CAN

ECU Measurement and Calibration Toolkit User Manual

July 2014
371601N-01

NATIONAL INSTRUMENTS™

Worldwide Technical Support and Product Information

ni.com

Worldwide Offices

Visit ni.com/niglobal to access the branch office websites, 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 website at ni.com/info and enter the Info Code feedback.

Legal Information

Limited Warranty

This document is provided ‘as is’ and is subject to being changed, without notice, in future editions. For the latest version, refer to `ni.com/manuals`. NI reviews this document carefully for technical accuracy; however, NI MAKES NO EXPRESS OR IMPLIED WARRANTIES AS TO THE ACCURACY OF THE INFORMATION CONTAINED HEREIN AND SHALL NOT BE LIABLE FOR ANY ERRORS.

NI warrants that its hardware products will be free of defects in materials and workmanship that cause the product to fail to substantially conform to the applicable NI published specifications for one (1) year from the date of invoice.

For a period of ninety (90) days from the date of invoice, NI warrants that (i) its software products will perform substantially in accordance with the applicable documentation provided with the software and (ii) the software media will be free from defects in materials and workmanship.

If NI receives notice of a defect or non-conformance during the applicable warranty period, NI will, in its discretion: (i) repair or replace the affected product, or (ii) refund the fees paid for the affected product. Repaired or replaced Hardware will be warranted for the remainder of the original warranty period or ninety (90) days, whichever is longer. If NI elects to repair or replace the product, NI may use new or refurbished parts or products that are equivalent to new in performance and reliability and are at least functionally equivalent to the original part or product.

You must obtain an RMA number from NI before returning any product to NI. NI reserves the right to charge a fee for examining and testing Hardware not covered by the Limited Warranty.

This Limited Warranty does not apply if the defect of the product resulted from improper or inadequate maintenance, installation, repair, or calibration (performed by a party other than NI); unauthorized modification; improper environment; use of an improper hardware or software key; improper use or operation outside of the specification for the product; improper voltages; accident, abuse, or neglect; or a hazard such as lightning, flood, or other act of nature.

THE REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND THE CUSTOMER’S SOLE REMEDIES, AND SHALL APPLY EVEN IF SUCH REMEDIES FAIL OF THEIR ESSENTIAL PURPOSE.

EXCEPT AS EXPRESSLY SET FORTH HEREIN, PRODUCTS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND AND NI DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, WITH RESPECT TO THE PRODUCTS, INCLUDING ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE OR NON-INFRINGEMENT, AND ANY WARRANTIES THAT MAY ARISE FROM USAGE OF TRADE OR COURSE OF DEALING. NI DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE OF OR THE RESULTS OF THE USE OF THE PRODUCTS IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE. NI DOES NOT WARRANT THAT THE OPERATION OF THE PRODUCTS WILL BE UNINTERRUPTED OR ERROR FREE.

In the event that you and NI have a separate signed written agreement with warranty terms covering the products, then the warranty terms in the separate agreement shall control.

Copyright

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 information on including legal information in installers built with NI products.

U.S. Government Restricted Rights

If you are an agency, department, or other entity of the United States Government (“Government”), the use, duplication, reproduction, release, modification, disclosure or transfer of the technical data included in this manual is governed by the Restricted Rights provisions under Federal Acquisition Regulation 52.227-14 for civilian agencies and Defense Federal Acquisition Regulation Supplement Section 252.227-7014 and 252.227-7015 for military agencies.

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.

TETRIX by Pitsco is a trademark of Pitsco, Inc.

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.

Linux® 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

YOU ARE ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY AND RELIABILITY OF THE PRODUCTS WHENEVER THE PRODUCTS ARE INCORPORATED IN YOUR SYSTEM OR APPLICATION, INCLUDING THE APPROPRIATE DESIGN, PROCESS, AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

PRODUCTS ARE NOT DESIGNED, MANUFACTURED, OR TESTED FOR USE IN LIFE OR SAFETY CRITICAL SYSTEMS, HAZARDOUS ENVIRONMENTS OR ANY OTHER ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, INCLUDING IN THE OPERATION OF NUCLEAR FACILITIES; AIRCRAFT NAVIGATION; AIR TRAFFIC CONTROL SYSTEMS; LIFE SAVING OR LIFE SUSTAINING SYSTEMS OR SUCH OTHER MEDICAL DEVICES; OR ANY OTHER APPLICATION IN WHICH THE FAILURE OF THE PRODUCT OR SERVICE COULD LEAD TO DEATH, PERSONAL INJURY, SEVERE PROPERTY DAMAGE OR ENVIRONMENTAL HARM (COLLECTIVELY, “HIGH-RISK USES”). FURTHER, PRUDENT STEPS MUST BE TAKEN TO PROTECT AGAINST FAILURES, INCLUDING PROVIDING BACK-UP AND SHUT-DOWN MECHANISMS. NI EXPRESSLY DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS OF THE PRODUCTS OR SERVICES FOR HIGH-RISK USES.

Contents

Chapter 5
ECU M&C API for LabVIEW

Chapter 6
ECU M&C API for C

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:

- *ANSI/ISO Standard 11898-1993, Road Vehicles—Interchange of Digital Information—Controller Area Network (CAN) for High-Speed Communication*
- *CAN Specification Version 2.0*, 1991, Robert Bosch GmbH., Postfach 106050, D-70049 Stuttgart 1
- *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.

1 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.

2

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

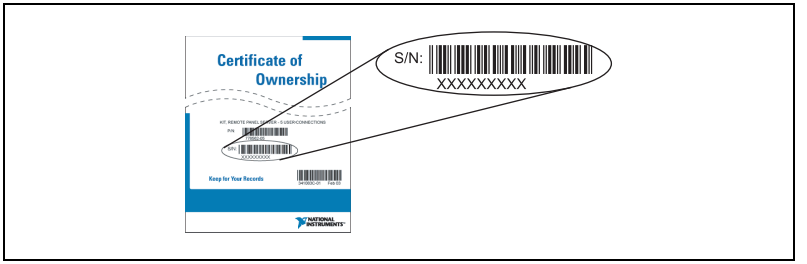
The ECU M&C Toolkit must be activated before using it, in accordance with its license agreement. To activate the ECU M&C Toolkit, you must first purchase a license. For information on purchasing licenses, contact your local National Instruments sales representative or visit `ni.com`.

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 1. Definition of Activation Terms

Serial Number	A 9-character, alphanumeric string that uniquely identifies your purchase of a single copy of software, included in your software kit on your *Certificate of Ownership*. The serial number for hardware products is printed either on the product box or on the device.
Computer ID or **Device ID**	A 16-character ID that uniquely identifies your computer or NI hardware, generated during the activation process.
Activation Code	A 20-character code that enables NI software to run on your computer, based on your serial number and computer ID. You generate and install an activation code by completing the activation process.

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.

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 2-1. Example of FTP Transfer

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

Table 2-1. Example of FTP Transfer (Continued)

Command	Result
put c:\index.htm index.htm	Copy the file from the FTP client machine to the target.
dir	Verify the copied file on the target.
cd c:	Change directory back to the internal compact flash or hard drive.
quit	Disconnect from the FTP server.

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 2-2.

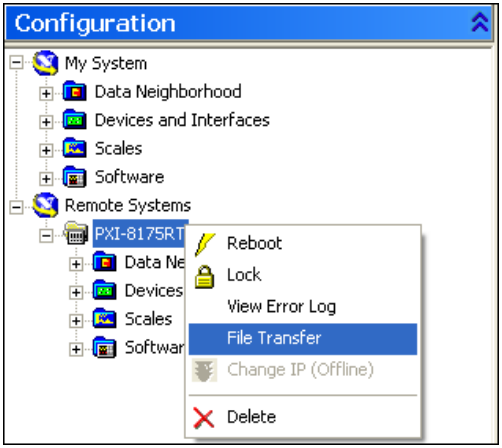


Figure 2-2. FTP Utility Access in MAX

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 2-3.

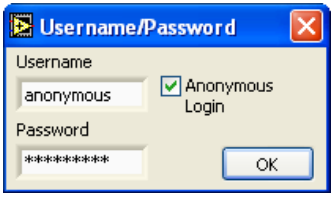


Figure 2-3. FTP Login Dialog Box

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 2-4 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.

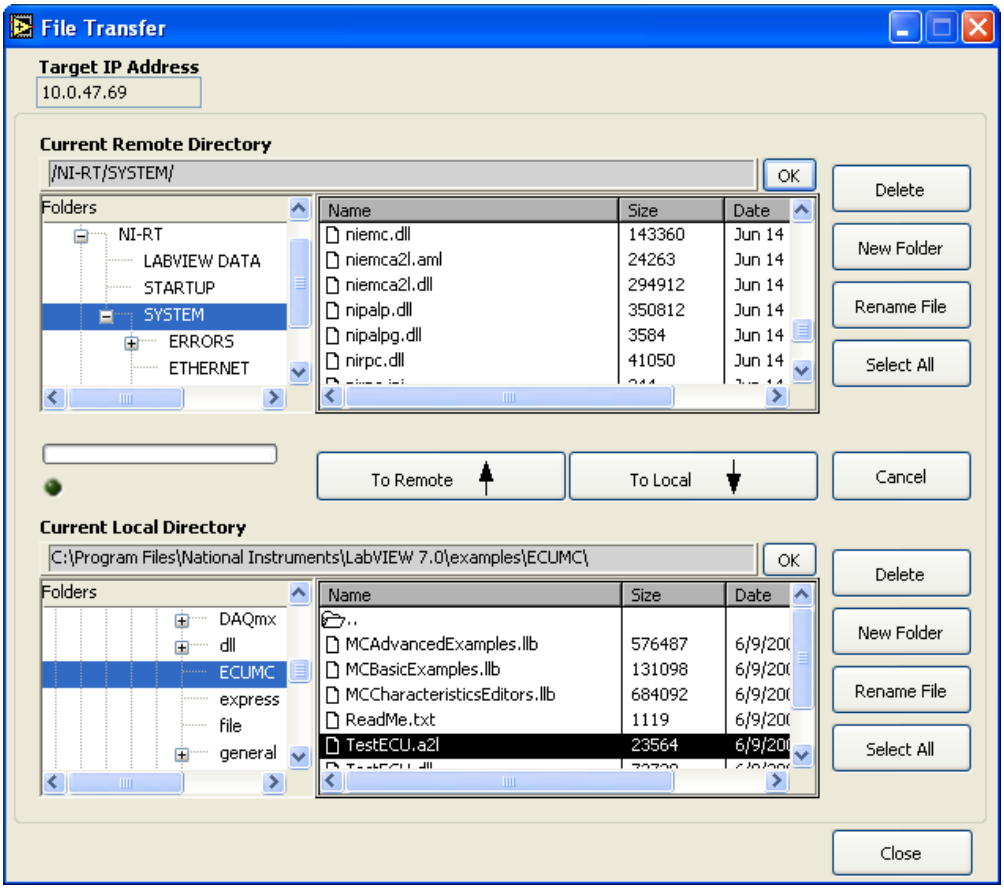


Figure 2-4. Transferring Files With the FTP Utility

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.

3

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:

```
#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);
```

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

To run a project on an FPGA target with an NI 985*x* 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 985*x* 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 985*x* C Series module, the interface name is based on the bitfile you use and the interface name you set. For example, `MyInterface@MyBitfile.lvbitx`, `CAN@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 986*x* 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 986*x* module as you would program on a Windows or PXI LabVIEW Real-Time target.

Debugging An Application

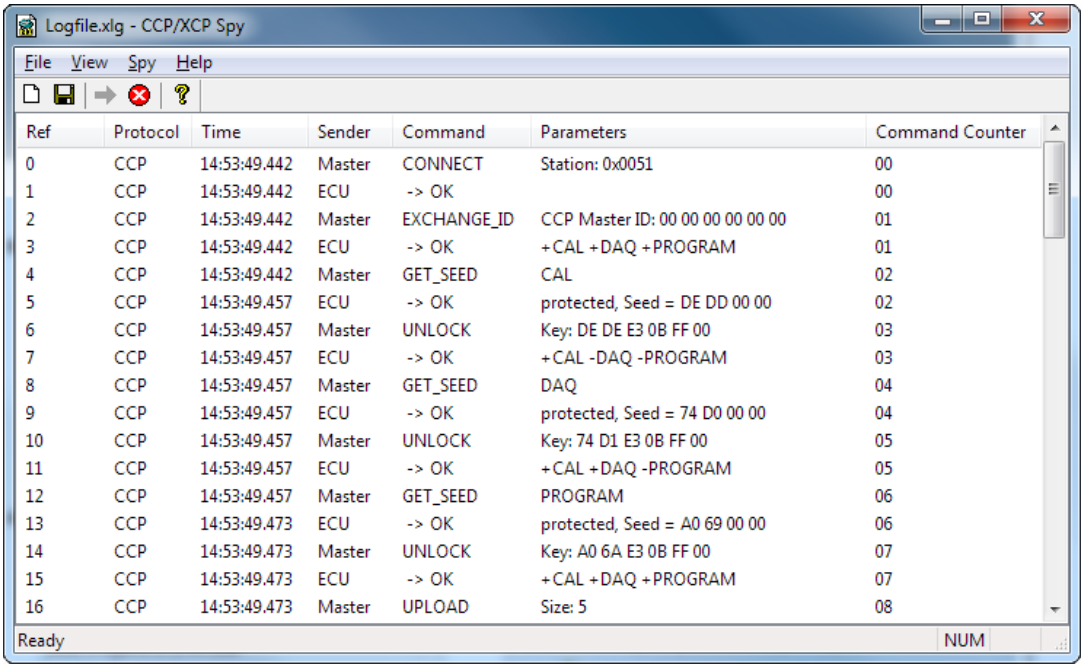
NI I/O Trace

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**.



Ref	Protocol	Time	Sender	Command	Parameters	Command Counter
0	CCP	14:53:49.442	Master	CONNECT	Station: 0x0051	00
1	CCP	14:53:49.442	ECU	-> OK		00
2	CCP	14:53:49.442	Master	EXCHANGE_ID	CCP Master ID: 00 00 00 00 00 00	01
3	CCP	14:53:49.442	ECU	-> OK	+CAL +DAQ +PROGRAM	01
4	CCP	14:53:49.442	Master	GET_SEED	CAL	02
5	CCP	14:53:49.457	ECU	-> OK	protected, Seed = DE DD 00 00	02
6	CCP	14:53:49.457	Master	UNLOCK	Key: DE DE E3 0B FF 00	03
7	CCP	14:53:49.457	ECU	-> OK	+CAL -DAQ -PROGRAM	03
8	CCP	14:53:49.457	Master	GET_SEED	DAQ	04
9	CCP	14:53:49.457	ECU	-> OK	protected, Seed = 74 D0 00 00	04
10	CCP	14:53:49.457	Master	UNLOCK	Key: 74 D1 E3 0B FF 00	05
11	CCP	14:53:49.457	ECU	-> OK	+CAL +DAQ -PROGRAM	05
12	CCP	14:53:49.457	Master	GET_SEED	PROGRAM	06
13	CCP	14:53:49.473	ECU	-> OK	protected, Seed = A0 69 00 00	06
14	CCP	14:53:49.473	Master	UNLOCK	Key: A0 6A E3 0B FF 00	07
15	CCP	14:53:49.473	ECU	-> OK	+CAL +DAQ +PROGRAM	07
16	CCP	14:53:49.473	Master	UPLOAD	Size: 5	08

Figure 3-1. CCP/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).

4

Using the ECU M&C API

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.

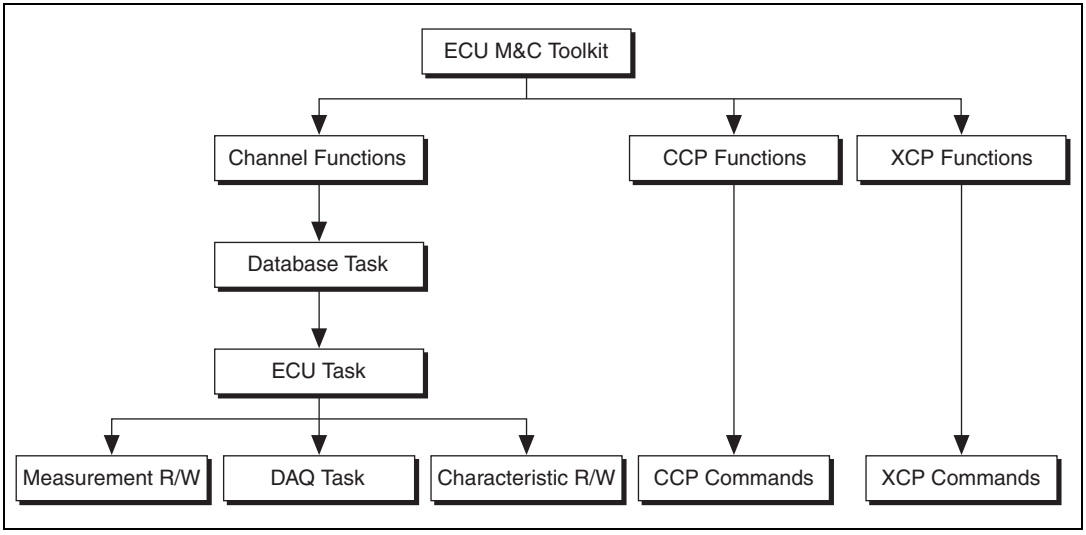


Figure 4-1. ECU Architectural Overview

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.

