSize,
    char *ConversionName);
```

**Input**

- **ECURefNum**
  
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUCreate`.

- **Address**
  
  Configures the target address for the programming operation in the ECU. `mcAddress` is a C struct consisting of:

  - **Address**
    
    Specifies the address part of the programming address.

  - **Extension**
    
    Extension contains the extension part of the address.

- **DataType**
  
  `DataType` sets the data type of the measurement task.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Unsigned byte</td>
</tr>
<tr>
<td>1</td>
<td>Signed byte</td>
</tr>
<tr>
<td>2</td>
<td>Unsigned word</td>
</tr>
<tr>
<td>3</td>
<td>Signed word</td>
</tr>
<tr>
<td>4</td>
<td>Unsigned long</td>
</tr>
<tr>
<td>5</td>
<td>Signed long</td>
</tr>
<tr>
<td>6</td>
<td>Float 32</td>
</tr>
</tbody>
</table>
DataSize

Sets the size of the measurement data and corresponds to the selected DataType.

<table>
<thead>
<tr>
<th>Data Format</th>
<th>DataSize</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned byte</td>
<td>1</td>
</tr>
<tr>
<td>Signed byte</td>
<td>1</td>
</tr>
<tr>
<td>Unsigned word</td>
<td>2</td>
</tr>
<tr>
<td>Signed word</td>
<td>2</td>
</tr>
<tr>
<td>Unsigned long</td>
<td>4</td>
</tr>
<tr>
<td>Signed long</td>
<td>4</td>
</tr>
<tr>
<td>Float 32</td>
<td>4</td>
</tr>
</tbody>
</table>

ConversionName

ConversionName identifies the referred conversion object that mcConversionCreate defines.

Output

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

Use mcMeasurementCreate to create a measurement object in memory instead of referring to a predefined measurement in the A2L database.
mcMeasurementRead

Purpose
Reads a single Measurement value from the ECU.

Format
```c
mcTypeStatus mcMeasurementRead(  
    mcTypeTaskRef ECURefNum,  
    char *MeasurementName,  
    f64 *Value);
```

Input
- ECURefNum ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.
- MeasurementName MeasurementName is the name of a Measurement channel stored in the A2L database file from which a Measurement value is to be read.

Output
- Value Returns a single sample for the Measurement channel initialized in MeasurementName.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcMeasurementRead performs a single point read (upload) of a single Measurement from the selected ECU without opening a Measurement task.
mcMeasurementWrite

Purpose

Writes a single Measurement value to the ECU.

Format

mcTypeStatus mcMeasurementWrite(
    mcTypeTaskRef ECURefNum,
    char *MeasurementName,
    f64 Values);

Input

ECURefNum ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.

MeasurementName MeasurementName is the name of a Measurement channel stored in the A2L database file to which a Measurement value is to be written.

Values Writes a single sample for the Measurement channel initialized in MeasurementName.

Output

Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcMeasurementWrite performs a single point write (download) of a Measurement into the selected ECU without opening a Measurement task. mcMeasurementWrite can only be performed if the Measurement channel is not set to read only. To query if an ECU Measurement channel can be accessed by mcMeasurementWrite, call mcGetProperty with the parameter mcPropMeas_ReadOnly.
**mcProgram**

**Purpose**
Programs a memory block on the ECU.

**Format**
```c
mcTypeStatus mcProgram(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u32 BlockSize,
    u8 *Data);
```

**Input**
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Address**
  Configures the target address for the programming operation in the ECU. `mcAddress` is a C struct consisting of:
  - **Address**
    Specifies the address part of the programming address.
  - **Extension**
    Contains the extension part of the address.
- **BlockSize**
  BlockSize determines the size of the data block which is transferred to the ECU and used for programming from the MTA0 target.
- **Data**
  Data contains the byte array that is transmitted to the ECU.

**Output**

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

If you are using the CCP protocol, `mcProgram` implements the CCP command PROGRAM. The command is used to program the specified data into non-volatile ECU memory (Flash, EEPROM, etc.). Programming starts at the selected MTA0 address and extension defined in the `Address` struct. The `mcProgram` function auto-increments the ECU MTA0 address.

If you are using the XCP protocol, `mcProgram` implements the XCP command PROGRAM. The command is used to program a non-volatile memory segment inside the ECU slave. Depending on the access mode (defined by `PROGRAM_FORMAT`), two different concepts are supported. The end of the memory segment is indicated when `BlockSize` is set to 0. The end of the overall programming sequence is indicated by a using the `mcProgramReset` command which executes the XCP command `PROGRAM_RESET`, causing the slave device to move into a disconnected state. Usually a hardware reset of the slave device is executed. This command may support block transfer similar to the commands `DOWNLOAD` and `DOWNLOAD_NEXT`. For further information on how to use `mcProgram` and details on block mode transfers refer to the ASAM XCP Part 2 Protocol Layer Specification.
mcProgramReset

Purpose
Indicates the end of a programming sequence.

Format
```c
mcTypeStatus mcProgramReset(
    mcTypeTaskRef ECURefNum);
```

Input

| ECURefNum | ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx. |

Output

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
If you are using the XCP protocol, mcSetProperty implements the XCP command PROGRAM_RESET. This optional command indicates the end of a non-volatile memory programming sequence and may or may not have a response from the ECU. In either case, the slave device will go into a disconnected state.

mcSetProperty may be used to reset a slave device for other purposes. For further information on how to use program ECU memory and to use the mcSetProperty command refer to the ASAM XCP Part 2 Protocol Layer Specification.
mcProgramStart

Purpose
Indicates the start of a programming sequence.

Format
\[
\text{mcTypeStatus \ mcProgramStart(}
\text{mcTypeTaskRef ECURefNum);} \nonumber
\]

Input
ECURefNum
ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.

Output
Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
If you are using the XCP protocol, mcProgramStart implements the XCP command PROGRAM_START. This optional command the beginning of a programming sequence into a non-volatile memory area. If the slave device is not in a state which permits programming, an error is returned. The memory programming commands The end of a non-volatile memory programming sequence is indicated by using the mcSetProperty function.

For further information on how to use program ECU memory and to use the mcProgramStart command refer to the ASAM XCP Part 2 Protocol Layer Specification.
mcSetProperty

Purpose
Sets a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.

Format
```
mcTypeStatus mcSetProperty(
    mcTypeTaskRef RefNum,
    cstr Name,
    u32 PropertyID,
    u32 SizeOfValue,
    void *Value);
```

Input
- **RefNum**
  RefNum is any ECU M&C task reference which consists of a valid link to the opened A2L database (DBRefNum), a selected ECU (ECURefNum) or a Measurement task (DAQRefNum). RefNum must be valid for the related PropertyID type.
- **Name**
  Name is not used and can be set to NULL. This parameter maybe used for further extensions.
- **PropertyID**
  Selects the property to set. For a description of each property, including its data type and PropertyId, refer to the Properties section.
- **SizeOfValue**
  Number of bytes allocated for the Value output. This size normally depends on the data type listed in the description of the property.
- **Value**
  Provides the property value. PropertyId determines the data type of the value.

Output

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

There are four types of properties which can be modified in the poly input value: ECU-specific properties, DAQ-specific properties, Characteristic-specific properties, and Measurement-specific properties.

ECU-Specific Properties

You cannot set an ECU property while the application is connected to the ECU. If you need to change a ECU property prior to connecting, call mcECUSelectEx, followed by mcSetProperty, and then mcECUConnect. After you connect to the ECU, you also can change a property by calling mcECUDisconnect, followed by mcSetProperty, and then mcECUConnect to restart the task. Table 6-5 contains a listing of ECU-specific values for PropertyID.

DAQ-Specific Properties

You cannot set a DAQ property while a Measurement task is running. If you need to change a property prior to starting a Measurement task call mcDAQInitialize, followed by mcSetProperty, and then mcDAQStartStop. After you start the Measurement task, you also can change a property by calling mcDAQStartStop, followed by mcSetProperty, and then mcDAQStartStop to restart the task. Table 6-6 contains a listing of ECU-specific values for PropertyID.