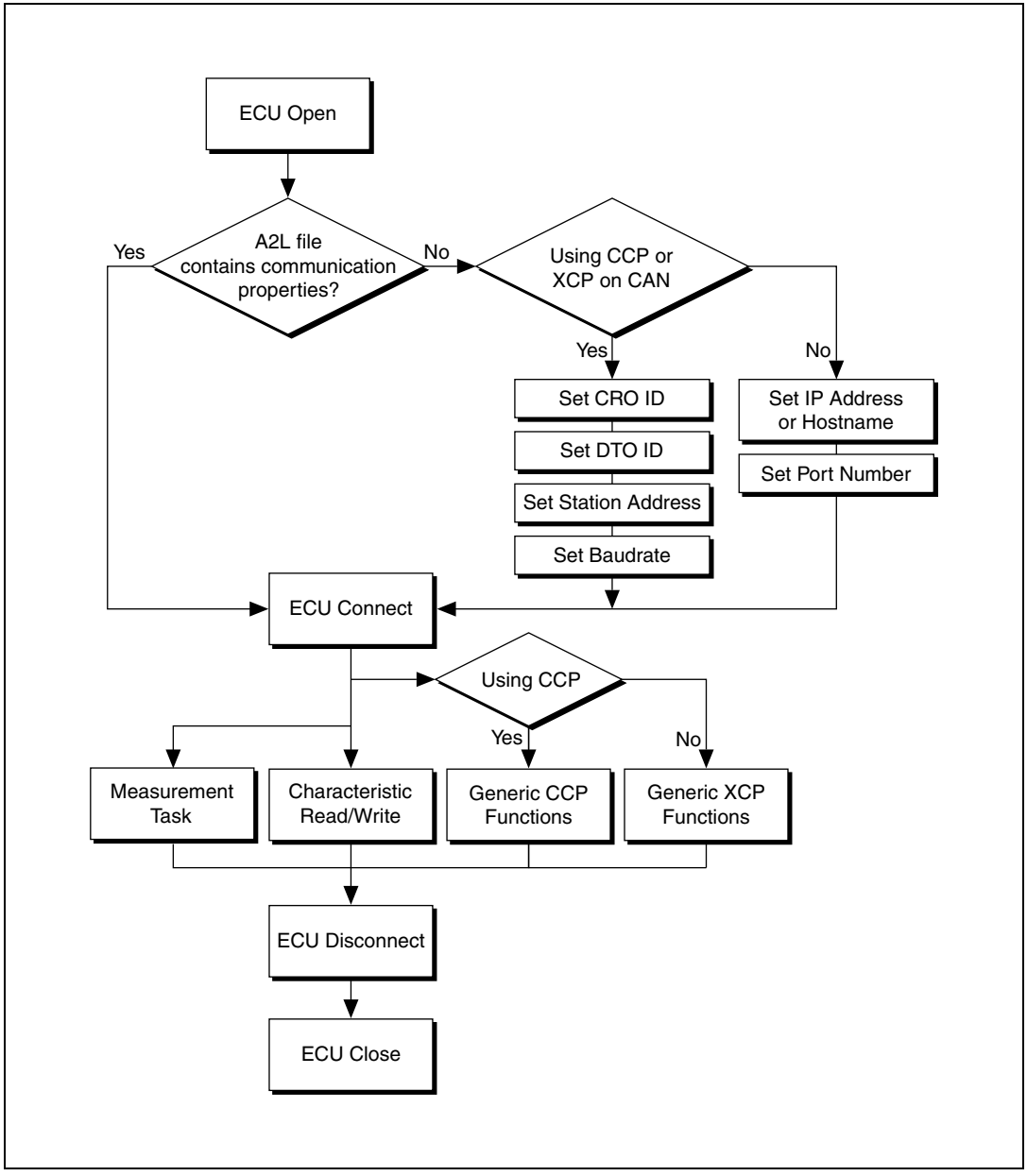


Figure 4-2. ECU Communication Decision Chart

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** (**C**ommand **R**eceive **O**bject) is used to send commands and data from the host to the slave device.

DTO ID

The **DTO ID** (**D**ata **T**ransmission **O**bject) 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.

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.

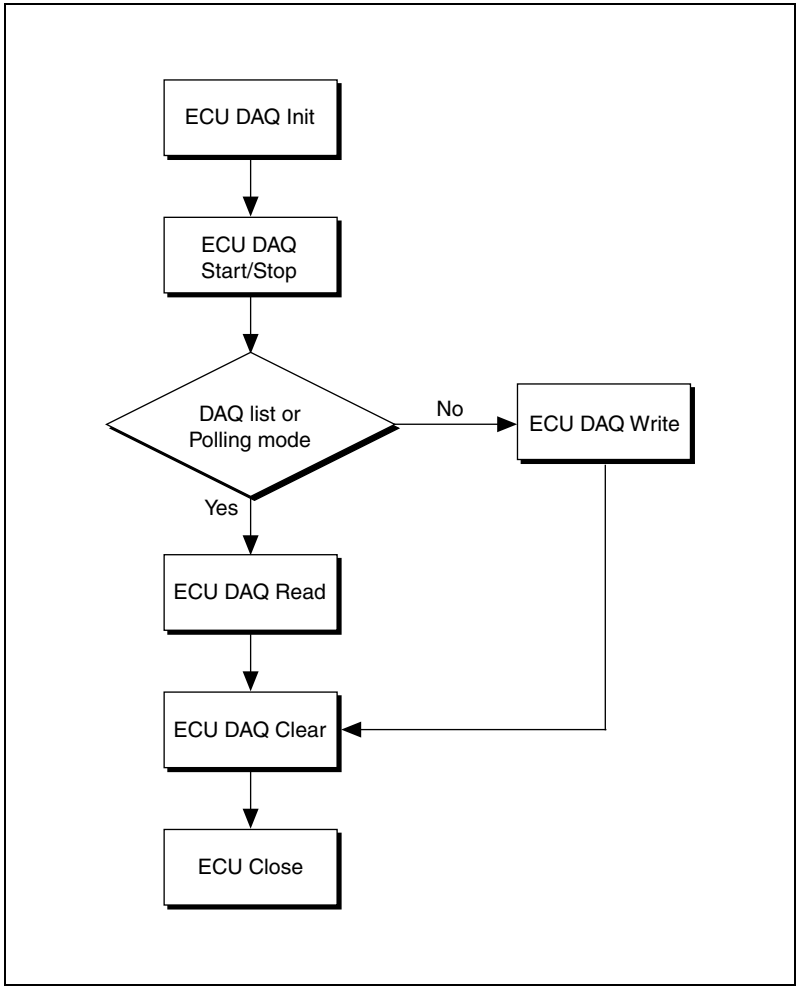


Figure 4-3. ECU Measurement Setup Flowchart

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** (**D**ata **T**ransmission **O**bject) 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.

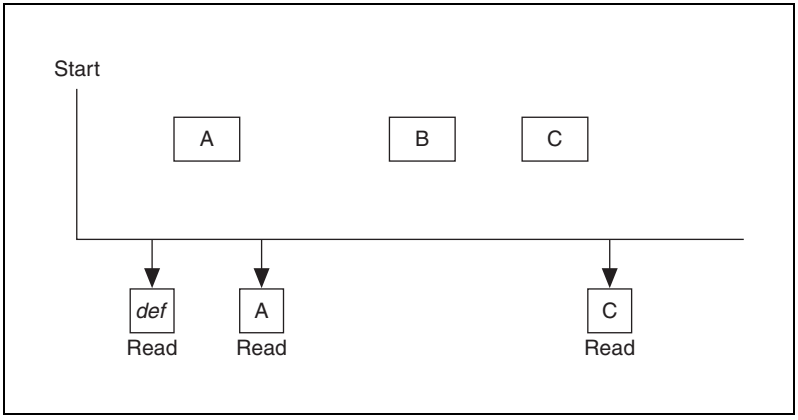


Figure 4-4. Example of Read With Sample Rate = 0

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. *def* represents the default value 0.

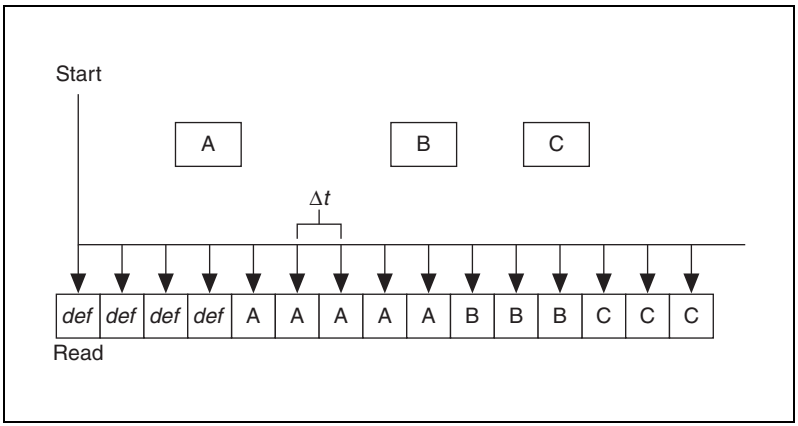


Figure 4-5. Example of Read With Sample Rate > 0

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.

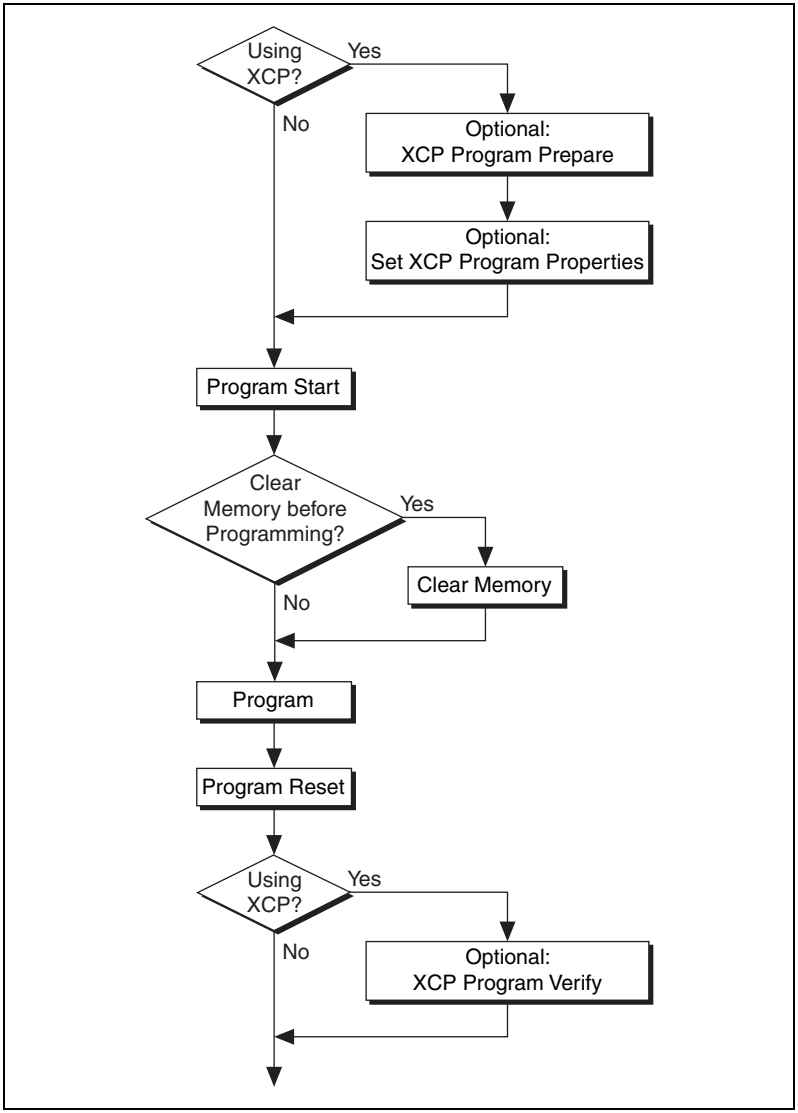


Figure 4-6. Memory Programming Process Decision Chart

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.

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**.

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 4-1. Overview of the CCP Commands with Related VIs and C Functions

CCP Command	LabVIEW VI Name	C Function Name
ACTION_SERVICE	**MC CCP Action Service.vi**	`mcCCPActionService`
BUILD_CHKSUM	**MC Build Checksum.vi**	`mcBuildChecksum`
CLEAR_MEMORY	**MC Clear Memory.vi**	`mcClearMemory`
DIAG_SERVICE	**MC CCP Diag Service.vi**	`mcCCPDiagService`
DNLOAD	**MC Download.vi**	`mcDownload`
GET_ACTIVE_CAL_PAGE	**MC CCP Get Active Cal Page.vi**	`mcCCPGetActiveCalPage`
GET_CCP_VERSION	**MC CCP Get Version.vi**	`mcCCPGetVersion`
GET_S_STATUS	**MC CCP Get Session Status.vi**	`mcCCPGetSessionStatus`
MOVE	**MC CCP Move Memory.vi**	`mcCCPMoveMemory`
PROGRAM	**MC Program.vi**	`mcProgram`
SELECT_CAL_PAGE	**MC CCP Select Cal Page.vi**	`mcCCPSelectCalPage`

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

CCP Command	LabVIEW VI Name	C Function Name
SET_S_STATUS	**MC CCP Set Session Status.vi**	mcCCPSetSessionStatus
UPLOAD	**MC Upload.vi**	mcUpload

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

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

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

XCP Command	LabVIEW VI Name	C Function Name
SET_SEGMENT_MODE	**MC XCP Set Segment Mode.vi**	`mcXCPSetSegmentMode`
UPLOAD	**MC Upload.vi**	`mcUpload`

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`

Parameter	Description
1	Pointer to the seed data, retrieved from the ECU `GET_SEED` command.
2	Seed data size in number of bytes.
3	Pointer to key data, returning the calculated.
4	Key data size in number of bytes.
5	Key data size in number of bytes.

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.
#ifdef __cplusplus
}
#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)`

Parameter	Description
1	Pointer to an array of ranges, stored in structures of type `TRange`.
2	Number of ranges stored in the array that parameter 1 points to.
3	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.
4	Length of actually calculated checksum (1...8).
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:

```
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.

```
/*
// 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 901*x* and 907*x* 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.

```
#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;
};


#ifdef __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")));
#ifdef __cplusplus
   }
#endif

#endif // __CCPECU_h__
```

5

ECU M&C API for LabVIEW

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 5-1. ECU M&C API VIs for LabVIEW

Function	Purpose
MC Build Checksum.vi	Calculates a checksum over a defined memory range within the ECU.
MC Calc Checksum.vi	Calculates the checksum of a data block in memory.
MC CCP Action Service.vi	Calls an implementation-specific action service on the ECU.

Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

Function	Purpose
MC CCP Diag Service.vi	Calls a diagnostic service on the ECU.
MC CCP Get Active Cal Page.vi	Retrieves the ECU Memory Transfer Address pointer to the calibration data page.
MC CCP Get Result.vi	Uploads requested data.
MC CCP Get Session Status.vi	Retrieves the current calibration status of the ECU.
MC CCP Get Version.vi	Retrieves the version of the CCP implemented in the ECU.
MC CCP Move Memory.vi	Moves a memory block on the ECU.
MC CCP Select Cal Page.vi	Sets the beginning of the calibration data page.
MC CCP Set Session Status.vi	Updates the ECU with the current state of the calibration session.
MC Characteristic Read.vi	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.
MC Characteristic Read Single Value.vi	Reads a value from a named Characteristic on the ECU which is identified by the ECU Reference handle.
MC Characteristic Write.vi	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.
MC Characteristic Write Single Value.vi	Writes a value to a named Characteristic on the ECU.
MC Clear Memory.vi	Clears the contents of a specified memory block.
MC Conversion Create.vi	Creates a signal conversion object in memory.
MC DAQ Clear.vi	Stops communication for the Measurement task and then clears the configuration.
MC DAQ Initialize.vi	Initializes a Measurement task for the specified Measurement channel list.
MC DAQ List Initialize.vi	Defines a DAQ list on a specific DAQ list number and initializes the Measurement task for the specified Measurement channel list.

Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

Function	Purpose
MC DAQ Read.vi	Reads samples from a Measurement task. Samples are obtained from received CAN messages.
MC DAQ Start Stop.vi	Starts or stops transmission of the DAQ lists for the specified Measurement task.
MC DAQ Write.vi	Writes samples to a Measurement task.
MC Database Close.vi	Closes a specified A2L Database.
MC Database Create.vi	Creates an A2L database in memory, for using the ECU M&C Toolkit VIs without access to a valid ASAM A2L file.
MC Database Open.vi	Opens a specified A2L Database.
MC Double to Text.vi	Converts a numerical value to a text string using an enumeration or range text type scaling.
MC Download.vi	Downloads data to an ECU.
MC ECU Close.vi	Closes the selected ECU and the associated A2L database.
MC ECU Connect.vi	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.
MC ECU Create.vi	Creates an ECU object in memory.
MC ECU Deselect.vi	Deselects an ECU and invalidates the ECU reference handle.
MC ECU Disconnect.vi	Permanently disconnects the CCP communication to the selected ECU and ends the calibration session.
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 the Database Open and ECU Select VIs.
MC ECU Select.vi	Selects an ECU from the names stored in an A2L database.
MC ECU Set Calibration Page.vi	Sets the appropriate RAM or ROM calibration page on the ECU.
MC Event Create.vi	Creates an Event object in memory.
MC Generic.vi	Sends a generic CCP or XCP command.

Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

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

Table 5-1. ECU M&C API VIs for LabVIEW (Continued)

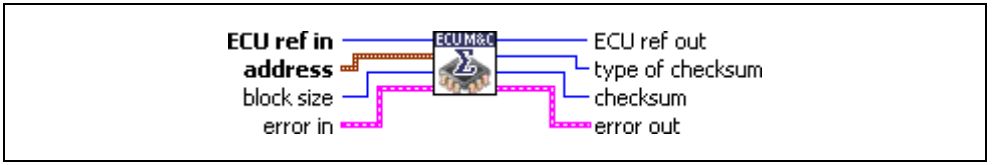
Function	Purpose
MC XCP Set Request.vi	Performs a request to save session and device information to non-volatile memory.
MC XCP Set Segment Mode.vi	Sets the mode of a specified segment.

MC Build Checksum.vi

Purpose

Calculates a checksum over a defined memory range within the 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 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:

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

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.

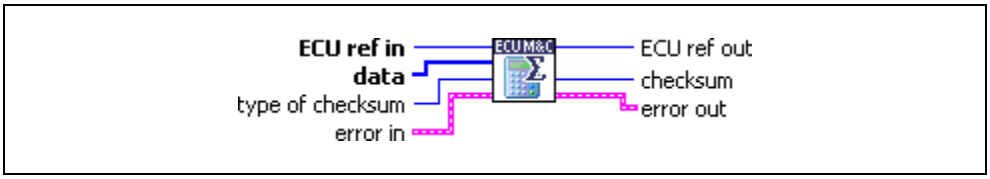
For a detailed description of the checksum algorithm, refer to the *XCP Part 2 Protocol Layer Specification*.

MC Calc Checksum.vi

Purpose

Calculates the checksum of a data block in memory.

Format



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.

Data is a byte array upon which the checksum calculation is performed.

Type of checksum specifies the kind of checksum which is 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.

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:

Type	Name	Description
0x01	XCP_ADD_11	Add BYTE into a BYTE checksum, ignore overflows
0x02	XCP_ADD_12	Add BYTE into a WORD checksum, ignore overflows
0x03	XCP_ADD_14	Add BYTE into a DWORD checksum, ignore overflows
0x04	XCP_ADD_22	Add WORD into a WORD checksum, ignore overflows, blocksize must be modulo 2
0x05	XCP_ADD_24	Add WORD into a DWORD checksum, ignore overflows, blocksize must be modulo 2

Type	Name	Description
0x06	XCP_ADD_44	Add DWORD into DWORD, ignore overflows, blocksize must be modulo 4
0x07	XCP_CRC_16	See CRC error detection algorithms
0x08	XCP_CRC_16_CITT	See CRC error detection algorithms
0x09	XCP_CRC_32	See CRC error detection algorithms
0xFF	XCP_USER_DEFINED	User defined algorithm, in externally calculated function

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:

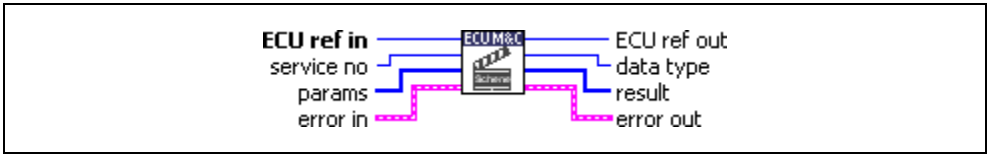
`http://www.repairfaq.org/filipg/LINK/F_crc_v34.html`

MC CCP Action Service.vi

Purpose

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

Format



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

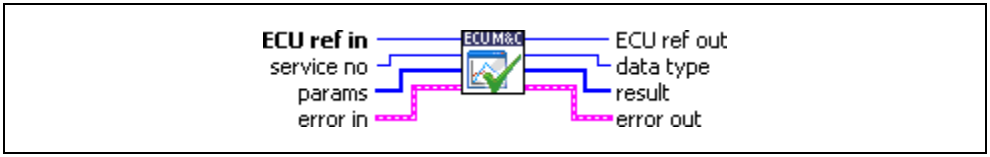
MC CCP Action Service.vi implements the CCP command ACTION_SERVICE. The ECU carries out the requested service and automatically uploads the requested action service return information.

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

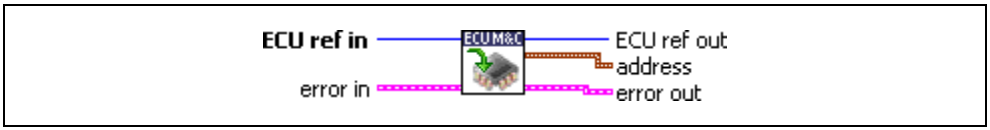
MC CCP Diag Service.vi implements the CCP command DIAG_SERVICE, which starts a diagnostic service on the ECU and waits until it is finished. The selected **Service no** specifies the diagnostic service that is executed inside the ECU. For more information about the available services that are implemented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

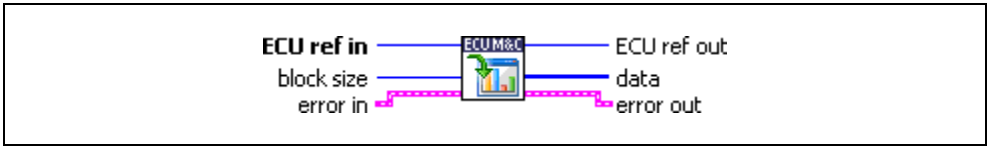
MC CCP Get Active Cal Page.vi implements the CCP command GET_ACTIVE_CAL_PAGE defined by the CCP specification.

MC CCP Get Result.vi

Purpose

Uploads requested data (CCP only).

Format



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

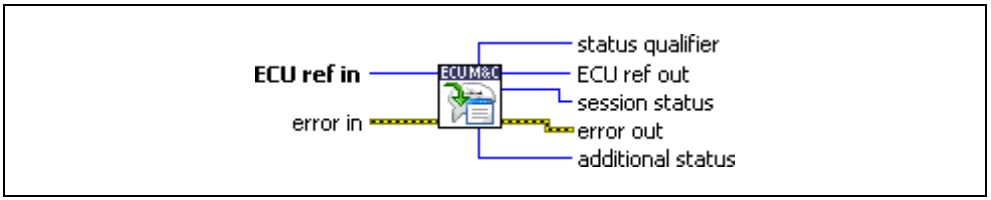
MC CCP Get Result.vi uploads data bytes from the ECU. It is assumed that the Memory Transfer Address 0 (MTA0) has been set by a previous VI like **MC Generic.vi** with the command SET_MTA.

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`.

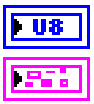
code is the error code number identifying an error. A value of `0` means success. A negative value means error: VI did not execute the intended operation. A positive value means warning: VI executed intended operation, but an informational warning is returned. For a description of the **code**, wire 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.

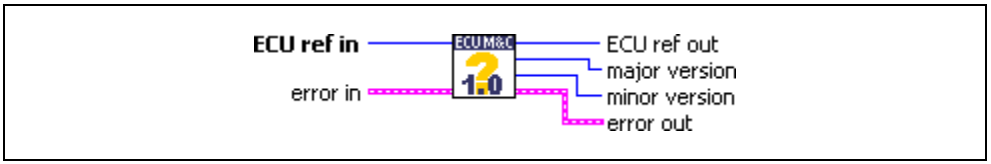
MC CCP Get Session Status.vi implements the CCP command GET_S_STATUS defined by the CCP specification.

MC CCP Get Version.vi

Purpose

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

Format



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.

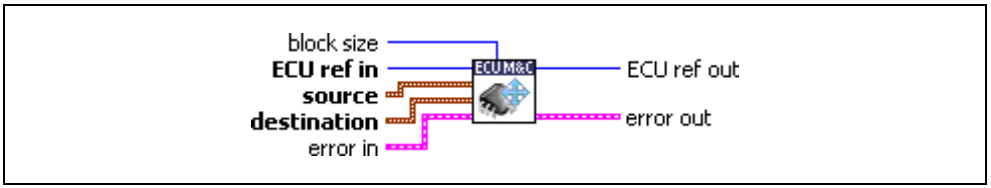
MC CCP Get Version.vi implements the CCP command GET_CCP_VERSION defined by the CCP specification.

MC CCP Move Memory.vi

Purpose

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

Format



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.

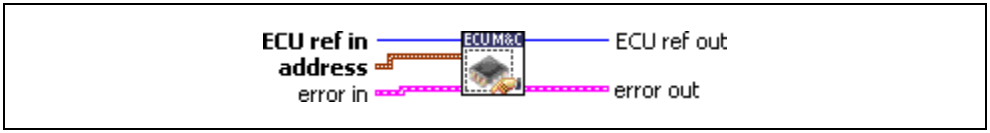
MC CCP Move Memory.vi implements the CCP command MOVE defined by the CCP specification.

MC CCP Select Cal Page.vi

Purpose

Sets the beginning of 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.

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

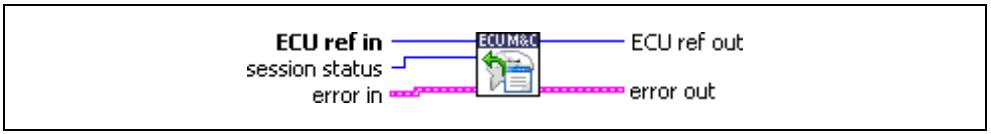
MC CCP Select Cal Page.vi implements the CCP command SELECT_CAL_PAGE. The operation of the command depends on the ECU implementation.

MC CCP Set Session Status.vi

Purpose

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

Format



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.

Session status is the new status to be set in 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

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 5-2. Bit Mask Assignment for Calibration Session Status

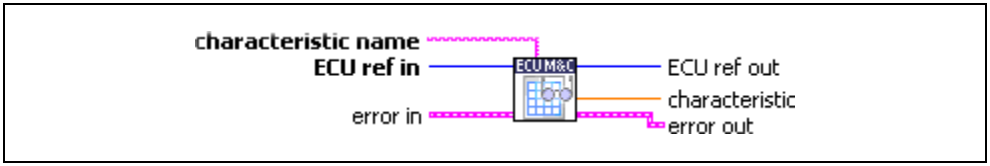
Bit	Name	Description
0	CAL	Calibration data initialized.
1	DAQ	DAQ list(s) initialized.
2	RESUME	Request to save DAQ set-up during shutdown in CCP slave. CCP slave automatically restarts DAQ after start-up.
3	Reserved	—
4	Reserved	—
5	Reserved	—
6	STORE	Request to save calibration data during shut-down in CCP slave.
7	RUN	Session in progress.

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



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

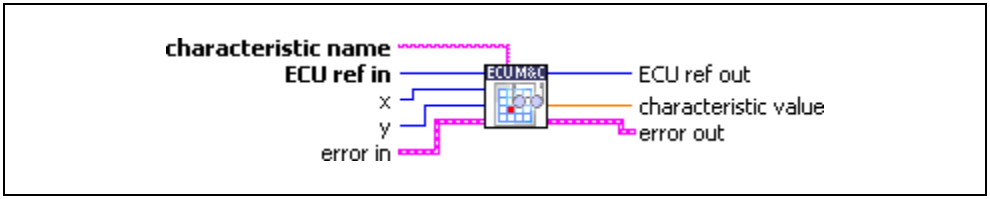
VI Type	Description
Parameter (DBL)	Returns a single double value for the selected Characteristic name.
Curve (1D)	Returns a 1-dimensional array of double values for the selected Characteristic name.
Field (2D)	Returns a 2-dimensional array of double values for the selected Characteristic name.

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



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

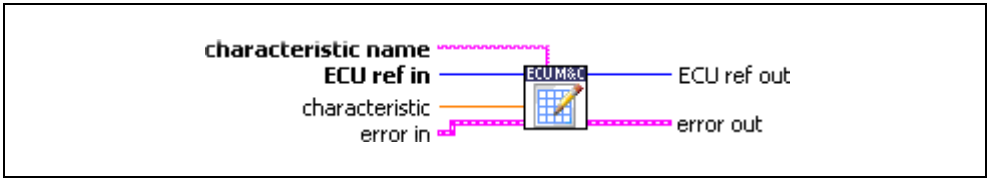
MC Characteristic Read Single Value.vi reads a value from a specified Characteristic on the ECU which is identified by the ECU Reference handle. The value to be read is identified by the **x** and **y** indices. If the Characteristic array has 0 or 1 dimensions, **y** and/or **x** can be left unwired.

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

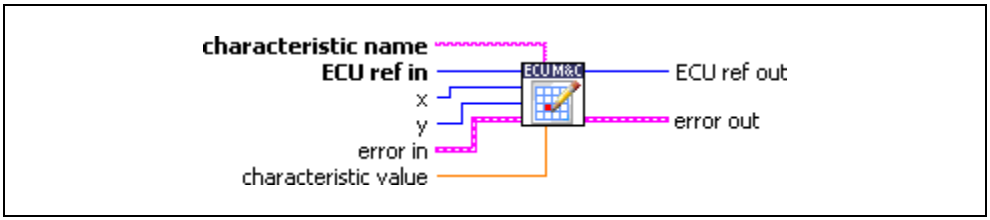
VI Type	Description
Parameter (DBL)	Writes a single double value to the selected Characteristic name.
Curve (1D)	Writes a 1-dimensional array of double values to the selected Characteristic name.
Field (2D)	Writes a 2-dimensional array of double values to the selected Characteristic name.

MC Characteristic Write Single Value.vi

Purpose

Writes a value to a named Characteristic on the ECU.

Format



Input

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

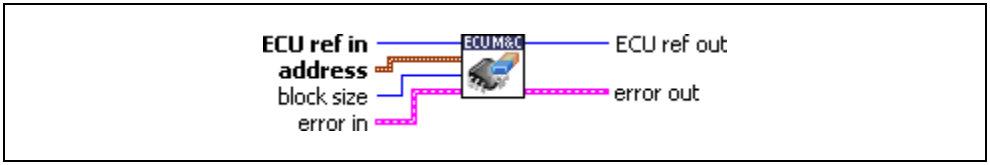
MC Characteristic Write Single Value.vi writes a value to a defined Characteristic on the ECU which is identified by the ECU Reference handle. The location to which the value is written is identified by the **x** and **y** indices. If the Characteristic array has 0 or 1 dimensions, **y** and/or **x**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

MC Clear Memory.vi

Purpose

Clears the contents of a specified memory block.

Format



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 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.

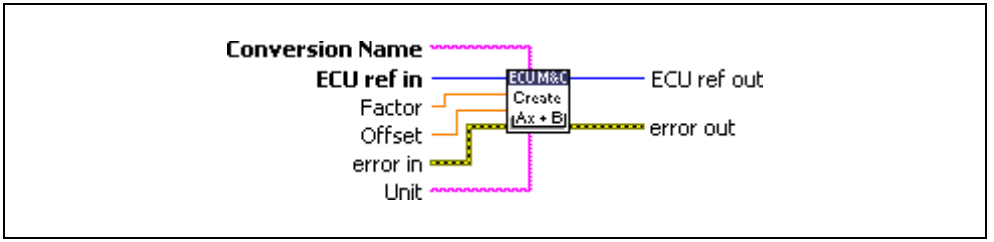
For further details on how to clear parts of non-volatile memory in the ECU refer to the ASAM *XCP Protocol Layer Specification*.

MC Conversion Create.vi

Purpose

Creates a signal conversion object in memory.

Format



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

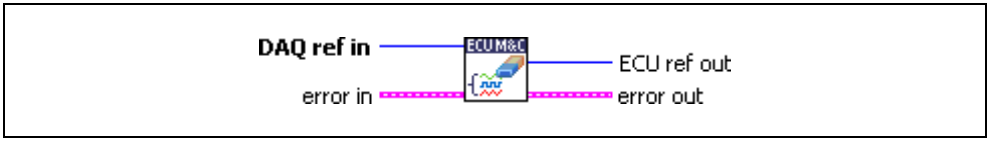
Use **MC Conversion Create.vi** to create a conversion object in memory instead of referring to measurement properties defined in the A2L database.

MC DAQ Clear.vi

Purpose

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

Format



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.