## Properties

**Table 6-5.** ECU-Specific Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropCANBaudRate</td>
<td>Sets the CAN Baud rate for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropCANTermination</td>
<td>For all XNET devices, the termination is software selectable. XNET provides the option of 80 Ω between Bus Plus and Bus Minus or no termination. The Termination property configures the onboard termination of the NI-XNET interface CAN connector (port). The Boolean property supports two values: TRUE = Termination ON and FALSE = Termination Off. However, different CAN hardware has different termination requirements, and the termination values have different meanings. Refer to the Termination attribute in the XNET API for more details. (This property is supported for NI-XNET devices only.)</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropCmd_EXCHANGE_ID</td>
<td>Sets whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropCROID</td>
<td>Sets the CRO CAN Identifier (Command Receive Object) for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropDTOID</td>
<td>Sets the DTO CAN Identifier (Data Transfer Object) for CCP or XCP on CAN which is used to send commands and data from the slave device to the host.</td>
</tr>
</tbody>
</table>
Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_BaudRate</td>
<td>Sets the Baud rate in use by the selected interface. This property applies to all tasks initialized with the NI-CAN or NI-XNET interface. You can specify the following basic baud rates as the numeric rate: 33333, 83333, 100000, 125000, 200000, 250000, 400000, 500000, 800000, and 1000000. You can specify advanced baud rates as 8000XXYY hex, where YY is the value of Bit Timing Register 0 (BTR0), and XX is the value of Bit Timing Register 1 (BTR1). For more information, refer to the Interface Properties dialog in MAX. The value of this property is originally set within MAX, but it can be changed using <code>mcSetProperty</code>.</td>
</tr>
</tbody>
</table>
| u32       | mcPropECU_ByteOrder | Sets the Byte Order of the slave device.  
0—MSB_LAST  
The Slave device uses the MSB_LAST (Intel) byte ordering.  
1—MSB_FIRST  
The Slave device uses the MSB_FIRST (Motorola) byte ordering. |
| str       | mcPropECU_Checksum  | Sets the file name of the Checksum DLL used for verifying the checksum. |
| u32       | mcPropECU_CmdByteOrder | Sets the byte order for multi-byte command parameters.  
0—MSB_LAST  
The CCP Slave device uses the MSB_LAST (Intel) byte ordering.  
1—MSB_FIRST  
The CCP Slave device uses the MSB_FIRST (Motorola) byte ordering. |
Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_CRO_ID</td>
<td>Sets the CAN identifier for the CRO ID (Command Receive Object), which is used to send commands and data from the host to the slave device.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_DTO_ID</td>
<td>Sets the DTO ID (Data Transmission Object) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_MasterID</td>
<td>Sets CCP master ID information. This ID information is optional and specific to the ECU implementation. For more information about the CCP master ID information refer to the documentation for the ECU.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedChkDllPath</td>
<td>Determines the directory where the ECU M&amp;C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&amp;C Toolkit expects the DLLs in the same directory as the A2L file. If your DLLs are in a different directory, set this property pointing to this directory.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_Cal</td>
<td>Sets the file name of the SeedKey DLL used for Calibration purposes.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_DAQ</td>
<td>Sets the file name of the SeedKey DLL used for DAQ purposes.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_Prog</td>
<td>Sets the file name of the SeedKey DLL used for programming purposes.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropECU_SeedKey_XCP</td>
<td>Sets the file name of the SeedKey DLL for XCP.</td>
</tr>
</tbody>
</table>
Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>str</td>
<td>mcPropECU_LogFileName</td>
<td>Sets a filename (full path) where the CCP or XCP protocol traffic is logged in ASCII format for debugging purposes. Setting this value to an empty path (NULL or empty string) disables logging (default). Note that on RT and cRIO systems, the logfile is created on the target system and must be transferred to the host after logging has been completed. Note that no additional CAN port is used for the logging, which makes this method superior to any other method such as running a bus monitor parallel.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropECU_Single_Byte_DAQ_Lists</td>
<td>Sets the ECU to support single-byte or multi-byte DAQ list entries.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_StationAddress</td>
<td>Sets the station address of the slave device. CCP is based on the idea that several ECUs can share the same CAN Arbitration IDs for CCP communication. To avoid communication conflicts, CCP defines a Station Address that must be unique for all ECUs sharing the same CAN Arbitration IDs. Unless an ECU has been addressed by its Station Address, the ECU must not react to CCP commands sent by the CCP master.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_TimingFactor</td>
<td>Sets the timing factor, which you can use to increase CCP or XCP command timeout values. For details on the default Command Timeout values, refer to the CCP or XCP Protocol Specification.</td>
</tr>
</tbody>
</table>
### Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T1</td>
<td>Sets one of the seven timeout values (in milliseconds) defined in the XCP standard for the various XCP commands. For details of which timeout applies to a specific command, refer to the XCP standard. The values are typically read from an A2L file, but may be overridden manually. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T4</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_XCP_Timeout_T7</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_std</td>
<td>Sets the timeout value (in milliseconds) for most of the CCP commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default: 40. Standard: 25. The default is chosen slightly higher to allow for slower ECUs. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td></td>
<td>mcPropECU_CCP_Timeout_T_pgm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>mcPropECU_CCP_Timeout_T_mem</td>
<td></td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_mem</td>
<td>Sets the timeout value (in milliseconds) for the CCP memory commands. For details of which timeout applies to a specific command, refer to the CCP standard. Default and Standard: 30000. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
</tbody>
</table>
### Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_diag</td>
<td>Sets the timeout value (in milliseconds) for the CCP DIAG_SERVICE command. Default and Standard: 500. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropECU_CCP_Timeout_T_act</td>
<td>Sets the timeout value (in milliseconds) for the CCP ACTION_SERVICE command. Default: 500. Standard: 5000. Note that the mcPropECU_TimingFactor property might modify this value.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropIPAddress</td>
<td>Sets the IP address for XCP on Ethernet (TCP or UDP) as a string.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropIPPort</td>
<td>Sets the IP port for XCP on Ethernet (TCP or UDP).</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_ACTION_SERVICE</td>
<td>Sets whether the ECU supports the optional CCP Command ACTION_SERVICE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_BUILDCHKSUM</td>
<td>Sets whether the ECU supports the optional CCP Command BUILD_CHKSUM.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_CLEARMEMORY</td>
<td>Sets whether the ECU supports the optional CCP Command CLEAR_MEMORY.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_DIAGSERVICE</td>
<td>Sets whether the ECU supports the optional CCP Command DIAG_SERVICE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_DNLOAD_6</td>
<td>Sets whether the ECU supports the optional CCP Command DNLOAD_6.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GETACTIVECALPAGE</td>
<td>Sets whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_PAGE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GETS_STATUS</td>
<td>Sets whether the ECU supports the optional CCP Command GET_S_STATUS.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_GETSEED</td>
<td>Sets whether the ECU supports the optional CCP Command GET_SEED.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_MOVE</td>
<td>Sets whether the ECU supports the optional CCP Command MOVE.</td>
</tr>
</tbody>
</table>
### Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropOptCmd_PROGRAM</td>
<td>Sets whether the ECU supports the optional CCP Command PROGRAM.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_PROGRAM_6</td>
<td>Sets whether the ECU supports the optional CCP Command PROGRAM_6.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SELECT_CAL_PAGE</td>
<td>Sets whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SET_S_STATUS</td>
<td>Sets whether the ECU supports the optional CCP Command SET_S_STATUS.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_SHORT_UP</td>
<td>Sets whether the ECU supports the optional CCP Command SHORT_UP.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_START_STOP_ALL</td>
<td>Sets whether the ECU supports the optional CCP Command START_STOP_ALL.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_TEST</td>
<td>Sets whether the ECU supports the optional CCP Command TEST.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropOptCmd_UNLOCK</td>
<td>Sets whether the ECU supports the optional CCP Command UNLOCK.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropPGM_AccessMethod</td>
<td>Selects the selected access mode for mcProgram and mcClearMemory:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x00—<strong>Absolute Access Mode</strong> (default). The MTA uses physical addresses.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x01—<strong>Functional Access Mode</strong>. The MTA functions as a block sequence number of the new flash content file.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80...0xFF—<strong>User defined</strong>. It is possible to use different access modes for clearing and programming.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropPGM_Compression Method</td>
<td>Selects the selected compression method used for mcProgram.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—Data is uncompressed (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80...0xFF—User defined.</td>
</tr>
</tbody>
</table>
Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8</td>
<td>mcPropPGM_Encryption Method</td>
<td>Selects the selected encryption method used for <code>mcProgram</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—Data is not encrypted (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80...0xFF—User defined.</td>
</tr>
<tr>
<td>u8</td>
<td>mcPropPGM_Programming Method</td>
<td>Selects the selected programming method used for <code>mcProgram</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—Sequential programming (default).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x80...0xFF—User defined.</td>
</tr>
</tbody>
</table>
### Table 6-6. DAQ-Specific Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropDAQ_DTO_ID</td>
<td>Sets the DTO ID (Data Transmission Object) which is used by the ECU to respond to send data from the DAQ lists to the CCP master.</td>
</tr>
<tr>
<td>str</td>
<td>mcPropDAQ_EventChannel.Name</td>
<td>Sets the event channel name to which the Measurement task is assigned. If there is no event channel name defined in the A2L file, you can set the Event Channel Number manually by passing a decimal number as a string.</td>
</tr>
<tr>
<td>i32</td>
<td>mcPropDAQ_Mode</td>
<td>Sets the mode of an M&amp;C Measurement task.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—DAQ List</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The data is transmitted from the ECU in equidistant time intervals as defined in the A2L database. The data can be read back with mcDAQRead as Single point data using sample rate = 0, or as waveform using a sample rate &gt; 0. Input channel data is received from the DAQ messages. Use mcDAQRead to obtain input samples as single-point, array, or waveform.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—Polling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In this mode the data from the Measurement task is uploaded from the ECU whenever mcDAQRead is called.</td>
</tr>
<tr>
<td>u16</td>
<td>mcPropDAQ_Prescaler</td>
<td>Sets the Prescaler, which reduces the desired transmission frequency of the associated DAQ list.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Characteristic-Specific Properties

Table 6-7. Characteristic-Specific Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>double[]</td>
<td>mcPropChar_X_Axis</td>
<td>Sets the X-axis values on which the Characteristic is defined. The Characteristic dimension must be at least 1.</td>
</tr>
<tr>
<td>double[]</td>
<td>mcPropChar_Y_Axis</td>
<td>Sets the Y-axis values on which the Characteristic is defined. The Characteristic dimension must be 2.</td>
</tr>
<tr>
<td>u32</td>
<td>mcPropChar_ByteOrder</td>
<td>Sets the specified byte order of the selected Characteristic:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—<strong>Intel format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes are in little-endian order, with least-significant bit first.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—<strong>Motorola format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes are in big-endian order, with most-significant bit first.</td>
</tr>
</tbody>
</table>

Measurement-Specific Properties

Table 6-8. Measurement-Specific Value Types for the PropertyID Input Value

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u32</td>
<td>mcPropMeas_ByteOrder</td>
<td>Sets the specified byte order of the selected Measurement:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—<strong>Intel format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes are in little-endian order, with least-significant bit first.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—<strong>Motorola format</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bytes are in big-endian order, with most-significant bit first.</td>
</tr>
</tbody>
</table>
**mcStatusToString**

**Purpose**
Converts a status code into a descriptive string.

**Format**
```
mcTypeStatus mcStatusToString(
    mcTypeTaskRef Status,
    u32 SizeofString,
    str ErrorString);
```

**Input**
- **Status**: Nonzero status code returned from an ECU M&C function.
- **SizeofString**: SizeofString buffer (in bytes).

**Output**
- **ErrorString**: ASCII string that describes Status.

**Description**
When the status code returned from an ECU M&C function is nonzero, an error or warning is indicated. This function is used to obtain a description of the error/warning for debugging purposes.

The return code is passed into the `Status` parameter. The `SizeofString` parameter indicates the number of bytes available in the string for the description. The description is truncated to size `SizeofString` if needed, but a size of 300 characters is large enough to hold any description. The text returned in `ErrorString` is null-terminated, so it can be used with ANSI C functions such as `printf`. For applications written in C or C++, each ECU M&C function returns a status code as a signed 32-bit integer. The following table summarizes the ECU M&C use of this status.

<table>
<thead>
<tr>
<th>Status Code</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative</td>
<td>Error—Function did not perform expected behavior.</td>
</tr>
<tr>
<td>Positive</td>
<td>Warning—Function performed as expected, but a condition arose that may require attention.</td>
</tr>
<tr>
<td>Zero</td>
<td>Success—Function completed successfully.</td>
</tr>
</tbody>
</table>
The application code should check the status returned from every ECU M&C function. If an error is detected, you should close all ECU M&C handles and exit the application. If a warning is detected, you can display a message for debugging purposes or simply ignore the warning.

The following piece of code shows an example of handling ECU M&C status during application debugging.

```c
status = ncDatabaseOpen("TestDataBase.A2L", &MyDbHandle);
PrintStat(status, "mcOpenDatabase");
```

where the function PrintStat has been defined at the top of the program as:

```c
void PrintStat(mcTypeStatus status, char *source)
{
    char statusString[300];
    if(status != 0)
    {
        mcStatusToString(status, sizeof(statusString), statusString);
        printf("\n%s\nSource = %s\n", statusString, source);
        if (status < 0)
        {
            mcDatabaseClose(MyDbHandle);
            exit(1);
        }
    }
}
```

In some situations, you may want to check for specific errors in the code. For example, when `mcCharacteristicRead` times out, you may want to continue communication, rather than exit the application. To check for specific errors, use the constants defined in `niemc.h`. These constants have the same names as described in this manual. For example, to check for a function timeout, use:

```c
if (status == mcErrorTimeout)
    ...
```
mcTextToDouble

Purpose
Converts a text string to a numerical value using an enumeration or range text scaling.

Format
```
mcTypeStatus mcTextToDouble(
    mcTypeTaskRef ECURefNum,
    u32 ObjectType,
    cstr ObjectName,
    cstr TextValue,
    double *Value);
```

Input
- **ECURefNum**: The task reference that links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **ObjectType**: Indicates the type of the object named in `ObjectName`. Valid values are:
  1. Measurement Name
  2. Characteristic Name
- **ObjectName**: Indicates the object (measurement or characteristic) for which the enumeration or range text scaling is performed.
- **TextValue**: The text that you want to turn into the numeric representation.

Output
- **Value**: Returns the converted value.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

mcTextToDouble performs text to double conversion for measurement or characteristic values.

Especially if the measurement or characteristic has an associated enumeration or range text type scaling, the text input will be converted into the numeric representation, using the related COMPU_VTAB or COMPU_VTAB_RANGE table.
**mcUpload**

**Purpose**
Uploads data from an ECU.

**Format**
```c
mcTypeStatus mcUpload(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u32 BlockSize,
    u8 *Data);
```

**Input**
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Address**
  Configures the source address for the upload operation in the ECU. `mcAddress` is a C struct consisting of:
  - **Address**
    Specifies the address part of the source address.
  - **Extension**
    Extension contains the extension part of the address.

- **BlockSize**
  BlockSize is the size of the data block in bytes to be uploaded.

**Output**
- **Data**
  Data is a byte array which receives the uploaded data information from the ECU.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

If you are using the CCP protocol, `mcUpload` implements the CCP command UPLOAD. A data block of the specified length starting at the specified address is uploaded from the ECU. This function sets the Memory Transfer Address pointer MTA0 to the appropriate value as defined in the `Address` struct.

If you are using the XCP protocol, `mcUpload` implements the XCP command UPLOAD. A data block of the specified length starting at the specified address is uploaded from the ECU. The Memory Transfer Address pointer MTA0 is post-incremented by the given number of data elements. If the slave device does not support block transfer mode, all uploaded data is transferred in a single response packet. If block transfer mode is supported, the uploaded data is transferred in multiple responses on the same request packet. There are no limitations allowed concerning the maximum block size for the master.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to upload data and to use the `mcUpload` command.
mcXCPCopyCalPage

Purpose
Forces a copy transaction of one calibration page to another.

Format
```
mcTypeStatus mcXCPCopyCalPage(
    mcTypeTaskRef ECURefNum,
    u8 SourceSegment,
    u8 SourcePage,
    u8 DestinationSegment,
    u8 DestinationPage);
```

Input
- **ECURefNum**
  - ECURefNum is the task reference which links to the selected ECU.
  - This reference is originally returned from `mcECUSelectEx`.
- **SourceSegment**
  - SourceSegment specifies the logical data segment number source.
- **SourcePage**
  - SourcePage specifies the logical page number source.
- **DestinationSegment**
  - DestinationSegment specifies the logical data segment number destination.
- **DestinationPage**
  - DestinationPage specifies the logical page number destination.

Output
None.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
**Description**

`mcXCPCopyCalPage` implements the XCP command COPY_CAL_PAGE and forces the slave to copy one calibration page to another. This command is only available if more than one calibration page is defined. In principal, any page of any segment can be copied to any page of any other segment but there may be restrictions.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
mcXCPGetCalPage

Purpose
Queries a calibration page setting.

Format
mcTypeStatus mcXCPGetCalPage(
    mcTypeTaskRef ECURefNum,
    u8 Mode,
    u8 Segment,
    u8 *Page);

Input
ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.
Mode specifies the access mode:
    Mode = 1
    The given page is used by the slave device application.
    Mode = 2
    The slave device XCP driver will access the given page.
Segment specifies the selected logical data segment number.

Output
Page returns the logical data page number.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcXCPGetCalPage implements the XCP command GET_CAL_PAGE and queries the logical number for the calibration data page that is currently activated for the specified access mode and data segment.
Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
mcXCPGetId

Purpose
Queries session configuration or slave device identification.

Format
```c
mcTypeStatus mcXCPGetId(
    mcTypeTaskRef ECURefNum,
    u8 Type,
    u32 *Length,
    char *Id);
```

Input
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Type**
  Type specifies the type of the requested identification:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ASCII text</td>
</tr>
<tr>
<td>1</td>
<td>ASAM-MC2 filename without path and extension</td>
</tr>
<tr>
<td>2</td>
<td>ASAM-MC2 filename with path and extension</td>
</tr>
<tr>
<td>3</td>
<td>URL where the ASAM-MC2 file can be found</td>
</tr>
<tr>
<td>4</td>
<td>ASAM-MC2 file to upload128..255 User defined</td>
</tr>
</tbody>
</table>

Output
- **Length**
  Length returns the string length of the Id string.