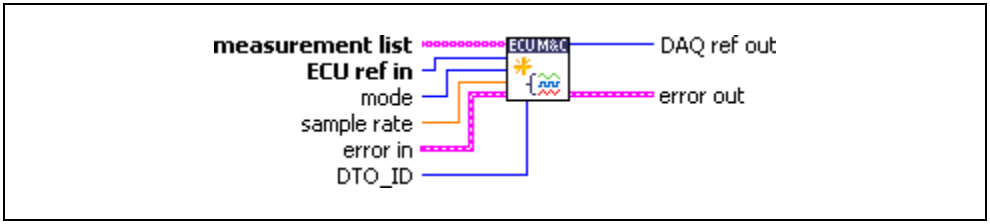
Because this VI clears the Measurement task, the Measurement task reference is not wired as an output but is transferred into an ECU reference task handle. To change properties of a running Measurement task, use **MC DAQ Start Stop.vi** to stop the task, **MC Set Property.vi** to change the desired DAQ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



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 **D**ata **T**ransmission **O**bject (**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

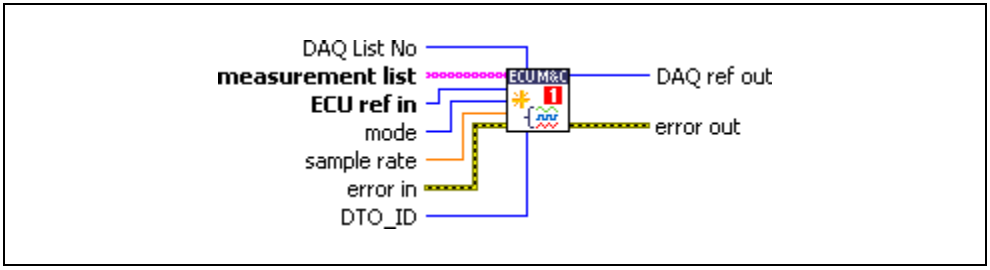
MC DAQ Initialize.vi does not start the transmission of the DAQ lists from the ECU to or from the application through CCP or XCP. This enables you to use **MC Set Property.vi** to change the properties of a Measurement task. After you change properties use **MC DAQ Start Stop.vi** to start the communication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



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.

DTO_ID is the CAN identifier for the **D**ata **T**ransmission **O**bject (**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

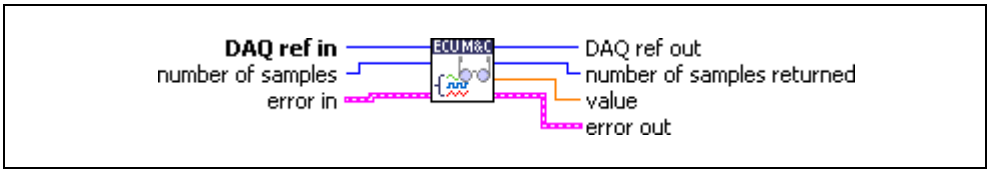
If an ECU offers a reduced and specific range of DAQ list entry numbers use **MC DAQ List Initialize.vi** to setup your Measurement list. **MC DAQ List Initialize.vi** does not start the transmission of the DAQ lists from the ECU to the application or vice versa through CCP or XCP. This enables you to 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



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.

Number of samples 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.

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.

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 to simply 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 **MC DAQ**

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 Read NChan NSamp Time-Value XY Array

Returns an array of clusters with one entry for each channel initialized in the measurement list. Each entry consists of a cluster of a timestamp array and a value (double) array. The timestamp and value arrays have *N* data points each, one for each sample returned. The timestamp sample indicates when the respective measurement sample was received. The order of channel entries in samples is the same as the order in the original channel list. You can wire the output of this type directly to a LabVIEW XY graph display. To use this type, you must set the initialized mode to timestamped read. The VI waits for **Number of samples** messages. 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. To avoid blocking, use mcPropDAQ_SamplesPending to check the number of available data points.

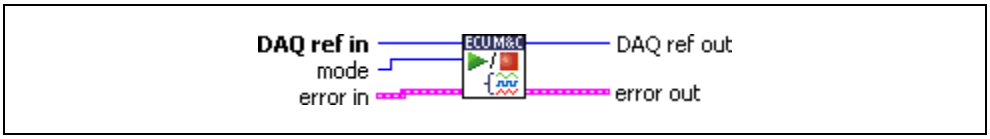
Because the timing of values in samples is determined by when the message is received, the **sample rate** input is not used with this poly VI type. To determine the number of channels in the task after initialization, get the **Number of Channels** property for the task reference.

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.

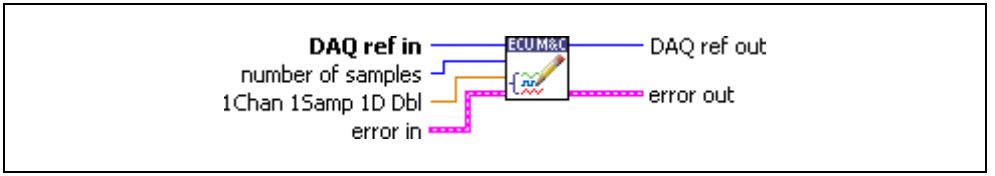
After you start the transmission of the DAQ lists, you can no longer change the configuration of the task with **MC Set Property.vi**. You must call **MC DAQ Start Stop.vi** (Stop DAQ list) first.

MC DAQ Write.vi

Purpose

Writes samples to an ECU DAQ list.

Format



Input

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.

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 is 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.

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

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

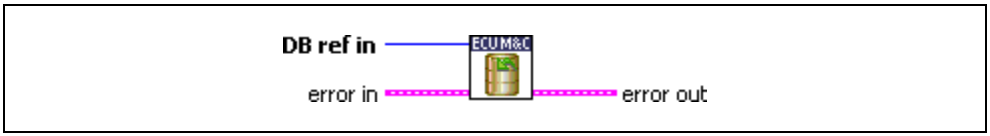
Writes an array, one entry for each channel initialized in the channel list. Each entry consists of a single sample. The order of channel entries in samples is the same as the order in the original channel list. If the initialized sample rate is greater than zero, this poly VI type waits for the next 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



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.

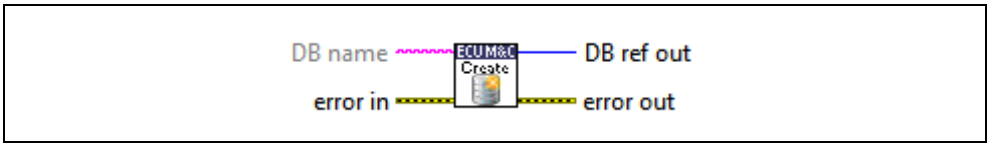
Unlike other VIs, **MC Database Close.vi** will execute when status is `TRUE` in **Error in**. Because this VI clears the task, the task reference is not wired as an output.

MC Database Create.vi

Purpose

Creates an ASAM A2L database in memory.

Format



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**.

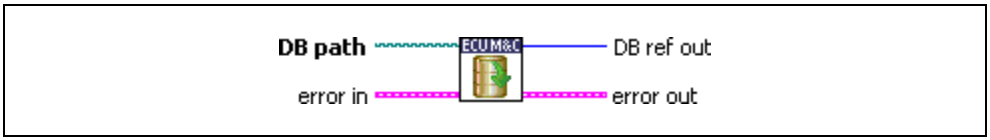
Note **MC Database Create.v**i does not support creating objects to access characteristics. To access a characteristic, assign a valid A2L database file with defined characteristics.

MC Database Open.vi

Purpose

Opens a specified A2L Database.

Format



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.

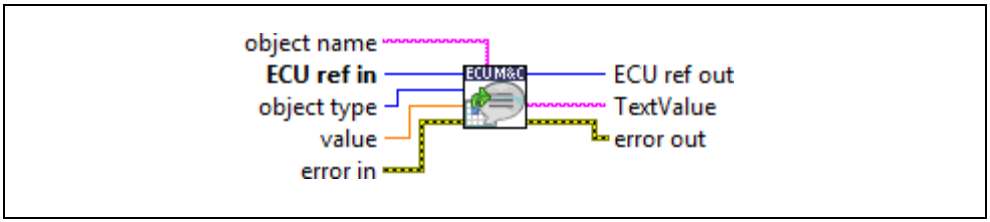
MC Database Open.vi is an advanced function for database handling. In most cases it is sufficient to use **MC ECU Open.vi** instead.

MC Double to Text.vi

Purpose

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

Format



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.

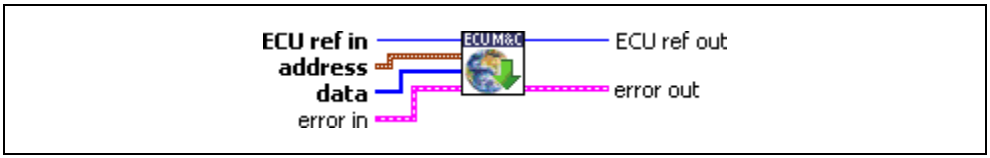
Especially if the measurement or characteristic has an associated enumeration or range text type scaling, the textual representation of the value is returned. If no such value is present, either because the object does not have a text scaling or the value does not have a textual representation in the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

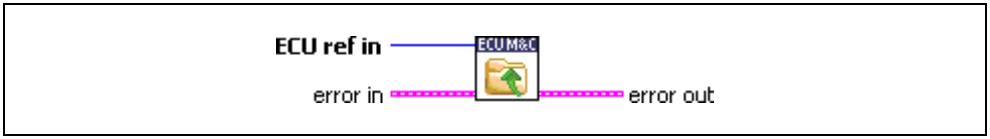
On XCP protocol, when the slave supports the block mode, ECU sends the data in blocks using the DOWNLOAD_NEXT command.

MC ECU Close.vi

Purpose

Closes the selected ECU and the associated A2L database.

Format



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. 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

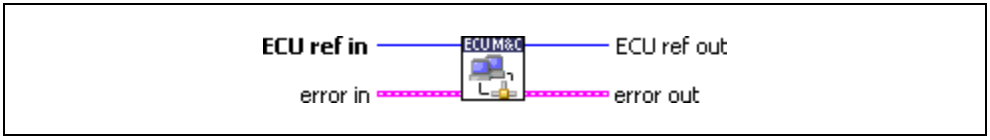
MC ECU Close.vi is the very last VI which must be called. It deselects the ECU and closes the remaining database reference handle. **MC ECU Close.vi** must always be the final M&C VI. If you do not use **MC ECU Close.vi**, the remaining task configurations can cause problems in the execution of subsequent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

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



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

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.

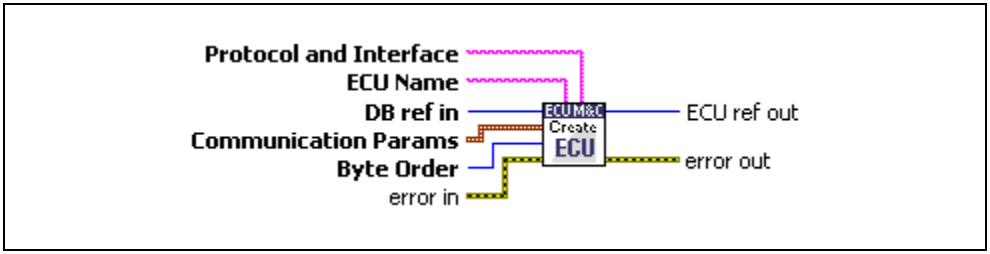
MC ECU Connect.vi is an optional function and is automatically performed before **MC Characteristic Read.vi**, **MC Characteristic Write.vi**, **MC DAQ Initialize.vi**, any MC CCP *xxx* command, or any MC XCP *xxx* command is performed.

MC ECU Create.vi

Purpose

Creates an ECU object in memory.

Format



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 8000*XXYY* 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

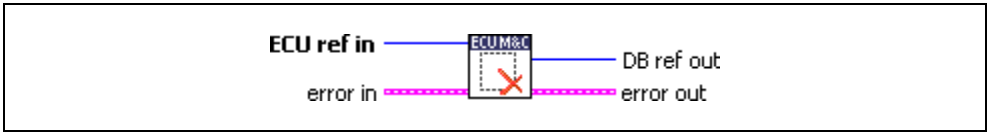
Use **MC ECU Create.vi** to create an ECU object in memory instead of referring to an ECU object defined in the A2L database. **MC ECU Create.vi** provides an alternative in which you create the ECU and DAQ List configuration within the application, without using an A2L database. **MC ECU Create.vi** 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



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. 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

DB ref 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

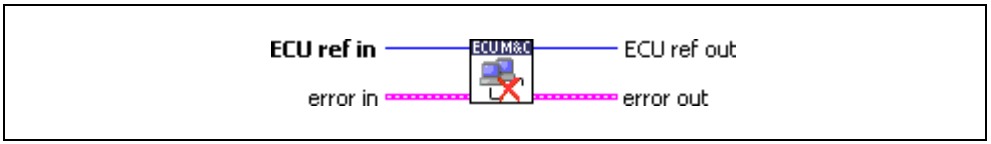
MC ECU Deselect.vi deselects the communication to the ECU. After calling this VI you can establish the communication to another ECU defined in the A2L database using **MC ECU Select.vi**.

MC ECU Disconnect.vi

Purpose

Disconnects the CCP or XCP communication to the selected ECU.

Format



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.

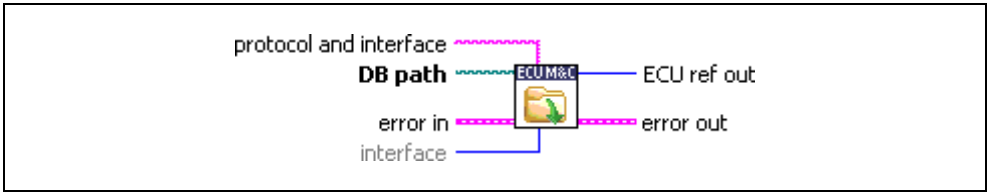
MC ECU Disconnect.vi is an optional function and is automatically performed prior to any **MC ECU Deselect.vi** or **MC ECU Close.vi** call.

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



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**.

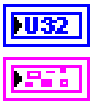
source identifies the VI where the error occurred.

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 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 CAN*x*
- **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 CAN*x*

- **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 CAN*x*
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 1
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 2, and so on with the form CAN*x*
- **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 CAN*x*
- **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 CAN*x*

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`

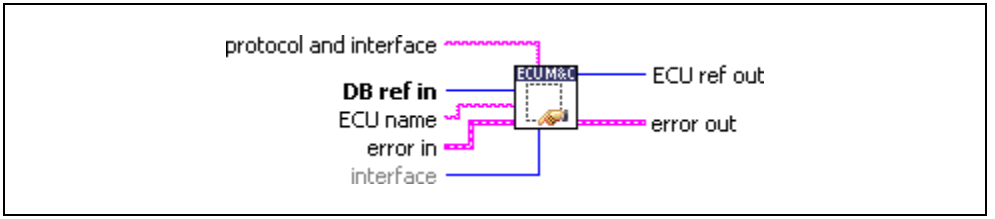
Note You can download the ASAM MCD 2MC database configuration file to a LabVIEW RT target by the File Transfer Protocol (FTP). An FTP file transfer is possible within MAX. Refer to the *LabVIEW Real-Time Graphical File Transfer Utility* section of Chapter 2, *Installation and Configuration*, for instructions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



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 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 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 CAN*x*
- **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 CAN*x*
- **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 CAN*x*
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 1
- **XCP:CAN1@nixnet**—uses XCP on CAN interface 2, and so on with the form CAN*x*
- **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 CAN*x*
- **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 CAN*x*

Using CompactRIO or R Series:

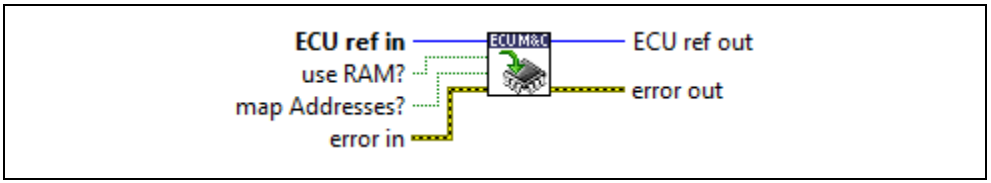
- **CCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx**—uses CCP on named target RIO1 as compiled into the bitfile at `c:\temp\MyFpgaBitfile.lvbitx`
- **XCP:CAN1@RIO1,c:\temp\MyFpgaBitfile.lvbitx**—uses XCP on named target RIO1 as compiled into the bitfile at `c:\temp\MyFpgaBitfile.lvbitx`

MC ECU Set Calibration Page.vi

Purpose

Sets the appropriate RAM or ROM calibration page on the ECU.

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**, and then wired through subsequent VIs.

use RAM? indicates which page should be set. Set this input to TRUE for the RAM page or FALSE for the ROM page.

map Addresses? activates address mapping from the ROM page to the target page specified in **use RAM?**.

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 Set Calibration Page.vi tries to identify a single RAM or ROM page on the ECU and select it according to the **use RAM?** input.

To identify an appropriate page, the VI searches the calibration page information from the A2L file or the online information from the ECU. If the VI can identify a unique calibration page, it is activated in the ECU, and the VI returns success.

If the VI cannot identify a unique page, an error is returned indicating this, and no further action is taken. This does not, however, state a fault, but just the algorithm's inability to uniquely identify the desired page. In this case, you can use the calibration page-related ECU properties (**MC Get Property.vi**, **ECU»CCP»Cal Pages»…** or **ECU»XCP»Cal Pages»…**) to gain the information about available calibration pages, and manually select the correct page using **MC CCP Select Cal Page.vi** or **MC XCP Set Cal Page.vi**.