- **Id**
  Id contains the queried identification string.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.
Description

mcXCPGetId implements the XCP command GET_ID and returns session configuration or slave device identification information of the selected ECU slave device. The supported types are implementation specific of the ECU slave device. The identification string is ASCII text format.
mcXCPGetStatus

Purpose
Queries the current session status from an ECU slave device.

Format
```c
mcTypeStatus mcXCPGetStatus(
    mcTypeTaskRef ECURefNum,
    u8  *SessionStatus,
    u8  *ResourceMask,
    u16 *SessionId);
```

Input
- **ECURefNum** ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

Output
- **SessionStatus** SessionStatus returns the current status of the selected ECU.
- **ResourceMask** ResourceMask is the current resource protection status of the selected ECU.
- **SessionId** SessionId returns the defined session configuration ID.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description
`mcXCPGetStatus` implements the XCP command GET_STATUS and returns all current status information of the selected ECU slave device, including the status of the resource protection, pending store requests and the general status of data acquisition and stimulation.
Current Session Status

SessionStatus contains a bit mask which is described below:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>STORE_CAL_REQ</td>
<td>REQUEST to STORE CALibration data:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—STORE_CAL_REQ mode is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—STORE_CAL_REQ mode is set.</td>
</tr>
<tr>
<td>1</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>2</td>
<td>STORE_DAQ_REQ</td>
<td>REQUEST to STORE DAQ list:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—STORE_DAQ_REQ mode is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—STORE_DAQ_REQ mode is set.</td>
</tr>
<tr>
<td>3</td>
<td>CLEAR_DAQ_REQ</td>
<td>REQUEST to CLEAR DAQ configuration:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—CLEAR_DAQ_REQ is reset.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—CLEAR_DAQ_REQ is set.</td>
</tr>
<tr>
<td>4</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>5</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>6</td>
<td>DAQ_RUNNING</td>
<td>Data Transfer:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—The data transfer is not running.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—The data transfer is running.</td>
</tr>
<tr>
<td>7</td>
<td>RESUME</td>
<td>RESUME Mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0—The slave device is not in RESUME mode.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1—The slave device is in RESUME mode.</td>
</tr>
</tbody>
</table>

The STORE_CAL_REQ flag indicates a pending request to save the calibration data into non-volatile memory. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_STORE_CAL event packet.

The STORE_DAQ_REQ flag indicates a pending request to save the DAQ list setup in non-volatile memory. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_STORE_DAQ event packet.
The CLEAR_DAQ_REQ flag indicates a pending request to clear all DAQ lists in non-volatile memory. All ODT entries are reset to address = 0, extension = 0, size = 0 and bit_offset = FF. Session configuration ID is reset to 0. As soon as the request has been fulfilled, the slave will reset the appropriate bit. The slave device may indicate this by transmitting an EV_CLEAR_DAQ event packet. If the slave device does not support the requested mode, an ERR_OUT_OF_RANGE is returned.

The DAQ_RUNNING flag indicates that at least one DAQ list has been started and is in RUNNING mode.

The RESUME flag indicates that the slave is in RESUME mode.

ResourceMask contains the current resource protection status as a bit mask described below:

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
</table>
| 0          | CAL/PAG | REQuest to STORE CALibration data:  
0—STORE_CAL_REQ mode is reset.  
1—STORE_CAL_REQ mode is set. |
| 1          | Unused | — |
| 2          | DAQ | DAQ list commands (DIRECTION = DAQ):  
0—DAQ list commands are not protected with SEED & Key mechanism.  
1—DAQ list commands are protected with SEED & Key mechanism. |
| 3          | STIM | DAQ list commands (DIRECTION = STIM):  
0—DAQ list commands are not protected with SEED & Key mechanism.  
1—DAQ list commands are protected with SEED & Key mechanism. |
| 4          | PGM | ProGraMming commands:  
0—ProGraMming commands are not protected with SEED & Key mechanism.  
1—ProGraMming commands are protected with SEED & Key mechanism |
| 5          | Unused | — |
The CAL/PAG flags indicate that all commands of the CALibration/PAGing group are protected and will return an ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.

The PGM flags indicate that all the commands of the ProGraMming group are protected and will return a ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.

The parameter `SessionId` contains the Session configuration ID. The session configuration ID must be set by a prior `mcXCPSetRequest` call with STORE_DAQ_REQ set. This allows the master device to verify that automatically started DAQ lists contain the expected data transfer configuration.

<table>
<thead>
<tr>
<th>Bit Number</th>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Unused</td>
<td>—</td>
</tr>
<tr>
<td>7</td>
<td>Unused</td>
<td>—</td>
</tr>
</tbody>
</table>

The CAL/PAG flags indicate that all commands of the CALibration/PAGing group are protected and will return an ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.

The PGM flags indicate that all the commands of the ProGraMming group are protected and will return a ERR_ACCESS_LOCKED upon an attempt to execute the command without a previous successful GET_SEED/UNLOCK sequence.

The parameter `SessionId` contains the Session configuration ID. The session configuration ID must be set by a prior `mcXCPSetRequest` call with STORE_DAQ_REQ set. This allows the master device to verify that automatically started DAQ lists contain the expected data transfer configuration.
**mcXCPProgramPrepare**

**Purpose**
Prepares the programming of non volatile memory.

**Format**
```c
mcTypeStatus mcXCPProgramPrepare(
    mcTypeTaskRef ECURefNum,
    mcAddress Address,
    u16 CodeSize);
```

**Input**
- **ECURefNum**
  - ECURefNum is the task reference which links to the selected ECU.
  - This reference is originally returned from `mcECUSelectEx`.
- **Address**
  - Address is the task reference which links to the selected ECU.
  - This reference is originally returned from `mcECUSelectEx`.
- **Address**
  - Specifies the address part of the target address.
- **Extension**
  - Contains the extension part of the target address.
- **CodeSize**
  - CodeSize determines the size of data to be downloaded.

**Output**

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**
`mcXCPProgramPrepare` may be used to indicate a data download as a pre-condition for non-volatile memory reprogramming. The Memory Transfer address (MTA) pointer is set to the volatile memory location specified by the parameters Address and Extension. The download itself is done by using subsequent standard commands like `mcDownload`. The slave device must ensure that the target memory area is available and it is in an operational state which permits the download of code. If not, an error will be returned.
mcXCPProgramPrepare implements the optional XCP PROGRAM_PREPARE command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM XCP Part 2 Protocol Layer Specification.
mcXCPProgramVerify

Purpose
Verifies the programming of non-volatile ECU memory.

Format
```
mcTypeStatus mcXCPProgramVerify(
    mcTypeTaskRef ECURefNum,
    u8 Mode,
    u16 VerType,
    u32 VerValue);
```

Input
- **ECURefNum** is the task reference which links to the selected ECU. This reference is originally returned from mcECUSelectEx.
- **Mode** describes the verification mode:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Request to start internal routine.</td>
</tr>
<tr>
<td>1</td>
<td>Send a Verification Value stored in VerValue.</td>
</tr>
</tbody>
</table>

- **VerType** specifies the Verification Type of the requested program verification. The Verification Type is a bit mask described below:

<table>
<thead>
<tr>
<th>Verification Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>Calibration area(s) of the flash.</td>
</tr>
<tr>
<td>0x0002</td>
<td>Code area(s) of the flash.</td>
</tr>
<tr>
<td>0x0004</td>
<td>Complete flash content.</td>
</tr>
<tr>
<td>0x0008 ... 0x0080</td>
<td>Reserved.</td>
</tr>
<tr>
<td>0x0100 ... 0xFF00</td>
<td>User defined.</td>
</tr>
</tbody>
</table>

- **VerValue** contains the selected verification value if Mode=1.

Output
None.
Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description

mcXCPProgramVerify implements the XCP command PROGRAM_VERIFY and performs a flash program verification. If VerMode = 0 the master can request the slave to start internal test routines to check whether the new flash contents fits to the rest of the flash. Only the result is of interest. If VerMode = 01, the master can tell the slave that he is sending a Verification Value to the slave. The definition of the Verification Mode is project specific. The master is getting the Verification Mode from the project specific programming flow control and passing it to the slave. The tool needs no further information about the details of the project specific check routines. The XCP parameters allow a wide range of project specific adaptations. The Verification Type is specified in the project specific programming flow control. The master is getting this parameter and passing it to the slave. The definition of the Verification Value is project specific and the use is defined in the project specific programming flow control.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.

mcXCPProgramVerify can be used to verify the success of non-volatile memory reprogramming.

With Mode set to 00 the master can request the slave to start internal test routines to check whether the new flash contents fits to the rest of the flash. Only the result is of interest. With Mode set to 01, the master can tell the slave that he will be sending a Verification value to the slave. The definition of the Verification mode is project-specific. The master receives the Verification mode from the project-specific programming flow control and passes it to the slave.

mcXCPProgramVerify implements the optional XCP PROGRAM_VERIFY command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM XCP Part 2 Protocol Layer Specification.
mcXCPSetCalPage

Purpose
Sets a calibration page.

Format
```
mcTypeStatus mcXCPSetCalPage(
    mcTypeTaskRef ECURefNum,
    u8 Mode,
    u8 Segment,
    u8 Page);
```

Input
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU.
  This reference is originally returned from `mcECUSelectEx`.
- **Mode**
  Mode is a bit mask described below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>The given page is used by the slave device application.</td>
</tr>
<tr>
<td>1</td>
<td>The slave device XCP driver will access the given page.</td>
</tr>
<tr>
<td>2</td>
<td>Unused.</td>
</tr>
<tr>
<td>3</td>
<td>Unused.</td>
</tr>
<tr>
<td>4</td>
<td>Unused.</td>
</tr>
<tr>
<td>5</td>
<td>Unused.</td>
</tr>
<tr>
<td>6</td>
<td>Unused.</td>
</tr>
<tr>
<td>7</td>
<td>The logical segment number is ignored. The command applies to all segments.</td>
</tr>
</tbody>
</table>

- **Segment**
  Segment specifies the selected logical data segment number.

- **Page**
  Page specifies the logical data page number.

Output
None.
Return Value

The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

Description

`mcXCPSetCalPage` implements the XCP command SET_CAL_PAGE and sets the access mode for a calibration data segment, if the slave device supports calibration data page switching. A calibration data segment and its pages are specified by logical numbers.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
mcXCPSetRequest

Purpose
Performs a request to save session and device information to non-volatile memory.

Format
```c
mcTypeStatus mcXCPSetRequest(
    mcTypeTaskRef ECURefNum,
    u8 Mode,
    u16 SessionID);
```

Input
- **ECURefNum**
  ECURefNum is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- **Mode**
  Mode is a bit mask described below:

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Request to store calibration data in non-volatile memory.</td>
</tr>
<tr>
<td>1</td>
<td>Unused.</td>
</tr>
<tr>
<td>2</td>
<td>Request to save all DAQ lists, which have been selected with <code>START_STOP_DAQ_LIST(Select)</code> into non-volatile memory. The slave also must store the session configuration ID in non-volatile memory. Upon saving, the slave first must clear any DAQ list configuration that might already be stored in non-volatile memory.</td>
</tr>
<tr>
<td>3</td>
<td>Request to clear all DAQ lists in non-volatile memory. All ODT entries reset to address = 0, extension = 0, size = 0 and bit_offset = FF. Session configuration ID reset to 0.</td>
</tr>
<tr>
<td>4</td>
<td>Unused.</td>
</tr>
<tr>
<td>5</td>
<td>Unused.</td>
</tr>
<tr>
<td>6</td>
<td>Unused.</td>
</tr>
<tr>
<td>7</td>
<td>Unused.</td>
</tr>
</tbody>
</table>

- **SessionID**
  SessionID is a session configuration ID that is stored in non-volatile memory together with the information requested by the Mode parameter.
Output
None.

Return Value
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the mcStatusToString function of the ECU M&C API to obtain a descriptive string for the return value.

Description
mcXCPSetRequest implements the XCP command SET_REQUEST and is used to save session configuration information into non-volatile memory in the ECU.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to setup a request.
mcXCPSetSegmentMode

**Purpose**
Sets the mode of a specified segment.

**Format**
```
mcTypeStatus mcXCPSetSegmentMode(
    mcTypeTaskRef ECURefNum,
    u8 Segment,
    u8 Mode);
```

**Input**
- ECURefNum
  - `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
- Segment
  - `Segment` specifies the logical data segment number.
- Mode
  - `Mode` specifies the mode for the segment.

**Output**
None.

**Return Value**
The return value indicates the status of the function call as a signed 32-bit integer. Zero means the function executed successfully. A negative value specifies an error, which means the function did not perform the expected behavior. A positive value specifies a warning, which means the function performed as expected, but a condition arose that may require attention.

Use the `mcStatusToString` function of the ECU M&C API to obtain a descriptive string for the return value.

**Description**
`mcXCPSetSegmentMode` implements the XCP command SET_SEGMENT_MODE and sets the selected segment into the specified mode. If `Mode = 0` the segment disables the FREEZE mode, if `Mode = 1` the segment is set to FREEZE mode through an XCP STORE_CAL_REQ operation.

Refer to the ASAM XCP Part 2 Protocol Layer Specification for more information on how to set up a request.
Summary of the CCP Standard

Controller Area Network (CAN)

Bosch developed the Controller Area Network (CAN) in the mid-1980s. Using CAN, devices (controllers, sensors, and actuators) are connected on a common serial bus. This network of devices can be thought of as a scaled-down, real-time, low-cost version of the networks used to connect personal computers. Any device on a CAN network can communicate with any other device using a common pair of wires.

As CAN implementations increased in the automotive industry, CAN was standardized internationally as ISO 11898. CAN chips were created by major semiconductor manufacturers such as Intel, Motorola, and Philips. With these developments, manufacturers of industrial automation equipment began to consider CAN for use in industrial applications. Comparison of the requirements for automotive and industrial device networks showed numerous similarities, including the transition away from dedicated signal lines, low cost, resistance to harsh environments, and high real-time capabilities.

CAN Calibration Protocol (CCP)

The amount of electronics introduced into the automobile has increased significantly. This trend is expected to continue as automobile manufacturers initiate further advances in safety, reliability and comfort. The introduction of advanced control systems—combining multiple sensors, actuators and electronic control units—has begun to place extensive demands on the existing Controller Area Network (CAN) communication bus. To enable the new generation of automotive electronics, new and highly sophisticated software, calibration, measurement, and diagnostic equipment must be used. At this time almost no standards exist in the area of software interfaces for such devices. Each company has its proprietary systems and interfaces to support the development of these high-end configurations.
The CAN Calibration Protocol was originally developed and introduced by Ingenieurbüro Helmut Kleinknecht, a manufacturer of calibration systems, and is used in various application areas in the automotive industry. Afterwards CCP was taken over by the ASAP working group and enhanced with optional functions and is now maintained by the ASAM organization.

**Scope of CCP**

The CAN Calibration Protocol is a CAN-based master-slave protocol for calibration and data acquisition using the CAN 2.0B standard (11-bit and 29-bit identifiers), which includes 2.0A (11-bit identifier). A single master device (host) can be connected to one or more slave devices. Before a slave device may accept commands from the host, the host must establish a logical point-to-point connection to the slave device. After this connection has been established, the slave device must acknowledge each command received from the host within a specific time.

CCP offers continuous or event driven data acquisition from the controllers, as well as memory transfers to and control functions in the controllers for calibration purposes.

With these functions, CCP may be used in:

- The development of electronic control units (ECU)
- Systems for functional and environmental tests of an ECU
- Test systems and test stands for controlled devices (combustion engines, gearboxes, suspension systems, climate-control systems, body systems, anti-locking systems)
- On-board test and measurement systems of pre-series vehicles
- Any non-automotive application of CAN-based distributed electronic control systems

CCP defines two function sets—one for control/memory transfer, and one for data acquisitions that are independent of each other and may run asynchronously. The control commands are used to carry out functions in the slave device, and may use the slave to perform tasks on other devices. The data acquisition commands are used for continuous data acquisition from a slave device. The devices continuously transmit internal data according to a list that has been configured by the host. Data acquisition is initiated by the host, then executed by the slave device, and may be based on a fixed sampling rate or be event-driven.
CCP Protocol Definition

Two communication objects are defined by CCP to handle the communication between host and slave devices—The CommandReceiveObject (CRO), which is used to send commands and data from the host to the slave device; and the DataTransmissionObject (DTO), which is used to transmit handshake messages, data and status information from the slave device to the host. Each of these message objects is assigned a unique CAN ID. Messages that are returned from the slave as a message to a command are called CommandReturnMessages (CRM).