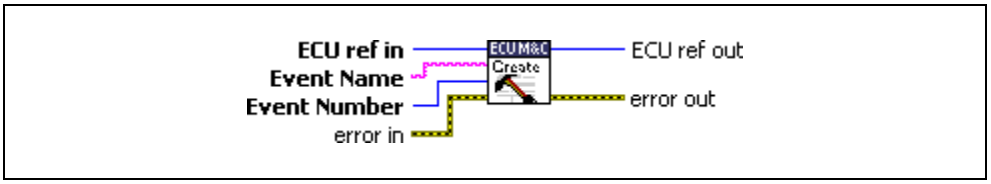
The **map Addresses?** input activates the address mapping from the ROM page, assumed to be the reference page to the target page specified in **use RAM?**. Address mapping is supported only for the CCP protocol and requires a unique ROM and unique RAM page in the A2L file. Addresses of measurements and characteristics in the A2L file must point to the ROM page as a reference page.

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.

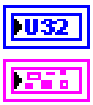
Error in is a cluster which describes error conditions occurring before the VI executes. If an error has already occurred, the VI returns the value of the **error in** cluster to **error out**.

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**.

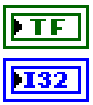
source identifies the VI where the error occurred.

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

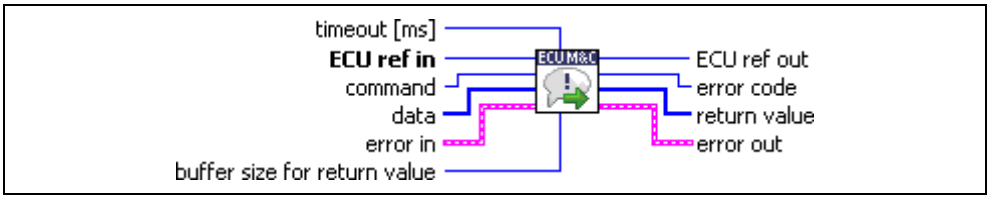
Use **MC Event Create.vi** to create an Event object in memory instead of referring to a predefined measurement in the A2L database. Assign the event channel object by name to a DAQ List in **MC Measurement Create.vi**.

MC Generic.vi

Purpose

Sends a generic CCP or XCP command.

Format



Input

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

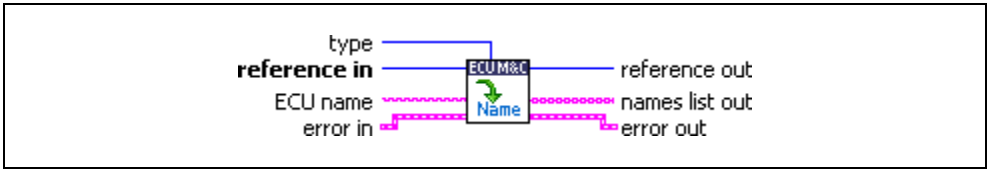
MC Generic.vi implements any generic CCP or XCP command that can be used to execute user-defined commands that are not defined in the CCP or XCP standard or not covered by the available LabVIEW CCP/XCP VIs.

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

Type (mode) is an input that specifies the type of names to return.

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 Name**s 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.

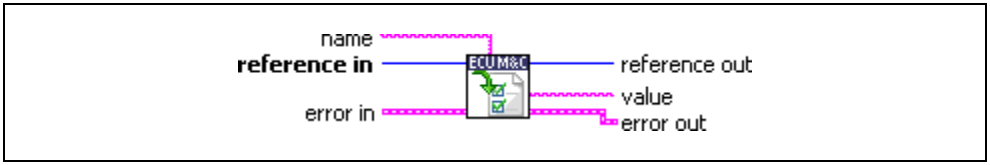
Note For more details on how to query the Group information out of an A2L file, refer to the installed advanced example (**Read A2L Group.vi**) in the Example Finder.

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 5-5. Poly Values for Value Output

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	—	—	—	**DB File Name**	Returns the A2L Database file name with which the task has been opened. The value of this property cannot be changed using **MC Set Property.vi**.
	ECU	—	—	**Byte Order**	Returns 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	—	—	**Seedkey/ Checksum DLL Path**	This property determines the directory where the ECU M&C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&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.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	—	—	**Checksum DLL Name**	Returns the file name of the Checksum DLL used for verifying the checksum.
	ECU	—	—	**LogFile Path**	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.
	ECU	—	—	**Command Byte Order**	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.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	—	—	**Name**	Returns the **Name** of the selected ECU opened by **MC ECU Open.vi** or **MC ECU Select.vi**.
	ECU	—	—	**Comment**	Returns the **Comment** string of the selected ECU.
	ECU	—	—	**DAQ List Number**	Returns an array of DAQ list numbers for all DAQ lists defined in the A2L file.
	ECU	—	—	**Event Channel**	Translates the event channel name to the event channel number. Pass the event channel name in the **Name** parameter of **Get Property**.
	ECU	CCP	—	**Baud Rate**	Returns the **Baud Rate** in use by the Interface. Basic baud rates such as 125000 and 500000 are specified as the numeric rate. Advanced baud rates are specified as 8000*XXYY* 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 **Interface Properties** dialog in MAX. The value of this property is originally set within MAX, but it can be changed using **MC Set Property.vi**.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U32	ECU	CCP	—	**CRO ID**	Returns the **CRO ID** (**C**ommand **R**eceive **O**bject) which is used to send commands and data from the host to the slave device.
U32	ECU	CCP	—	**CRO Task**	Returns the NI-CAN task reference for the **CRO** (**C**ommand **R**eceive **O**bject, 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&C Toolkit itself.
U32	ECU	CCP	—	**DTO ID**	Returns the **DTO ID** (**D**ata **T**ransmission **O**bject) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.
U32	ECU	CCP	—	**DTO Task**	Returns the NI-CAN task reference for the **DTO ID** (**D**ata **T**ransmission **O**bject, 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&C Toolkit itself.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**ID**	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.
	ECU	CCP	—	**ID Data Byte**	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.
	ECU	CCP	—	**Interface**	Returns the interface initialized for the task, such as with **MC DAQ Initialize.vi**.
	ECU	CCP	—	**Master ID**	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.
	ECU	CCP	—	**SeedKey Cal Name**	Returns the filename of the SeedKey DLL used for Calibration purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**SeedKey DAQ Name**	Returns the filename of the SeedKey DLL used for DAQ purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
	ECU	CCP	—	**SeedKey Prog Name**	Returns the filename of the SeedKey DLL used for programming purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
	ECU	CCP	—	**Single Byte DAQ List?**	Determines if an ECU supports single-byte or multi-byte DAQ list entries.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**Termination**	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)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	CAN	**Termination**	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 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U32	ECU	CCP	—	**Station Address**	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.
TF	ECU	CCP	Misc	**Skip EXCHANGE ID**	Returns a Boolean value that indicates whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.
TF	ECU	CCP	Optional Commands	**ACTION SERVICE**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command ACTION_SERVICE.
TF	ECU	CCP	Optional Commands	**BUILD CHECKSUM**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command BUILD_CHKSUM.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
TF	ECU	CCP	Optional Commands	**CLEAR MEMORY**	Returns a Boolean value that indicates whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.
TF	ECU	CCP	Optional Commands	**CLEAR MEMORY**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command CLEAR_MEMORY.
TF	ECU	CCP	Optional Commands	**DIAG SERVICE**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command DIAG_SERVICE.
TF	ECU	CCP	Optional Commands	**DNLOAD 6**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command DNLOAD_6.
TF	ECU	CCP	Optional Commands	**GET ACTIVE CAL PAGE**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_ PAGE.
TF	ECU	CCP	Optional Commands	**GET S STATUS**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_S_STATUS.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
TF	ECU	CCP	Optional Commands	**GET SEED**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command GET_SEED.
TF	ECU	CCP	Optional Commands	**MOVE**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command MOVE.
TF	ECU	CCP	Optional Commands	**PROGRAM**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command PROGRAM.
TF	ECU	CCP	Optional Commands	**PROGRAM 6**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command PROGRAM_6.
TF	ECU	CCP	Optional Commands	**SELECT CAL PAGE**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.
TF	ECU	CCP	Optional Commands	**SET S STATUS**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command SET_S_STATUS.
TF	ECU	CCP	Optional Commands	**SHORT UP**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command SHORT_UP.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
TF	ECU	CCP	Optional Commands	**START STOP ALL**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command START_STOP_ALL.
TF	ECU	CCP	Optional Commands	**TEST**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command TEST.
TF	ECU	CCP	Optional Commands	**UNLOCK**	Returns a Boolean value that indicates whether the ECU supports the optional CCP Command UNLOCK.
U32	ECU	Misc	—	**Timing Factor**	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.
U8	ECU	XCP	—	**Compression Method**	Returns the selected compression method used for **MC Program.vi**. 0—data is uncompressed. 0x80...0xFF—User defined.
U8	ECU	XCP	—	**Encryption Method**	Returns the selected encryption method used for **MC Program.vi**. 0x00—data is not encrypted 0x80...0xFF—User defined

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	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 filename of the SeedKey DLL. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	CAN	**Baudrate**	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 8000*XXYY* 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 **Interface Properties** dialog in MAX. The value of this property is originally set within MAX, but it can be changed using **MC Set Property.vi**.
	ECU	XCP	CAN	**CRO Id**	Returns the CRO ID (Command Receive Object) which is used to send commands and data from the host to the slave device.
	ECU	XCP	CAN	**DTO Id**	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.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	Ethernet	**IP Address**	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.
	ECU	XCP	Ethernet	**IP Port**	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.
	ECU	XCP	Timeout	**T1** **T2** **T3** **T4** **T5** **T6** **T7**	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 **Timing Factor** property might modify this value.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Timeout	**T_std**	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 **Timing Factor** property might modify this value.
	ECU	CCP	Timeout	**T_pgm**	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 **Timing Factor** property might modify this value.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U32	ECU	CCP	Timeout	**T_mem**	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 **Timing Factor** property might modify this value.
U32	ECU	CCP	Timeout	**T_diag**	Returns the timeout value (in milliseconds) for the CCP DIAG_SERVICE command. Default and Standard: 500. Note that the **Timing Factor** property might modify this value.
U32	ECU	CCP	Timeout	**T_act**	Returns the timeout value (in milliseconds) for the CCP ACTION_SERVICE command. Default: 500. Standard: 5000. Note that the **Timing Factor** property might modify this value.
U32	ECU	CCP	Cal Pages	**Number of Pages**	Returns the number of DEFINED_PAGES structures for this ECU in the A2L file.
U32	ECU	CCP	Cal Pages	**Page Number**	Returns the page number of the page selected with the **name** input.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Cal Pages	**Page Flags**	Returns the page flags of the page selected with the **name** input. The value returned is a bitmask ored from the following values: 1 RAM page 2 ROM page 4 FLASH page 8 EEPROM page 16 RAM_INIT_BY_ ECU RAM page initialized at ECU startup. 32 RAM_INIT_BY_ TOOL RAM page that the calibration tool initializes. 64 AUTO_FLASH_ BACK RAM page automatically flashed back. 128 FLASH_BACK RAM page that the calibration tool can flash back. 256 DEFAULT Page is standard (fallback).

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Cal Pages	**Page Address**	Returns the memory address (and extension) of the page selected with the **name** input.
	ECU	XCP	Cal Pages	**Number of Segments**	Returns the number of XCP memory segments found for this ECU.
	ECU	XCP	Cal Pages	**Number of Pages**	Returns the number of memory pages defined for the memory segment specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property).
	ECU	XCP	Cal Pages	**Address Extension**	Returns the address extension for the memory segment specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property).
	ECU	XCP	Cal Pages	**Compression Method**	Returns the compression method for the memory segment specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property). A value of 0 means no compression. Other values are user defined.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	Cal Pages	**Encryption Method**	Returns the encryption method for the memory segment specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property). A value of 0 means no encryption. Other values are user defined.
	ECU	XCP	Cal Pages	**Page Number**	Returns the logical page number for the memory segment page specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property) and **page** input (0..*M*-1, where *M* is the value returned from the **Number of Pages** property).

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	Cal Pages	**ECU Access**	Returns a flag indicating ECU access rights for the memory segment page specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property) and **page** input (0..*M*-1, where *M* is the value returned from the **Number of Pages** property). Defined values are: 0 ECU access not allowed 1 ECU access allowed without XCP access only 2 ECU access allowed with XCP access only 3 ECU access allowed always

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	Cal Pages	**XCP Read Access**	Returns a flag indicating XCP Read access rights for the memory segment page specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property) and **page** input (0..*M*-1, where *M* is the value returned from the **Number of Pages** property). Defined values are: 0 XCP Read access not allowed 1 XCP Read access allowed without ECU access only 2 XCP Read access allowed with ECU access only 3 XCP Read access allowed always

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U8	ECU	XCP	Cal Pages	**XCP Write Access**	Returns a flag indicating XCP Write access rights for the memory segment page specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property) and **page** input (0..*M*-1, where *M* is the value returned from the **Number of Pages** property). Defined values are: 0 XCP Write access not allowed 1 XCP Write access allowed without ECU access only 2 XCP Write access allowed with ECU access only 3 XCP Write access allowed always
U8	ECU	XCP	Cal Pages	**Page InitSegment**	Returns the number of the segment that initializes the memory segment page specified in the **segment** input (0..*N*-1, where *N* is the value returned from the **Number of Segments** property) and **page** input (0..*M*-1, where *M* is the value returned from the **Number of Pages** property).

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Characteristic	—	—	**Address**	Returns the address of the selected Characteristic in the memory of the ECU.
	Characteristic	—	—	**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.
	Characteristic	—	—	**Comment**	Returns the **Comment** string of the selected Characteristic.
	Characteristic	—	—	**Data Type**	Returns the data type of the Characteristic.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U32	Characteristic	—	—	**Dimension**	Returns the dimension of a Characteristic. 0—0 dimension The Characteristic can be accessed (read/write) through a double value. 1—1 dimension The Characteristic can be accessed (read/write) through a one-dimensional array of double values. 2—2 dimensions The Characteristic can be accessed (read/write) through a two-dimensional array of double values.
U8	Characteristic	—	—	**Extension**	Returns additional address information. For instance it can be used, to distinguish different address spaces of an ECU (multi-microcontroller devices).
DBL	Characteristic	—	—	**Maximum**	Returns the maximum value of the Characteristic.
DBL	Characteristic	—	—	**Minimum**	Returns the minimum value of the Characteristic.
TF	Characteristic	—	—	**Read Only?**	Returns if a Characteristic is set to Read Only. In this case it is not allowed to call **MC Characteristic Write.vi** for this Characteristic.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Characteristic	—	—	**Sizes**	Returns the array **Sizes** for *X* and *Y* direction of the Characteristic.
	Characteristic	—	—	**Unit**	Returns the unit string defined for this Characteristic in the A2L database.
	Characteristic	—	—	**X Axis**	Returns X-axis values on which the Characteristic is defined. It is valid if the dimension of the selected Characteristic is 1 or 2.
	Characteristic	—	—	**Y Axis**	Returns Y-axis values on which the Characteristic is defined. It is valid if the dimension of the selected Characteristic is 2.
	Characteristic	Scaling	—	**Factor**	Returns the scaling factor defined for this Characteristic in the A2L database.
	Characteristic	Scaling	—	**Offset**	Returns the scaling offset defined for this Characteristic in the A2L database.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Characteristic	Scaling	—	**Type**	Returns the scaling type defined for this Characteristic in the A2L database. **0: Unknown** The scaling type 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. **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.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		• 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_RANGE 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 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		**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.
	Characteristic	Scaling	—	**Text Values**	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.
	Group	—	—	**Is Root?**	Returns whether the selected Group is a root-level Group entity.
	Group	—	—	**Comment**	Returns the **Comment** string of the selected Group.
	Function	—	—	**Comment**	Returns the **Comment** string of the selected Function.
	DAQ	—	—	**Event Channel Name**	Returns the selected event channel name to which the Measurement task is assigned.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	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.
	DAQ	—	—	**# Channels**	Returns the number of channels initialized in a DAQ channel list of a M&C Measurement task. This is the number of array entries required when using **MC DAQ Read.vi**.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	DAQ	—	—	**Prescaler**	Returns the prescaling factor which is used to reduce the desired transmission frequency of the associated DAQ list.
	DAQ	—	—	**Sample Rate**	Returns the selected **Sample Rate** in Hz for the M&C Measurement task, which may be obtained with **MC DAQ Initialize.vi**.
	DAQ	—	—	**Samples Pending**	Returns the number of samples available to be read using **MC DAQ Read.vi**. If you set the number of samples to read input of **MC DAQ Read.vi** to this value, **DAQ Read** returns immediately without waiting. This property applies only to tasks initialized with mode of **Input** and sample rate greater than zero. For all other configurations, it returns an error. If this property is queried before the DAQ list is started, it always returns zero. Start the DAQ list first with **MC DAQ Start Stop.vi** before you query this property.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	DAQ	—	—	**Time Since Last Frame**	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.
	DAQ	CCP	—	**DTO ID**	Returns the **DTO ID** (**D**ata **T**ransmission **O**bject) which is used by the ECU to send DAQ list data to the CCP master.
	DAQ	CCP	—	**DTO Task**	Returns the NI-CAN task reference for the **DTO ID** (**D**ata **T**ransmission **O**bject, 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&C Toolkit itself.
	DAQ List	—	—	**CAN ID**	Returns the CAN ID for the specified DAQ list if mcPropDAQList_ CANIdSelectMode == CAN_ID_FIXED.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U32	DAQ List	—	—	**CAN ID Select Mode**	Returns the condition for selecting the CAN ID for the specified DAQ list. 0—CAN_ID_FIXED The CAN Identifier is a predefined fixed number. 1—CAN_ID_ VARIABLE The CAN Identifier is a variable number. 2—CAN_ID_DTO_ID The CAN Identifier is the same as the DTO identifier.
[U16]	DAQ List	—	—	**Excluded DAQ Lists**	Returns an array containing the numbers of DAQ lists not working together with the current DAQ list.
U8	DAQ List	—	—	**First PID**	Returns the first Packet ID for the specified DAQ list.
U16	DAQ List	—	—	**MAX Length**	Returns the maximal length of the DAQ list.
U32	DAQ List	—	—	**Reduction Allowed**	Returns whether or not the specified DAQ list allows reduction.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
abc	DAQ List	—	—	**Name**	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.
abc	DAQ List	—	—	**Display Identifier**	Optional property you can use as a display name as an alternative to the DAQ List **Name** property.
U32	Measurement	—	—	**Address**	Returns the address part of the address of the selected Measurement in the memory of the control unit.
U32	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.
abc	Measurement	—	—	**Comment**	Returns the **Comment** string of the selected Measurement.
U8	Measurement	—	—	**Data Type**	Returns the data type of the Measurement task.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U8	Measurement	—	—	**Extension**	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).
TF	Measurement	—	—	**Is Virtual?**	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.
DBL	Measurement	—	—	**Maximum**	Returns the maximum value of the Measurement.
DBL	Measurement	—	—	**Minimum**	Returns the minimum value of the Measurement.
TF	Measurement	—	—	**Read Only?**	Returns `TRUE` if the selected Measurement is read only and can only be accessed through **MC DAQ Read.vi**, or returns `FALSE` if the Measurement can be accessed through **MC Measurement Write.vi** as well.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Measurement	—	—	**Unit**	Returns the unit string defined for this Measurement in the A2L database.
	Measurement	Scaling	—	**Factor**	Returns the scaling factor defined for this Measurement in the A2L database.
	Measurement	Scaling	—	**Offset**	Returns the scaling offset defined for this Measurement in the A2L database.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Measurement	Scaling	—	**Type**	Returns the scaling type defined for this Measurement in the A2L database. **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.