A Command Receive Object is a CAN message consisting of eight bytes. The first byte of a CRO is the command code, followed by the command counter byte. The command counter is generated for reference by the host to make sure that the CRM returned by a slave device corresponds to the correct host command. The rest of the message builds the parameter and data fields. The structure is as follows:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMD</td>
<td>CTR</td>
<td>Parameter and Data Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A DataTransmissionObject has a PacketID (PID) as the first byte. This PID determines how the rest of the message is interpreted. CCP differentiates between three types of DTOs:

<table>
<thead>
<tr>
<th>PID</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00—0xFD</td>
<td>Data Acquisition Message</td>
</tr>
<tr>
<td>0xFE</td>
<td>Event Message</td>
</tr>
<tr>
<td>0xFF</td>
<td>Command Return Message</td>
</tr>
</tbody>
</table>

Command Return Messages and Event Messages have the following structure:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>ERR</td>
<td>Parameter and Data Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the case of an Event Message, the **Counter** field does not contain valid data and must be ignored by the host. For Command Return Messages the Counter field must have the same value as the counter field of the corresponding CRO. The error field contains information about the error state. The **parameter** and **data** fields contain the data returned from the slave device to the host. Command Return Messages and Event Messages consist of eight bytes.

**Data Acquisition Messages** (DAQ Messages or DAMs) have a PID in the first byte, and the rest of the message contains data. DAMs may be shorter than eight bytes:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Parameter and Data Field</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Since the PIDs 0x00—0xFD are reserved for Data Acquisition Messages, a CCP slave device can send up to 253 different DAMs. Each DAQ message can transfer up to seven bytes of data. The number of DAQ Messages supported by a slave device depends on the device itself.

Data acquisition is performed through a CCP slave device by reading data from a device's memory and copying it into the data field of a DAQ message. So the CCP slave device keeps a list of entries for each DAM. These lists are called **ObjectDefinitionTables** (ODTs). Each ODT entry holds information about the memory address where data is stored inside the device and the size of the data to be sent. The data of the first ODT entry is placed in the first byte of the data field of the DAQ message. The data of the next entry is placed at the first free byte of the DAQ message, and so on.
Technical Support and Professional Services

Log in to your National Instruments ni.com User Profile to get personalized access to your services. Visit the following sections of ni.com for technical support and professional services:

- **Support**—Technical support at ni.com/support includes the following resources:
  - **Self-Help Technical Resources**—For answers and solutions, visit ni.com/support for software drivers and updates, a searchable KnowledgeBase, product manuals, step-by-step troubleshooting wizards, thousands of example programs, tutorials, application notes, instrument drivers, and so on. Registered users also receive access to the NI Discussion Forums at ni.com/forums. NI Applications Engineers make sure every question submitted online receives an answer.
  - **Standard Service Program Membership**—This program entitles members to direct access to NI Applications Engineers via phone and email for one-to-one technical support, as well as exclusive access to self-paced online training modules at ni.com/self-paced-training. All customers automatically receive a one-year membership in the Standard Service Program (SSP) with the purchase of most software products and bundles including NI Developer Suite. NI also offers flexible extended contract options that guarantee your SSP benefits are available without interruption for as long as you need them. Visit ni.com/ssp for more information.

For information about other technical support options in your area, visit ni.com/services, or contact your local office at ni.com/contact.

- **Training and Certification**—Visit ni.com/training for training and certification program information. You can also register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, National Instruments
Alliance Partner members can help. To learn more, call your local
NI office or visit ni.com/alliance.

You also can visit the Worldwide Offices section of ni.com/niglobal
to access the branch office Web sites, which provide up-to-date contact
information, support phone numbers, email addresses, and current events.
Glossary

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Prefix</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>milli</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>k</td>
<td>kilo</td>
<td>$10^{3}$</td>
</tr>
<tr>
<td>M</td>
<td>mega</td>
<td>$10^{6}$</td>
</tr>
</tbody>
</table>

Numbers

2MC (*.A2L) database file
See ASAM MCD 2MC.

A

A2L file
ECU device database file in ASAM MCD 2MC format.

address extension
An additional parameter to the address that may be used to switch between data of several memory banks.

API
Application Program Interface—A set of routines, protocols, and tools for building software applications.

arbitration ID
An 11- or 29-bit ID transmitted as the first field of a CAN frame. The arbitration ID determines the priority of the frame, and is normally used to identify the data transmitted in the frame.

ASAM
Association of Standardization of Automation and Measurement Systems.

ASAM MCD 2MC
ASAM MCD 2MC is a file interface standardized by ASAM which describes the internal ECU data, interfaces, and communication protocols. It contains all information about relevant data objects in the ECU like Characteristic variables (parameters, characteristic curves, and maps), real/virtual measurement variables, and variant dependencies. For each of these objects information is needed, such as storage address, record layout, data type, and conversion rules to convert the data into their physical units.
Glossary

B

baudrate A user-defined property which provides the baud rate at which communication will occur. For more information, refer to the Interface Properties dialog in MAX, or the NI-CAN Hardware and Software Manual. The baud rate is originally set within MAX.

byte order The byte order refers to which bytes are most significant in multi-byte data types. The term describes the order in which a sequence of bytes is stored in computer memory.

C

calibration data page A portion of the ECU memory containing data that controls the behavior of the ECU.

CAN Controller Area Network. The Controller Area Network (CAN) is a joint development of Robert Bosch GmbH and Intel Corporation. CAN is used in many high-end automotive control systems, like engine management, as well as in industrial control systems. Controller chips for CAN are available from various semiconductor manufacturers.

CCP CAN Calibration Protocol.

CCP master The CCP master device (host) is a calibration/monitoring tool for initiating data transfers on the CAN by sending commands to slave devices.

CCP slave Typically an ECU which communicates through CCP with the CCP master.

Characteristic A Characteristic is a memory area within the ECU which defines the behavior of a control subsystem. Calibration is a process to optimize the Characteristic. A Characteristic can be represented by a single value (parameter), a one-dimensional array of values (curve), or a two-dimensional array of values (map).

Checksum DLL A Dynamic Link Library which implements a function to calculate a checksum over a given data block.

Command Receive Object (CRO) A Command Receive Object (CRO) is sent from the CCP master device to one of the slave devices. The slave device answers with a Data Transmission Object (DTO) containing a Command Return Message (CRM).
Controller Area Network  See CAN.

CRM  Command Return Message—A CCP communication object used to send commands and data from a host device to a slave device. The CRO is 8 bytes wide, consisting of a Command byte, a Command Counter byte, and a 6-byte parameter/data field.

CRO  CommandReceiveObject—A CCP communication object used to send commands and data from a host device to a slave device. The CRO is 8 bytes wide, consisting of a Command byte, a Command Counter byte, and a 6-byte parameter/data field.

CRO ID  CAN identifier of the Command Receive Object (CRO)

D

DAQ  Data Acquisition.

DAQ channel  A single DAQ Measurement entry in a DAQ list.

DAQ list  A list of DAQ channels that is transmitted by the ECU.

DAQ mode  Data acquisition mode.

Data Transfer Object  A message sent from the slave device to the master device (Command Return Message, Event Message, or Data Acquisition Message).

database task  A task reference handle to the selected ASAM MCD 2MC database file.

DLL  Dynamic Link Library.

DTO  See Data Transfer Object.

DTO ID  CAN identifier of the DTO.
<table>
<thead>
<tr>
<th><strong>E</strong></th>
<th><strong>M</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ECU</td>
<td>Master ID</td>
</tr>
<tr>
<td>Electronic Control Unit—An electronic device with a central processing unit performing programmed functions with its peripheral circuitry.</td>
<td>A 6-byte string identifying the CCP master device.</td>
</tr>
<tr>
<td>ECU M&amp;C Channel functions</td>
<td>Measurement</td>
</tr>
<tr>
<td>The part of the ECU M&amp;C Toolkit API that you use to read and write channels.</td>
<td>See DAQ.</td>
</tr>
<tr>
<td>A Characteristic or Measurement channel consists of one more floating-point values in physical units (such as Volts, rpm, km/h, °C, and so on) that is converted to/from a raw value in measurement hardware. The ECU M&amp;C API Read and Write functions provide access to Characteristic or Measurement channels. When a CAN message is received, ECU M&amp;C Toolkit converts raw fields in the message into physical units, which you then obtain using the ECU M&amp;C API Read function. When you call a ECU M&amp;C API Write function, you provide floating-point values in physical units, which ECU M&amp;C Toolkit converts into raw fields and transmits as a CAN message based on the CCP protocol.</td>
<td>Measurement task</td>
</tr>
<tr>
<td>ECU reference</td>
<td>Memory Transfer Address</td>
</tr>
<tr>
<td>Reference handle to a selected ECU.</td>
<td>Address pointer in the ECU that holds the source/target address for data sent or received via CCP. The address extension depends on the slave controller's organization and may identify a switchable memory bank or a memory segment.</td>
</tr>
<tr>
<td>ECU task</td>
<td>MTA</td>
</tr>
<tr>
<td>See ECU reference.</td>
<td>See Memory Transfer Address</td>
</tr>
</tbody>
</table>
Glossary

0
ODT
Object Descriptor Table—A list of elements (variables) used for organization of data acquisition (DAQ).

P
PID
PacketID—The first byte of a DTO corresponding to the ODT to which the DTO is assigned. The values for DAQ list PIDs range from 0x00–0xFD. The PIDs 0xFE and 0xFF are reserved for Event Messages and Command Return Messages.

Prescaler
A factor defined to allow reduction of the desired transmission rate. The prescaler is applied to the Event Channel. The prescaler value factor must be greater than or equal to 1.

S
SeedKey DLL
A Dynamic Link Library that implements a function to calculate a key to a given seed to unlock access to ECU resources.

slave device identifier
An ECU-specific array of bytes used by the master device to identify the ECU.

Station Address
A property which specifies an address to generate a logical point-to-point connection with a selected slave station for the master-slave command protocol. One ECU may support several station addresses.

T
task reference
An identifier returned as an output parameter of Database, ECU or Measurement initialization functions.
Index

A

accessing Characteristics, 4-7
activating the ECU toolkit
   home computer use, 2-4
   moving software after installation, 2-4
   online activation, 2-4
   privacy policy, 2-4
   procedure, 2-2
   terms defined, 2-3
   volume licensing, 2-4
activating your software, xv
additional programming topics, 4-16
   generic CCP functions, 4-17
   generic XCP functions, 4-18
   Get Names, 4-16
   seed and key algorithm, 4-19
   Set/Get Properties, 4-16
application development
   on CompactRIO or R Series using NI 985x or NI 986x C Series module, 3-4
ASAM definition, 1-1
ASAM MCD 2MC
   communication properties
      Baudrate, 4-6
      CRO ID, 4-5
      DTO ID, 4-5
      Station Address, 4-5
      with CAN, 4-5
      with UDP or TCP, 4-6
   overview, 1-1

B

basic programming model, 4-3
   Characteristic Read and Write, 4-7
   communication (figure), 4-4
   ECU Close, 4-7

   ECU Connect, 4-6
   ECU Disconnect, 4-7
   ECU Open, 4-5
   Measurement tasks, 4-9
   Baudrate (property), 4-6

C

C functions
   list of functions, 6-2
   mcBuildChecksum, 6-6
   mcCalculateChecksum, 6-10
   mcCCPActionService, 6-12
   mcCCPDiagService, 6-14
   mcCCPGetActiveCalPage, 6-16
   mcCCPGetResult, 6-17
   mcCCPGetSessionStatus, 6-18
   mcCCPGetVersion, 6-19
   mcCCPMoveMemory, 6-20
   mcCCPSelectCalPage, 6-22
   mcCCPSetSessionStatus, 6-23
      options (table), 6-23
   mcCharacteristicRead, 6-25
   mcCharacteristicReadSingleValue, 6-26
   mcCharacteristicWrite, 6-28
   mcCharacteristicWriteSingleValue, 6-29
   mcClearMemory, 6-31
   mcConversionCreate, 6-32
   mcDAQClear, 6-34
   mcDAQInitialize, 6-35
   mcDAQListInitialize, 6-38
   mcDAQRead, 6-40
   mcDAQReadTimestamped, 6-43
   mcDAQStartStop, 6-46
   mcDAQWrite, 6-48
   mcDatabaseClose, 6-50
   mcDatabaseOpen, 6-51
Index

mcDatabaseOpenEx, 6-52
mcDoubleToText, 6-53
mcDownload, 6-55
mcECUConnect, 6-57
mcECUCreate, 6-58
mcECUDeselect, 6-62
mcECUDisconnect, 6-63
mcECUSelectEx, 6-64
mcEventCreate, 6-67
mcGeneric, 6-68
mcGetNames, 6-70
mcGetNamesLength, 6-73
mcGetProperty, 6-76
   options (table), 6-77
mcMeasurementCreate, 6-96
mcMeasurementRead, 6-98
mcMeasurementWrite, 6-99
mcProgram, 6-100
mcProgramReset, 6-102
mcProgramStart, 6-103
mcSetProperty, 6-104
   Characteristic-specific options
      (table), 6-115
   DAQ-specific options (table), 6-114
   ECU-specific options (table), 6-106
   Measurement-specific options
      (table), 6-115
mcStatusToString, 6-116
   return codes (table), 6-116
mcTextToDouble, 6-118
mcUpload, 6-120
mcXCPCopyCalPage, 6-122
mcXCPGetCalPage, 6-124
mcXCPGetId, 6-126
mcXCPGetStatus, 6-128
mcXCPProgramPrepare, 6-132
mcXCPProgramVerify, 6-134
mcXCPSetCalPage, 6-136
mcXCPSetRequest, 6-138
mcXCPSetSegmentMode, 6-140
CAN calibration protocol (CCP)
   overview, 1-2, A-1
   version, 1-2
CAN overview, A-1
CCP
   functions, 4-3
   overview, A-1
   protocol definition, A-3
   scope, A-2
Channel functions, 4-2
Characteristic Read and Write, 4-7
Characteristics
   accessing, 4-7
   reading, 4-8
   writing, 4-8
checksum algorithm, 4-21
   definition, 4-21, 4-22
   for VxWorks targets, 4-23
      example, 4-23
choosing programming languages, 3-1
CompactRIO
   application development on using
      NI 985x or NI 986x C Series
      module, 3-4
computer ID, xvi
CRO ID (property), 4-5

D
deactivating a product, xvii
debugging an application, 3-6
definition of activation terms, 2-3
developing an application, 3-1
diagnostic tools (NI resources), B-1
documentation
   NI resources, B-1
      related documentation, xiii
drivers (NI resources), B-1
DTO ID (property), 4-5
Index

E

ECU API
  C, 6-1
  LabVIEW, 5-1
ECU Characteristics
  definition, 4-2
  overview, 1-4
ECU Close, 4-7
ECU Connect, 4-6
ECU databases, 1-4
ECU Disconnect, 4-7
ECU M&C API
  additional programming topics, 4-16
  architecture (figure), 4-1
  CCP functions overview, 4-3
  Channel functions, 4-2
  structure, 4-1
  XCP functions overview, 4-3
ECU Measurements
  DAQ Clear, 4-13
  DAQ Read, 4-11
  DAQ Start Stop, 4-10
  DAQ Write, 4-12
  definition, 4-2
  DTO ID, 4-10
  ECU DAQ Initialize, 4-10
  ECU reference handle, 4-10
  flowchart (figure), 4-9
  list, 4-10
  mode, 4-10
  overview, 4-9
  sample rate, 4-10
ECU Open, 4-5
ECU toolkit
  activation, 2-2
  API overview, 4-1
  basic programming model, 4-3
  Characteristics, 1-4
databases
  ASAM MCD 2MC, 1-4
  ASAP, 1-4
  definition, 1-1
  hardware and software requirements, 2-10
  installation, 2-1
  introduction, 1-1
  LabVIEW RT, 2-5
  license management, 2-1
  Measurements, 1-4
examples (NI resources), B-1

F

FTP transfers (table), 2-6
FTP with LabVIEW, 2-9
FTP with LabVIEW RT graphical file transfer utility, 2-7
FTP with web browsers, 2-7

G
generic CCP functions, 4-17
generic XCP functions, 4-18
Get Names, 4-16

H

help, technical support, B-1
home software use, 2-4

I

instrument drivers (NI resources), B-1

K

KnowledgeBase, B-1
Index

L

LabVIEW

list of VIs, 5-1
MC Build Checksum.vi, 5-5
MC Calc Checksum.vi, 5-8
MC CCP Action Service.vi, 5-11
MC CCP Diag Service.vi, 5-13
MC CCP Generic.vi, 5-93
MC CCP Get Active Cal Page.vi, 5-15
MC CCP Get Result.vi, 5-17
MC CCP Get Session Status.vi, 5-19
MC CCP Get Version.vi, 5-21
MC CCP Move Memory.vi, 5-23
MC CCP Select Cal Page.vi, 5-25
MC CCP Set Session Status.vi, 5-27
options (table), 5-28
MC Characteristic Read Single Value.vi, 5-31
MC Characteristic Read.vi, 5-29
options (table), 5-30
MC Characteristic Write Single Value.vi, 5-35
MC Characteristic Write.vi, 5-33
options (table), 5-34
MC Clear Memory.vi, 5-37
MC Conversion Create.vi, 5-39
MC DAQ Clear.vi, 5-41
MC DAQ Initialize.vi, 5-43
MC DAQ List Initialize.vi, 5-46
MC DAQ Read.vi, 5-49
MC DAQ Start Stop.vi, 5-55
MC DAQ Write.vi, 5-57
MC Database Close.vi, 5-60
MC Database Create.vi, 5-62
MC Database Open.vi, 5-64
MC Double To Text.vi, 5-66
MC Download.vi, 5-68
MC ECU Close.vi, 5-70
MC ECU Connect.vi, 5-72
MC ECU Create.vi, 5-74