Table 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		• 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_RANGE 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 5-5. Poly Values for Value Output (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		**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.
	Measurement	Scaling	—	**Text Values**	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.
	Version	—	—	**Build**	Returns the build number of the ECU M&C software. This number applies to the Development, Alpha, and Beta phases only, and should be ignored for the Release phase.
	Version	—	—	**Comment**	Returns a comment string for the ECU M&C software. If you received a custom release of ECU M&C from National Instruments, this comment often describes special features of the release.

Table 5-5. Poly Values for Value Output (Continued)

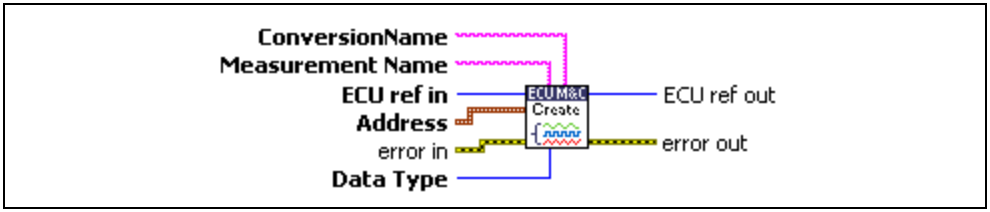
Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Version	—	—	**Major**	Returns the major version of the ECU M&C software, such as the 1 in version 1.2.5.
	Version	—	—	**Minor**	Returns the minor version of the ECU M&C software, such as the 2 in version 1.2.5.
	Version	—	—	**Update**	Returns the update version of the ECU M&C software, such as the 5 in version 1.1.5.

MC Measurement Create.vi

Purpose

Creates a Measurement object in memory.

Format



Input

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:

Data Type	Data Format
0	Unsigned byte
1	Signed byte
2	Unsigned word
3	Signed word
4	Unsigned long
5	Signed long
6	Float 32

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

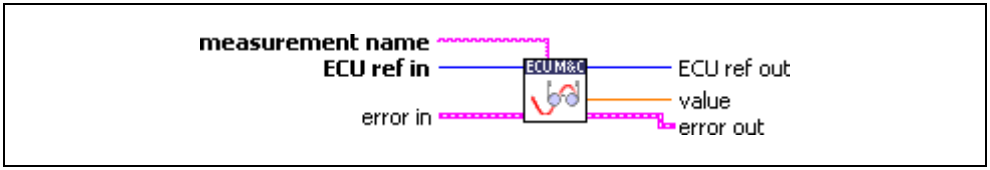
Use **MC Measurement Create.vi** to create a measurement object in memory instead of referring to a predefined measurement in the A2L database.

MC Measurement Read.vi

Purpose

Reads a single Measurement value from the ECU.

Format



Input

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.

Output

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**.

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

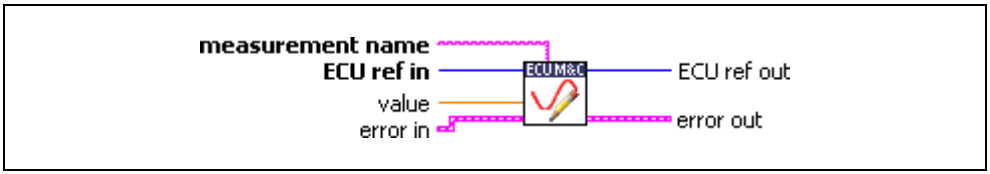
MC Measurement Read.vi performs a single point read of a single Measurement from the selected ECU without opening a Measurement task.

MC Measurement Write.vi

Purpose

Writes a single Measurement value to the ECU.

Format



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.

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

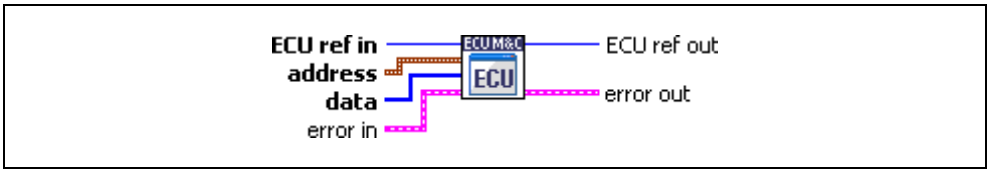
MC Measurement Write.vi performs a single point write of a Measurement into the selected ECU without opening a Measurement task. **MC Measurement Write.vi** can only be performed if the Measurement is not set to **read only**. To query if an ECU Measurement channel can be accessed by **MC Measurement 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



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.

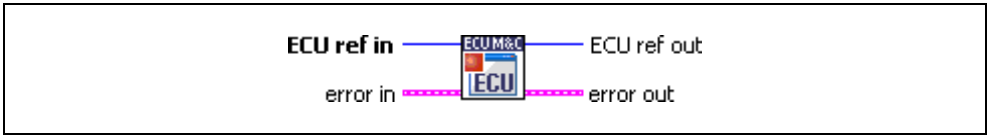
For further information on how to use the **MC Program.vi** and details on block mode transfers, refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

MC Program Reset.vi

Purpose

Indicates the end of a programming sequence.

Format



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.

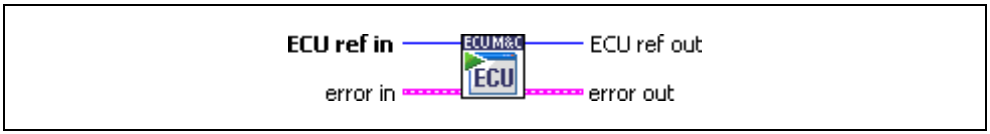
MC Program Reset.vi may be used to reset a slave device for other purposes. For further information on how to use program ECU memory and to use the **MC Program Reset.vi** command refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

MC Program Start.vi

Purpose

Indicates the start of a programming sequence.

Format



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.

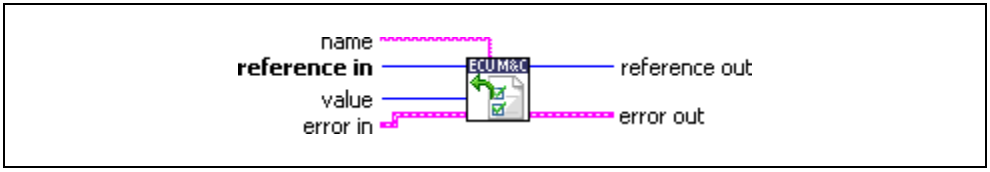
For further information on how to use program ECU memory and to use the **MC Program Start.vi** command refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

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



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

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	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)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	—	—	**Seedkey/ Checksum DLL Path**	Determines the directory where the ECU M&C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&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.
	ECU	—	—	**Checksum DLL Name**	Sets the file name of the Checksum DLL used for verifying the checksum.
	ECU	—	—	**Logfile Path**	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.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**Baud Rate**	Sets the **Baud Rate** 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 8000*XXYY* hex, where *YY* is the value of Bit Timing Register 0 (BTR0), and *XX* is the value of Bit Timing Register 1 (BTR1).
	ECU	CCP	—	**CRO ID**	Sets the **CRO ID** (**C**ommand **R**eceive **O**bject) which is used to send commands and data from the host to the slave device.
	ECU	CCP	—	**DTO ID**	Sets the **DTO ID**, which is the CAN identifier for the **D**ata **T**ransmission **O**bject (DTO). The DTO is used by the CCP slave devices to return data and status information to the application.
	ECU	CCP	—	**Master ID**	Sets the CAN identifier of the CCP master that is used by the CCP command EXCHANGE_ID as a parameter.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**SeedKey Cal Name**	Sets the filename of the SeedKey DLL used for Calibration purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
	ECU	CCP	—	**SeedKey DAQ Name**	Sets the filename of the SeedKey DLL used for DAQ purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
	ECU	CCP	—	**SeedKey Prog Name**	Sets the filename of the SeedKey DLL used for programming purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
	ECU	CCP	—	**Single Byte DAQ List?**	Sets the ECU to support single-byte or multi-byte DAQ list entries.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**Termination**	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)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	CAN	**Termination**	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)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	—	**Station Address**	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.
	ECU	CCP	Misc	**Skip EXCHANGE ID**	Sets whether or not the CCP command EXCHANGE_ID should be suppressed during connection to the ECU.
	ECU	CCP	Optional Commands	**ACTION SERVICE**	Sets whether the ECU supports the optional ASAM CCP Command ACTION_SERVICE.
	ECU	CCP	Optional Commands	**BUILD CHECKSUM**	Sets whether the ECU supports the optional ASAM CCP Command BUILD_CHKSUM.
	ECU	CCP	Optional Commands	**CLEAR MEMORY**	Sets whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Optional Commands	**CLEAR MEMORY**	Sets whether the ECU supports the optional ASAM CCP Command CLEAR_MEMORY.
	ECU	CCP	Optional Commands	**DIAG SERVICE**	Sets whether the ECU supports the optional ASAM CCP Command DIAG_SERVICE.
	ECU	CCP	Optional Commands	**DNLOAD 6**	Sets whether the ECU supports the optional ASAM CCP Command DNLOAD_6.
	ECU	CCP	Optional Commands	**GET ACTIVE CAL PAGE**	Sets whether the ECU supports the optional ASAM CCP Command GET_ACTIVE_CAL_ PAGE.
	ECU	CCP	Optional Commands	**GET S STATUS**	Sets whether the ECU supports the optional ASAM CCP Command GET_S_STATUS.
	ECU	CCP	Optional Commands	**GET SEED**	Sets whether the ECU supports the optional ASAM CCP Command GET_SEED.
	ECU	CCP	Optional Commands	**MOVE**	Sets whether the ECU supports the optional ASAM CCP Command MOVE.
	ECU	CCP	Optional Commands	**PROGRAM**	Sets whether the ECU supports the optional ASAM CCP Command PROGRAM.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Optional Commands	**PROGRAM 6**	Sets whether the ECU supports the optional ASAM CCP Command PROGRAM_6.
	ECU	CCP	Optional Commands	**SELECT CAL PAGE**	Sets whether the ECU supports the optional ASAM CCP Command SELECT_CAL_PAGE.
	ECU	CCP	Optional Commands	**SET S STATUS**	Sets whether the ECU supports the optional ASAM CCP Command SET_S_STATUS.
	ECU	CCP	Optional Commands	**SHORT UP**	Sets whether the ECU supports the optional ASAM CCP Command SHORT_UP.
	ECU	CCP	Optional Commands	**START STOP ALL**	Sets whether the ECU supports the optional ASAM CCP Command START_STOP_ALL.
	ECU	CCP	Optional Commands	**TEST**	Sets whether the ECU supports the optional ASAM CCP Command TEST.
	ECU	CCP	Optional Commands	**UNLOCK**	Sets whether the ECU supports the optional ASAM CCP Command UNLOCK.
	ECU	Misc	—	**Timing Factor**	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.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	—	**SeedKey DLL**	Sets the file name of the XCP SeedKey DLL. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
	ECU	XCP	—	**Access Method**	Sets 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	—	**Compression Method**	Sets the selected compression method used for **MC Program.vi**. 0—data is uncompressed. 0x80...0xFF—User defined.
	ECU	XCP	—	**Encryption Method**	Sets the selected encryption method used for **MC Program.vi**. 0x00—data is not encrypted 0x80...0xFF—User defined

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
U8	ECU	XCP	—	**Programming Method**	Sets the selected programming method used for **MC Program.vi**. 0x00—Sequential programming. 0x80...0xFF—User defined.
U32	ECU	XCP	CAN	**Baudrate**	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 8000*XXYY* 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.
U32	ECU	XCP	CAN	**CRO Id**	Sets the CRO ID (Command Receive Object) which is used to send commands and data from the host to the slave device.
U32	ECU	XCP	CAN	**DTO Id**	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.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	XCP	Ethernet	**IP Address**	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.
	ECU	XCP	Ethernet	**IP Port**	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.
	ECU	XCP	Timeout	**T1** **T2** **T3** **T4** **T5** **T6** **T7**	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 **Timing Factor** property may modify this value.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Timeout	**T_std**	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 **Timing Factor** property may modify this value.
	ECU	CCP	Timeout	**T_pgm**	Sets 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 **Timing Factor** property may modify this value.

Table 5-6. ECU-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	ECU	CCP	Timeout	**T_mem**	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 **Timing Factor** property may modify this value.
	ECU	CCP	Timeout	**T_diag**	Sets the timeout value (in milliseconds) for the CCP DIAG_SERVICE command. Default and Standard: 500. Note that the **Timing Factor** property may modify this value.
	ECU	CCP	Timeout	**T_act**	Sets the timeout value (in milliseconds) for the CCP ACTION_SERVICE command. Default: 500. Standard: 5000. Note that the **Timing Factor** property may modify this value.

Table 5-7. DAQ-Specific Property Value Types for the POLY Input Value

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	DAQ	—	—	**Event Channel Name**	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.
	DAQ	—	—	**Mode**	Sets the selected I/O mode for the M&C Measurement task. 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 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.
	DAQ	—	—	**Prescaler**	Sets the prescaling factor, which reduces the desired transmission frequency of the associated DAQ list.

Table 5-7. DAQ-Specific Property Value Types for the POLY Input Value (Continued)

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	DAQ	—	—	**Sample Rate**	**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.
	DAQ	CCP	—	**DTO ID**	Sets the **DTO ID** (**D**ata **T**ransmission **O**bject) which is used by the ECU to send DAQ list data to the CCP master.

Characteristic-Specific Properties

Table 5-8. Characteristic-Specific Property Value Types for the PropertyID Input Value

Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Characteristic	—	—	**X Axis**	Sets the X-axis values on which the Characteristic is defined. The Characteristic dimension must be at least 1.
	Characteristic	—	—	**Y Axis**	Sets the Y-axis values on which the Characteristic is defined. The Characteristic dimension must be 2.
	Characteristic	—	—	**Byte Order**	Sets the specified byte order of the entire characteristic: 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-Specific Properties

Table 5-9. Measurement-Specific Property Value Types for the PropertyID Input Value

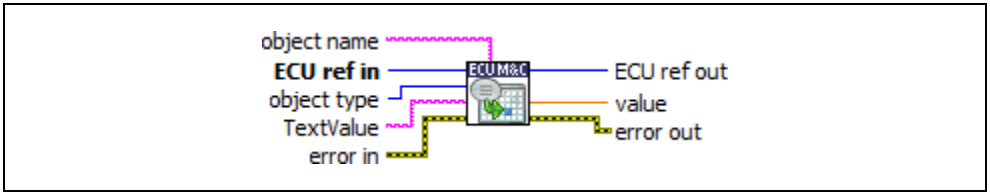
Type	Hierarchy	Sub-Hierarchy		Param	Description
		Sub 1	Sub 2		
	Measurement	—	—	**Byte Order**	Sets the specified byte order of the selected Measurement: 0—**Intel format** Bytes are in little-endian order, with least-significant bit first. 1—**Motorola format** Bytes are in big-endian order, with most-significant bit first.