MC ECU Deselect.vi, 5-79
MC ECU Disconnect.vi, 5-81
MC ECU Open.vi, 5-83
MC ECU Select.vi, 5-87
MC Event Create.vi, 5-91
MC Get Names.vi, 5-95
MC Get Property.vi, 5-99
poly output values (table), 5-101
MC Measurement Create.vi, 5-137
MC Measurement Read.vi, 5-139
MC Measurement Write.vi, 5-141
MC Program Reset.vi, 5-145
MC Program Start.vi, 5-147
MC Program.vi, 5-143
MC Set Property.vi, 5-149
Characteristic-specific input values (table), 5-167
DAQ-specific poly input values (table), 5-165
ECU-specific poly input values (table), 5-151
Measurement-specific input values (table), 5-167
MC Text To Double.vi, 5-168
MC Upload.vi, 5-170
MC XCP Copy Cal Page.vi, 5-172
MC XCP Get Cal Page.vi, 5-174
MC XCP Get ID.vi, 5-176
MC XCP Get Status.vi, 5-178
MC XCP Program Prepare.vi, 5-183
MC XCP Program Verify.vi, 5-185
MC XCP Set Cal Page.vi, 5-188
MC XCP Set Request.vi, 5-190
MC XCP Set Segment Mode.vi, 5-193
LabVIEW Real-Time (RT) configuration, 2-5
CompactRIO system, 2-5
DOS prompt, 2-6
FTP transfers (table), 2-6
LabVIEW, 2-9
LabVIEW RT graphical file transfer utility, 2-7
Index

NI-CAN on PXI RT system, 2-5
NI-XNET on PXI RT system, 2-5
PXI system, 2-5
web browsers, 2-7
license management overview, 2-1
list of C functions, 6-2
list of LabVIEW VIs, 5-1

M
MC Build Checksum.vi, 5-5
MC Calc Checksum.vi, 5-8
MC CCP Action Service.vi, 5-11
MC CCP Diag Service.vi, 5-13
MC CCP Generic.vi, 5-93
MC CCP Get Active Cal Page.vi, 5-15
MC CCP Get Result.vi, 5-17
MC CCP Get Session Status.vi, 5-19
MC CCP Get Version.vi, 5-21
MC CCP Move Memory.vi, 5-23
MC CCP Select Cal Page.vi, 5-25
MC CCP Set Session Status.vi, 5-27
options (table), 5-28
MC Characteristic Read Single Value.vi, 5-31
MC Characteristic Read.vi, 5-29
options (table), 5-30
MC Characteristic Write Single Value.vi, 5-35
MC Characteristic Write.vi, 5-33
options (table), 5-34
MC Clear Memory.vi, 5-37
MC Conversion Create.vi, 5-39
MC DAQ Clear.vi, 5-41
MC DAQ Initialize.vi, 5-43
MC DAQ List Initialize.vi, 5-46
MC DAQ Read.vi, 5-49
MC DAQ Start Stop.vi, 5-55
MC DAQ Write.vi, 5-57
MC Database Close.vi, 5-60
MC Database Create.vi, 5-62
MC Database Open.vi, 5-64
MC Double To Text.vi, 5-66
MC Download.vi, 5-68
MC ECU Close.vi, 5-70
MC ECU Connect.vi, 5-72
MC ECU Create.vi, 5-74
MC ECU Deselect.vi, 5-79
MC ECU Disconnect.vi, 5-81
MC ECU Open.vi, 5-83
MC ECU Select.vi, 5-87
MC Event Create.vi, 5-91
MC Get Names.vi, 5-95
MC Get Property.vi, 5-99
poly output values (table), 5-101
MC Measurement Create.vi, 5-137
MC Measurement Read.vi, 5-139
MC Measurement Write.vi, 5-141
MC Program Reset.vi, 5-145
MC Program Start.vi, 5-147
MC Program.vi, 5-143
MC Set Property.vi, 5-149
Characteristic-specific input values
(options table), 5-167
DAQ-specific poly input values
(options table), 5-165
ECU-specific poly input values
(options table), 5-151
Measurement-specific input values
(options table), 5-167
MC Text To Double.vi, 5-168
MC Upload.vi, 5-170
MC XCP Copy Cal Page.vi, 5-172
MC XCP Get Cal Page.vi, 5-174
MC XCP Get ID.vi, 5-176
MC XCP Get Status.vi, 5-178
MC XCP Program Prepare.vi, 5-183
MC XCP Program Verify.vi, 5-185
MC XCP Set Cal Page.vi, 5-188
MC XCP Set Request.vi, 5-190
MC XCP Set Segment Mode.vi, 5-193
mcBuildChecksum, 6-6
mcCCPActionService, 6-12
mcCCPCalculateChecksum, 6-10
mcCCPDiagService, 6-14
mcCCPGetActiveCalPage, 6-16
mcCCPGetResult, 6-17
mcCCPGetSessionStatus, 6-18
mcCCPGetVersion, 6-19
mcCCPMoveMemory, 6-20
mcCCPSelectCalPage, 6-22
mcCCPSetSessionStatus, 6-23
   options (table), 6-23
mcCharacteristicRead, 6-25
mcCharacteristicReadSingleValue, 6-26
mcCharacteristicWrite, 6-28
mcCharacteristicWriteSingleValue, 6-29
mcClearMemory, 6-31
mcConversionCreate, 6-32
mcDAQClear, 6-34
mcDAQInitialize, 6-35
mcDAQListInitialize, 6-38
mcDAQRead, 6-40
mcDAQReadTimestamped, 6-43
mcDAQStartStop, 6-46
mcDAQWrite, 6-48
mcDatabaseClose, 6-50
mcDatabaseOpen, 6-51
mcDatabaseOpenEx, 6-52
mcDoubleToText, 6-53
mcDownload, 6-55
mcECUConnect, 6-57
mcECUCreate, 6-58
mcECUDeselect, 6-62
mcECUDisconnect, 6-63
mcECUSelectEx, 6-64
mcEventCreate, 6-67
mcGeneric, 6-68
mcGetNames, 6-70
mcGetNamesLength, 6-73
mcGetProperty, 6-76
   options (table), 6-77
mcMeasurementCreate, 6-96
mcMeasurementRead, 6-98
mcMeasurementWrite, 6-99
mcProgram, 6-100
mcProgramReset, 6-102
mcProgramStart, 6-103
mcSetProperty, 6-104
   Characteristic-specific options
   (table), 6-115
   DAQ-specific options (table), 6-114
   ECU-specific options (table), 6-106
   Measurement-specific options
   (table), 6-115
mcStatusToString, 6-116
   return codes (table), 6-116
mcTextToDouble, 6-118
mcUpload, 6-120
mcXCPCopyCalPage, 6-122
mcXCPGetCalPage, 6-124
mcXCPGetId, 6-126
mcXCPGetStatus, 6-128
mcXCPProgramPrepare, 6-132
mcXCPProgramVerify, 6-134
mcXCPSetCalPage, 6-136
mcXCPSetRequest, 6-138
mcXCPSetSegmentMode, 6-140
measurement and calibration databases, 1-4

N
National Instruments support and
services, B-1
NI Activation Wizard, xv
NI support and services, B-1

O
online software activation, 2-4

P
privacy policy, 2-4
programming examples (NI resources), B-1
programming languages
   LabVIEW, 3-1
   LabWindows/CVI, 3-1
   other, 3-3
   Visual C++, 3-2

R
R Series
   application development on using
      NI 985x or NI 986x C Series
      module, 3-4
   reactivation on another system, 2-4
   reading Characteristics, 4-8
   related documentation, xiii
RT configuration
   DOS prompt, 2-6
   FTP transfers (table), 2-6
   LabVIEW, 2-9
   LabVIEW RT graphical file transfer
      utility, 2-7
   web browsers, 2-7

S
   sample rate greater than 0, 4-12
      read sample timing (figure), 4-12
   sample rate=0, 4-11
      read sample timing (figure), 4-11
   seed and key algorithm, 4-19
      definition, 4-19
      example, 4-20
      for VxWorks targets, 4-23
      example, 4-23
   serial number, finding, xvi
   Set/Get Properties, 4-16
   setting up an ECU Measurement
      DAQ Clear, 4-13
      DAQ Read, 4-11
      DAQ Start Stop, 4-10
      DAQ Write, 4-12
   DTO ID, 4-10
   ECU DAQ Initialize, 4-10
   ECU reference handle, 4-10
      flowchart (figure), 4-9
      list, 4-10
      mode, 4-10
      overview, 4-9
      sample rate, 4-10
   software
      activating, xv
      evaluating, xvi
      moving after activation, xvii
   software (NI resources), B-1
   Station Address (property), 4-5
   structure of ECU M&C API, 4-1
   support, technical, B-1

T
   task (concept), 1-4
   technical support, B-1
   training and certification (NI resources), B-1
   troubleshooting (NI resources), B-1

U
   using with FTP, 2-6

V
   volume licensing program, 2-4

W
   Web resources, B-1
   Windows Guest accounts, xvii
   writing Characteristics, 4-8

X
   XCP, functions overview, 4-3