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.

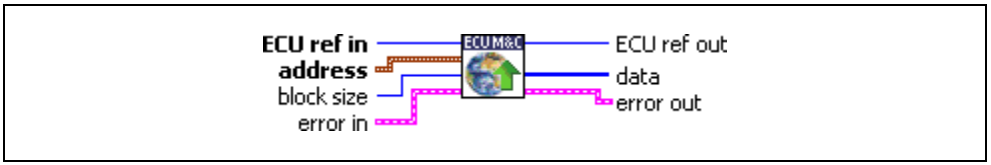
You can use this especially if the measurement or characteristic has an associated enumeration or range text type scaling before writing the double values to the ECU, using the regular Write VIs (**MC DAQ Write.vi**, **MC Measurement Write.vi**, **MC Characteristic Write.vi**).

MC Upload.vi

Purpose

Uploads data from 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 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.

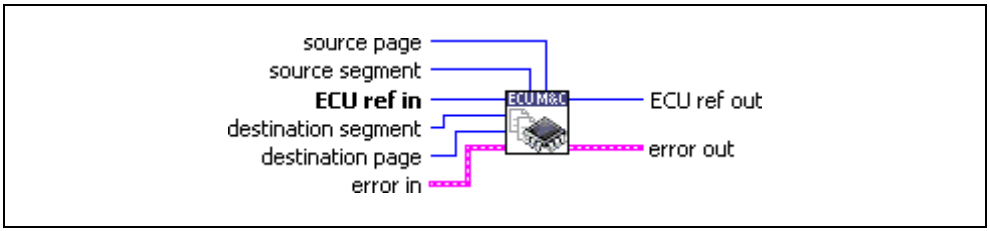
For further information on how to upload data and to use the **MC Upload.vi** command refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

MC XCP Copy Cal Page.vi

Purpose

Forces a copy transaction of one calibration page to another.

Format



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.

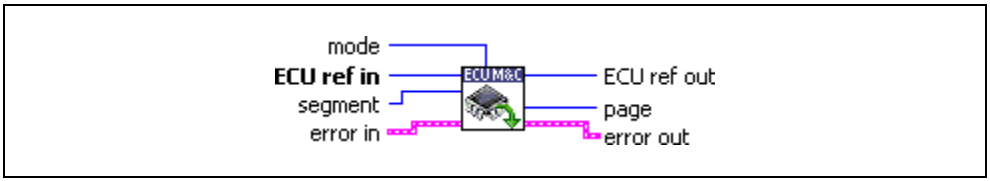
Refer to the ASAM *XCP Part 2 Protocol Layer Specification* for more information on how to set up a request.

MC XCP Get Cal Page.vi

Purpose

Queries a calibration page setting.

Format



Input

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.

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.

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.

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.

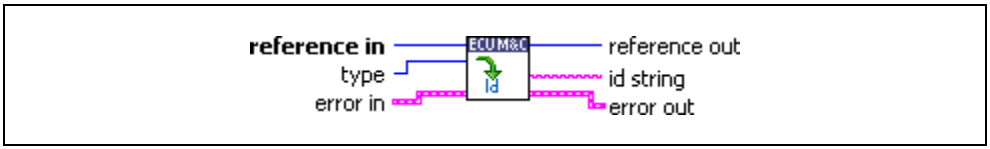
Refer to the ASAM *XCP Part 2 Protocol Layer Specification* for more information on how to set up a request.

MC XCP Get ID.vi

Purpose

Queries session configuration or slave device identification.

Format



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:

Type	Description
0	ASCII text
1	ASAM-MC2 filename without path and extension
2	ASAM-MC2 filename with path and extension
3	URL where the ASAM-MC2 file can be found
4	ASAM-MC2 file to upload
128..255	User defined

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.

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

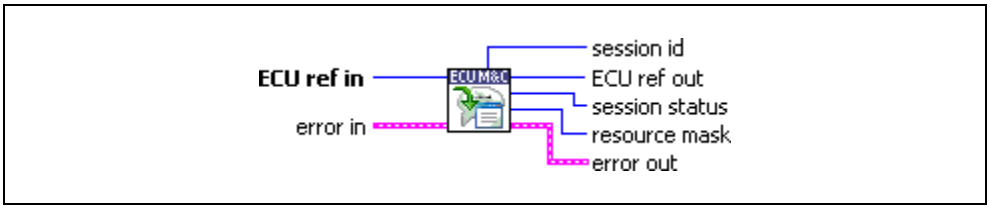
MC XCP Get ID.vi implements the XCP command GET_ID and returns session configuration or slave device identification information of the selected ECU slave device. The supported types are implementation specific of the ECU slave device. The identification string is ASCII text format.

MC XCP Get Status.vi

Purpose

Queries the current session status from an ECU slave device.

Format



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:

Bit Number	Flag	Description
0	STORE_CAL_REQ	REQuest to STORE CALibration data: 0—STORE_CAL_REQ mode is reset. 1—STORE_CAL_REQ mode is set.
1	Unused	—
2	STORE_DAQ_REQ	REQuest to STORE DAQ list: 0—STORE_DAQ_REQ mode is reset. 1—STORE_DAQ_REQ mode is set.

Bit Number	Flag	Description
3	CLEAR_DAQ_REQ	REQuest to CLEAR DAQ configuration: 0—CLEAR_DAQ_REQ is reset. 1—CLEAR_DAQ_REQ is set.
4	Unused	—
5	Unused	—
6	DAQ_RUNNING	Data Transfer: 0—The data transfer is not running. 1—The data transfer is running.
7	RESUME	RESUME Mode: 0—The slave device is not in RESUME mode. 1—The slave device is in RESUME mode.

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.

Resource mask contains the current resource protection status as a bit mask described below:

Bit Number	Flag	Description
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	—
6	Unused	—
7	Unused	—

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.

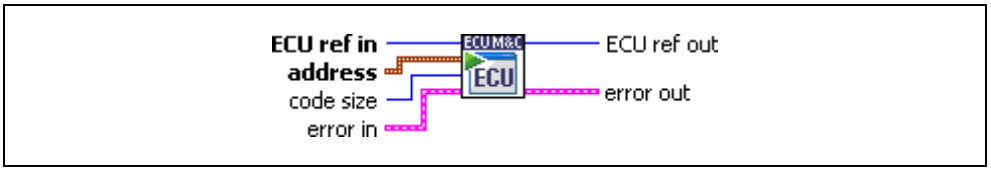
The parameter **Session Id** contains the Session configuration ID. The session configuration ID must be set by a prior **MC XCP Set Request.vi** call with STORE_DAQ_REQ set. This allows the master device to verify that automatically started DAQ lists contain the expected data transfer configuration.

MC XCP Program Prepare.vi

Purpose

Prepares the programming of non volatile memory.

Format



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.

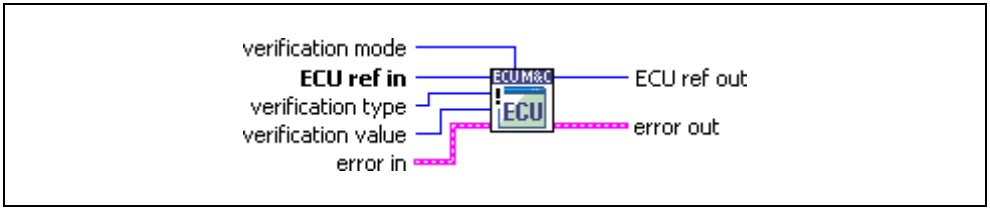
MC XCP Program Prepare.vi implements the optional XCP PROGRAM_PREPARE command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

MC XCP Program Verify.vi

Purpose

Performs a non-volatile memory certification task on the ECU device.

Format



Input

U8

Verification mode specifies the type of the requested identification:

Type	Description
0	Request to start internal routine.
1	Send a Verification Value stored in **Verification value**.

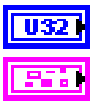
U32

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.

U16

Verification type specifies the Verification Type of the requested program verification. The Verification Type is a bit mask described below:

Verification Type	Description
0x0001	Calibration area(s) of the flash.
0x0002	Code area(s) of the flash.
0x0004	Complete flash content.
0x0008 ... 0x0080	Reserved.
0x0100 ... 0xFF00	User defined.



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

project-specific. The master receives the **Verification mode** from the project-specific programming flow control and passes it to the slave.

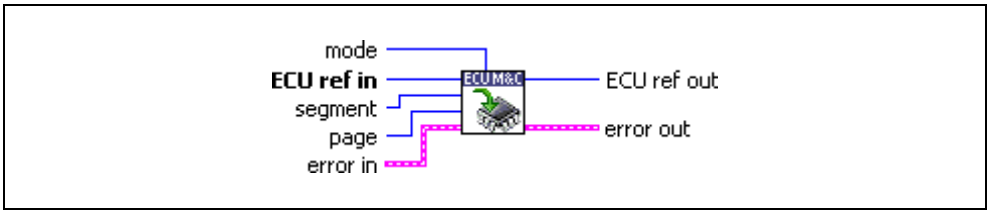
MC XCP Program Verify.vi implements the optional XCP PROGRAM_VERIFY command defined by the XCP specification. For further information on how to program non-volatile ECU memory refer to the ASAM *XCP Part 2 Protocol Layer Specification*.

MC XCP Set Cal Page.vi

Purpose

Sets a calibration page.

Format



Input

U8

Mode is a bit mask described below:

Bit	Description
0	The given page is used by the slave device application.
1	The slave device XCP driver will access the given page.
2	Unused.
3	Unused.
4	Unused.
5	Unused.
6	Unused.
7	The logical segment number is ignored. The command applies to all segments.

U32

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.

U8

Segment specifies the selected logical data segment number.

U8

Page specifies the logical data page number.

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 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

Mode is a bit mask described below:

Bit	Description
0	Request to store calibration data in non-volatile memory.
1	Unused.
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.
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.
4	Unused.
5	Unused.
6	Unused.
7	Unused.

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.

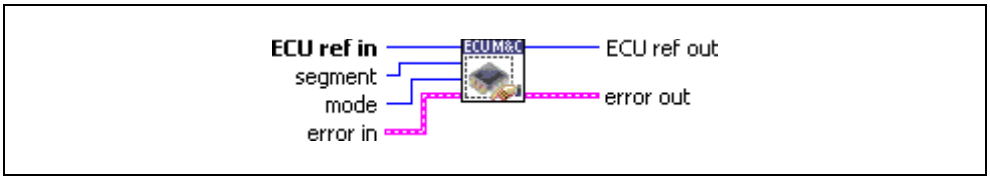
Refer to the ASAM *XCP Part 2 Protocol Layer Specification* for more information on how to set up a request.

MC XCP Set Segment Mode.vi

Purpose

Sets the mode of a specified segment.

Format



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.

6

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 6-1. Data Types for the ECU M&C API for C

Data Type	Purpose
i8	8-bit signed integer
i16	16-bit signed integer
i32	32-bit signed integer

Table 6-1. Data Types for the ECU M&C API for C (Continued)

Data Type	Purpose
u8	8-bit unsigned integer
u16	16-bit unsigned integer
u32	32-bit unsigned integer
f32	32-bit floating-point number
f64	64-bit floating-point number
str	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&C API as a pointer to a string, as `char*`.
cstr	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&C API as a pointer to a string, as `const char*`.
mcTypeTaskRef	Reference to an initialized database task, ECU task, or Measurement task.
mcAddress	C struct which represents the target address for a specific CCP operation in the ECU.

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

Function	Purpose
`mcBuildChecksum`	Calculates a checksum over a defined memory range within the ECU.
`mcCalculateChecksum`	Calculates the checksum of a data block in memory.
`mcCCPActionService`	Calls an implementation-specific action service on the ECU.
`mcCCPDiagService`	Calls an implementation-specific diagnostic service on the ECU.
`mcCCPGetActiveCalPage`	Retrieves the ECU Memory Transfer Address pointer to the calibration data page.
`mcCCPGetResult`	Uploads data from the ECU when the Memory Transfer Address pointer 0 (MTA0) has been set.

Table 6-2. Functions for the ECU M&C API for C (Continued)

Function	Purpose
mcCCPGetSessionStatus	Retrieves the current status of the Calibration Session.
mcCCPGetVersion	Retrieves CCP version implemented in the ECU.
mcCCPMoveMemory	Moves a memory block on the ECU.
mcCCPSelectCalPage	Sets the specified address to be the start address of the calibration data page.
mcCCPSetSessionStatus	Updates the ECU with the current state of the calibration session.
mcCharacteristicRead	Reads all data from a named Characteristic on the ECU which is identified by the ECU Reference handle.
mcCharacteristicReadSingleValue	Reads a single value from a named Characteristic on the ECU which is identified by the ECU Reference handle.
mcCharacteristicWrite	Downloads data to a Characteristic for a selected ECU.
mcCharacteristicWriteSingleValue	Writes a single value to a named Characteristic on the ECU.
mcClearMemory	Clears the contents of the specified ECU memory.
mcConversionCreate	Creates a signal conversion object in memory.
mcDAQClear	Stops communication for the Measurement task and clears the task.
mcDAQInitialize	Initializes a Measurement task for the specified Measurement channel list.
mcDAQListInitialize	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.
mcDAQRead	Reads samples from a Measurement task. Samples are obtained from received CAN messages.
mcDAQReadTimestamped	Reads timestamped samples from a DAQ task initialized with the selected mode of mcDAQModeDAQListTimeStamped.
mcDAQStartStop	Starts or stops the transmission of the DAQ lists for the specified Measurement task.
mcDAQWrite	Writes samples to an ECU DAQ list.

Table 6-2. Functions for the ECU M&C API for C (Continued)

Function	Purpose
mcDatabaseClose	Stops transmission of the DAQ lists for the specified Measurement task.
mcDatabaseClose	Closes a specified A2L Database reference.
mcDatabaseOpen	Opens a specified A2L Database.
mcDatabaseOpenEx	Creates an A2L database in memory, for using the ECU M&C Toolkit functions without access to a valid ASAM A2L file.
mcDoubleToText	Converts a numerical value to a text string using an enumeration or range text type scaling.
mcDownload	Downloads data to an ECU.
mcECUConnect	Establishes communication to the selected ECU through CCP. After a successful ECU Connect you can create a Measurement Task or read/write a Characteristic.
mcECUCreate	Creates an ECU object in memory.
mcECUDeselect	Deselects an ECU and invalidates the ECU reference handle.
mcECUDisconnect	Disconnects CCP communication to the selected ECU.
mcECUSelectEx	Selects an ECU from the names stored in an A2L database.
mcECUSetCalibrationPage	Sets the appropriate RAM or ROM calibration page on the ECU.
mcEventCreate	Creates an Event object in memory.
mcGeneric	Sends a generic CCP command.
mcGetNames	Retrieves a comma-separated list of ECU names, Measurement names, Characteristic names, Event names, Characteristic pages, or Group names from a specified A2L database.
mcGetNamesLength	Retrieves the amount of memory required to store the names returned by mcGetNames.
mcGetProperty	Retrieves a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.
mcMeasurementCreate	Creates a Measurement object in memory.
mcMeasurementRead	Reads a single Measurement value from the ECU.
mcMeasurementWrite	Writes a single Measurement value to the ECU.

Table 6-2. Functions for the ECU M&C API for C (Continued)

Function	Purpose
mcProgram	Programs a memory block on the ECU.
mcProgramReset	Indicates the end of a programming sequence.
mcProgramStart	Indicates the start of a programming sequence.
mcSetProperty	Sets a property of the driver, the database, the ECU, a Characteristic, a Measurement, or a Measurement task.
mcStatusToString	Converts a status code into a descriptive string.
mcTextToDouble	Converts a text string to a numerical value using an enumeration or range text scaling.
mcUpload	Uploads data from an ECU.
mcXCPCopyCalPage	Forces a copy transaction of one calibration page to another.
mcXCPGetCalPage	Queries a calibration page setting.
mcXCPGetId	Queries session configuration or slave device identification.
mcXCPGetStatus	Queries the current session status from an ECU slave device.
mcXCPProgramPrepare	Prepares the programming of non volatile memory.
mcXCPProgramVerify	Performs a non-volatile memory certification task on the ECU device.
mcXCPSetCalPage	Sets a calibration page.
mcXCPSetRequest	Performs a request to save session and device information to non-volatile memory.
mcXCPSetSegmentMode	Sets the mode of a specified segment.

mcBuildChecksum

Purpose

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

Format

```
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:

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

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:

Name	Width	Poly	Init	Refin	Refout	XORout
XCP_CRC_16	16	0x8005	0x0000	TRUE	TRUE	0x0000
XCP_CRC16_CITT	16	0x1021	0xFFFF	FALSE	FALSE	0x0000
XCP_CRC_32 32	32	0x04C11DB7	0xFFFFFFFF	TRUE	TRUE	0xFFFFFFFF

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

```
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:

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

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` — `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

`ServiceNo` — `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` — `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` — `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` — `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

```
mcTypeStatus        mcCCPGetActiveCalPage(
                        mcTypeTaskRef ECURefNum,
                        mcAddress *Address);
```

Input

`ECURefNum` — `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. `mcAddress` is a C struct consisting of:

Address
Specifies the address part of the address.

Extension
Extension contains the extension part of the address.

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 `mcCCPDiagService` 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

```
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

```
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

```
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

```
mcTypeStatus        mcCCPSetSessionStatus(
                        mcTypeTaskRef ECURefNum,
                        u8 SessionStatus);
```

Input

`ECURefNum`	`ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
`SessionStatus`	`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

Bit	Name	Description
0	CAL	Calibration data initialized.
1	DAQ	DAQ list(s) initialized.
2	RESUME	Request to save DAQ set-up during shutdown in CCP slave. CCP slave automatically restarts DAQ after start-up.

Table 6-3. Bit Mask Assignments for Calibration Session Status (Continued)

Bit	Name	Description
3	Reserved	—
4	Reserved	—
5	Reserved	—
6	STORE	Request to save calibration data during shut-down in CCP slave.
7	RUN	Session in progress.

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

```
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.

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

```
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

```
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. `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.

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

```
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, `mcSetProperty` 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 **D**ata **T**ransmission **O**bject (**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

```
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 **D**ata **T**ransmission **O**bject (**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

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)

format. 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:

```
__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)

format. 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 waits until `NumberOfSamplesToRead` messages have been received. 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. To avoid blocking a mcDAQReadTimestamped function, read the mcPropDAQ_SamplesPending property to check the number of collected sample points before calling mcDAQReadTimestamped.

mcDAQStartStop

Purpose

Starts the transmission of the DAQ lists assigned to the Measurement task on the ECU.

Format

```
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

```
mcTypeStatus        mcDAQWrite(
                        mcTypeTaskRef DAQRefNum,
                        u32 NumberofSamplesToWrite,
                        f64 *SampleArray);
```

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

```
mcTypeStatus        mcDatabaseClose(
                        mcTypeTaskRef *DBRefNum);
```

Input

`DBRefNum` — `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

```
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

```
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 8000*XXYY* 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:

- CAN*xx*
- 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, *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*).

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

```
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

```
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*xx*
- 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 compability 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*).

mcECUSetCalibrationPage

Purpose

Sets the appropriate RAM or ROM calibration page on the ECU.

Format

```
mcTypeStatus        mcECUSetCalibrationPage (
                        mcTypeTaskRef ECURefNum,
                        u8 UseRAM,
                        u8 mapAddresses);
```

Input

`ECURefNum`	`ECURefNum` is the task reference that links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.
`UseRAM`	0: Select ROM calibration page. 1: Select RAM calibration page.
`mapAddresses`	0: Do not map addresses. 1: Map addresses from ROM to the page specified in `UseRAM`.

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

`mcECUSetCalibrationPage` tries to identify a single RAM or ROM page on the ECU and select it according to the `UseRAM` input.

To identify an appropriate page, the function searches for the calibration page information from the A2L file or online information from the ECU. If the function identifies a unique calibration page, it is activated in the ECU, and the function returns success.

If the function does not identify a unique page, an error indicating this is returned, and no further action is taken. This does not state a fault, but just the algorithm's inability to uniquely

identify the desired page. In this case, you can use the calibration page-related ECU properties (`mcGetProperty`, **ECU»CCP»Cal Pages»…** or **ECU»XCP»Cal Pages»…**) to gain the information about available calibration pages, and manually select the correct page using `mcCCPSelectCalPage` or `mcXCPSetCalPage`.

The `mapAddresses` parameter activates the address mapping from the ROM page, assumed to be the reference page to the target page specified in `UseRAM`. Address mapping is supported for only the CCP protocol and requires a unique ROM and unique RAM page in the A2L file. Addresses of measurements and characteristics in the A2L file must point to the ROM page as a reference page.

mcEventCreate

Purpose

Creates an Event object in memory.

Format

```
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

```
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 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.
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

```
mcTypeStatus        mcGetNamesLength(
                        mcTypeTaskRef RefNum,
                        u32 Type,
                        cstr ECUName,
                        u32 *SizeOfNamesList);
```

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.

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

```
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

Data Type	Name	Description
u32	mcPropCANBaudRate	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.
u8	mcPropCANTermination	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.)
u32	mcPropChar_Address	Returns the address of the selected Characteristic in the memory of the ECU.
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)

Data Type	Name	Description
u32	mcPropChar_Dimension	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.
u8	mcPropChar_Extension	Returns additional address information. For instance it can be used to distinguish different address spaces of an ECU (multi-microcontroller devices).
f64	mcPropChar_Maximum	Returns the Maximum value of the Characteristic.
f64	mcPropChar_Minimum	Returns the Minimum value of the Characteristic.
u32	mcPropChar_ReadOnly	Returns if a Characteristic is set to read only. In this case it is not allowed to call mcCharacteristicWrite for this Characteristic.
u32	mcPropChar_Sizes	Returns the Array Sizes for the X and Y directions of the Characteristic.
str	mcPropChar_Unit	Returns the unit string defined for this Characteristic in the A2L database.
u32	mcPropChar_Unit_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropChar_Unit.
f64	mcPropChar_X_Axis	Returns X-axis values on which the Characteristic is defined. Valid if the selected Characteristic is 1- or 2-dimensional.
f64	mcPropChar_Y_Axis	Returns Y-axis values on which the Characteristic is defined, Valid if the selected Characteristic is 2-dimensional.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
f64	`mcPropChar_Scale_Factor`	Returns the scaling factor defined for this Characteristic in the A2L database.
f64	`mcPropChar_Scale_Offset`	Returns the scaling offset defined for this Characteristic in the A2L database.
u32	`mcPropChar_Scale_Type`	Returns the scaling type defined for this Characteristic in the A2L database. **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)

Data Type	Name	Description
		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)

Data Type	Name	Description
u32	mcPropChar_Scale_TextValues_Size	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).
str	mcPropChar_Scale_TextValues	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.
u32	mcPropCmd_EXCHANGE_ID	Returns whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.
u32	mcPropCROID	Returns the CRO CAN Identifier (**C**ommand **R**eceive **O**bject) for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.
u32	mcPropDAQ_DTO_ID	Returns the DTO ID (**D**ata **T**ransmission **O**bject) which is used by the ECU to respond to send data from the DAQ lists to the CCP master.
nctType Taskref	mcPropDAQ_DTO_Task	NI-CAN task reference to the CAN Task assigned to the DTO ID of the Measurement task.
str	mcPropDAQ_EventChannelName	Returns the selected event channel name to which the Measurement task is assigned.
u32	mcPropDAQ_EventChannelName_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropDAQ_EventChannelName.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropDAQ_Mode	Returns the selected mode of an M&C Measurement task. 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 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. Use mcDAQRead 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 mcDAQRead is called.
u32	mcPropDAQ_NumChannels	Returns the number of channels initialized in a DAQ channel list of a M&C Measurement task. This is the number of array entries required when using mcDAQRead.
u16	mcPropDAQ_Prescaler	Prescaler for the Measurement task on the ECU.
f64	mcPropDAQ_SampleRate	Returns the selected Sample Rate in Hz for the M&C Measurement task.
u32	mcPropDAQ_Samples Pending	Returns the number of samples available for read in DAQ tasks defined with sample rate > 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.
f64	mcPropDAQ_TimeSince LastFrame	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.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
str	mcPropDB_Filename	Returns the A2L Database file name with which the task has been opened. The value of this property cannot be changed using mcSetProperty.
u32	mcPropDB_Filename_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropDB_Filename.
u32	mcPropDTOID	Returns the DTO CAN Identifier (**D**ata **T**ransfer **O**bject) for CCP or XCP on CAN which is used to send commands and data from the slave device to the host.
u32	mcPropECU_BaudRate	Returns the baud rate in use.
u32	mcPropECU_ByteOrder	Returns 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.
u32	mcPropECU_CCP_NumPages	Returns the number of DEFINED_PAGES structures for this ECU in the A2L file.
u32	mcPropECU_CCP_PageNo	Returns the page number of the page selected with the Name input.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropECU_CCP_PageFlags	Returns the page flags of the page selected with the Name input. The value returned is a bitmask ored from the following values: 1 RAM page 2 ROM page 4 FLASH page 8 EEPROM page 16 RAM_INIT_BY_ECU RAM page initialized at ECU startup. 32 RAM_INIT_BY_TOOL RAM page that the calibration tool initializes. 64 AUTO_FLASH_BACK RAM page automatically flashed back. 128 FLASH_BACK RAM page that the calibration tool can flash back. 256 DEFAULT Page is standard (fallback).
u32	mcPropECU_CCP_PageAddress	Returns the memory address of the page selected with the Name input.
u8	mcPropECU_CCP_PageAddressExtension	Returns the memory address extension of the page selected with the Name input.
str	mcPropECU_Checksum	Returns the file name of the Checksum DLL used for verifying the checksum.
u32	mcPropECU_Checksum_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_Checksum.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropECU_CmdByteOrder	Returns 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.
u32	mcPropECU_CRO_ID	Returns the **CRO ID** (**C**ommand **R**eceive **O**bject) which is used to send commands and data from the host to the slave device.
nctType Taskref	mcPropECU_CRO_Task	NI-CAN Task reference to the CAN Task assigned to the CRO ID.
u32	mcPropECU_DTO_ID	Returns the **DTO ID** (**D**ata **T**ransmission **O**bject) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.
nctType Taskref	mcPropECU_DTO_Task	NI-CAN Task reference to the CAN Task assigned to the DTO ID.
i32	mcPropECU_EventChannel	Translates the event channel name to the event channel number. Pass the event channel name in the Name parameter of GetProperty.
[u8]	mcPropECU_ID	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.
u8	mcPropECU_ID_DataType	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.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	mcPropECU_ID_Length	Returns the length of the slave device identifier in bytes.
u32	mcPropECU_Interface	Returns the interface initialized for the task, such as with mcDAQInitialize.
[u8]	mcPropECU_MasterID	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.
str	mcPropECU_Name	Returns the name of the selected ECU opened by mcECUSelectEx.
str	mcPropECU_Comment	Returns the comment of the selected ECU opened by mcECUSelectEx.
u32	mcPropECU_Comment_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_Comment.
u8	mcPropECU_XCP_ NumSegments	Returns the number of XCP memory segments found for this ECU.
u8	mcPropECU_XCP_NumPages	Returns the number of memory pages defined for the memory segment specified in the Name input. Specify the segment by the string *SEGMENT[<n>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the mcPropECU_XCP_NumSegments property).
u8	mcPropECU_XCP_ AddressExtension	Returns the memory address extension for the memory segment specified in the Name input. Specify the segment by the string *SEGMENT[<n>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the mcPropECU_XCP_NumSegments property).

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	`mcPropECU_XCP_CompressionMethod`	Returns the compression method for the memory segment specified in the `Name` input. A value of 0 means no compression. Other values are user defined. Specify the segment by the string *SEGMENT[<n>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the `mcPropECU_XCP_NumSegments` property).
u8	`mcPropECU_XCP_EncryptionMethod`	Returns the encryption method for the memory segment specified in the `Name` input. A value of 0 means no encryption. Other values are user defined. Specify the segment by the string *SEGMENT[<n>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the `mcPropECU_XCP_NumSegments` property).
u8	`mcPropECU_XCP_PageNo`	Returns the logical page number for the memory segment page specified in the `Name` input. Specify the page by the string *SEGMENT[<n>]PAGE[<m>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the `mcPropECU_XCP_NumSegments` property) and <*m*> is the decimal representation of the page number within the segment (0..*M*-1, where *M* is the number returned from the `mcPropECU_XCP_NumPages` property for this segment).

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	`mcPropECU_XCP_PageECUAccess`	Returns a flag indicating ECU access rights for the memory segment page specified in the `Name` input. Defined values are: 0 ECU access not allowed 1 ECU access allowed without XCP access only 2 ECU access allowed with XCP access only 3 ECU access allowed always Specify the page by the string *SEGMENT[<n>]PAGE[<m>]*, where <*n*> is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the `mcPropECU_XCP_NumSegments` property) and <*m*> is the decimal representation of the page number within the segment (0..*M*-1, where *M* is the number returned from the `mcPropECU_XCP_NumPages` property for this segment).

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	`mcPropECU_XCP_PageXCPReadAccess`	Returns a flag indicating XCP Read access rights for the memory segment page specified in the `Name` input. Defined values are: 0 XCP Read access not allowed 1 XCP Read access allowed without ECU access only 2 XCP Read access allowed with ECU access only 3 XCP Read access allowed always Specify the page by the string *SEGMENT[<n>]PAGE[<m>]*, where *<n>* is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the `mcPropECU_XCP_NumSegments` property) and *<m>* is the decimal representation of the page number within the segment (0..*M*-1, where *M* is the number returned from the `mcPropECU_XCP_NumPages` property for this segment).

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	mcPropECU_XCP_PageXCPWriteAccess	Returns a flag indicating XCP Write access rights for the memory segment page specified in the Name input. Defined values are: 0 XCP Write access not allowed 1 XCP Write access allowed without ECU access only 2 XCP Write access allowed with ECU access only 3 XCP Write access allowed always Specify the page by the string *SEGMENT[<n>]PAGE[<m>]*, where *<n>* is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the mcPropECU_XCP_NumSegments property) and *<m>* is the decimal representation of the page number within the segment (0..*M*-1, where *M* is the number returned from the mcPropECU_XCP_NumPages property for this segment).
u8	mcPropECU_XCP_PageInitSegment	Returns the number of the segment that initializes the memory segment page specified in the Name input. Specify the page by the string *SEGMENT[<n>]PAGE[<m>]*, where *<n>* is the decimal representation of the segment number (0..*N*-1, where *N* is the number returned from the mcPropECU_XCP_NumSegments property) and *<m>* is the decimal representation of the page number within the segment (0..*M*-1, where *M* is the number returned from the mcPropECU_XCP_NumPages property for this segment).
[u16]	mcPropECU_DAQListNumbers	Returns an array of DAQ list numbers for all DAQ lists defined in the A2L file.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropECU_TimingFactor	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.
u16	mcPropDAQList_Max Length	Returns the maximum length of the DAQ list.
u32	mcPropDAQList_CANId SelectMode	Returns how to select the CAN ID for the specified DAQ list: 0—CAN_ID_FIXED The CAN Identifier is a predefined fixed number. 1—CAN_ID_VARIABLE The CAN Identifier is a variable number. 2—CAN_ID_DTO_ID The CAN Identifier is the same as the DTO identifier.
u32	mcPropDAQList_CANId	Returns the CAN ID for the specified DAQ list if mcPropDAQList_CANIdSelectMode == CAN_ID_FIXED.
u8	mcPropDAQList_FirstPID	Returns the first Packet ID for the specified DAQ list.
u32	mcPropDAQList_NumberOf EventChannels	Returns the number of allowed event channels for the specified DAQ list.
u32	mcPropDAQList_ ReductionAllowed	Returns whether or not the specified DAQ list allows reduction.
u32	mcPropDAQList_NumberOf ExcludedDAQLists	Returns the length of the array containing the numbers of DAQ lists not working together with the current DAQ list.
u16	mcPropDAQList_Excluded DAQLists	Returns an array containing the numbers of DAQ lists not working together with the current DAQ list.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
str	mcPropDAQList_Name	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.
str	mcPropDAQList_Name_Size	Call this property before calling mcPropDAQList_Name to find the amount of storage needed to get the name value.
str	mcPropDAQList_DisplayIdentifier	Optional property you can use as a display name as an alternative to the DAQList_Name property.
str	mcPropDAQList_DisplayIdentifier_Size	Call this property before calling mcPropDAQList_DisplayIdentifier to find the amount of storage needed to get the display identifier value.
u32	mcPropECU_Name_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_Name.
str	mcPropECU_SeedChkDllPath	Determines the directory where the ECU M&C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&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.
str	mcPropECU_SeedChkDllPath_Size	Returns the required buffer size to read the mcPropECU_SeedChkDllPath property.
str	mcPropECU_SeedKey_Cal	Returns the filename of the SeedKey DLL used for Calibration purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
u32	mcPropECU_SeedKey_Cal_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_Cal.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
str	mcPropECU_SeedKey_DAQ	Returns the filename of the SeedKey DLL used for DAQ purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
u32	mcPropECU_SeedKey_DAQ_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_DAQ.
str	mcPropECU_SeedKey_Prog	Returns the file name of the SeedKey DLL used for programming purposes. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
u32	mcPropECU_SeedKey_Prog_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_Prog.
str	mcPropECU_SeedKey_XCP	Returns the file name of the SeedKey DLL for XCP. If SeedKey is configured for remote access, the output is *RSK:<server ip address>,<port>*.
str	mcPropECU_LogFileName	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.
u32	mcPropECU_LogFileName_Size	Returns the size of the buffer needed to retrieve the mcPropECU_LogFileName property.
u32	mcPropECU_SeedKey_XCP_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropECU_SeedKey_XCP.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u8	mcPropECU_Single_Byte_DAQ_Lists	Determines if an ECU supports single-byte or multi-byte DAQ list entries.
u32	mcPropECU_Station Address	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.
u32	mcPropECU_XCP_Timeout_T1 mcPropECU_XCP_Timeout_T2 mcPropECU_XCP_Timeout_T3 mcPropECU_XCP_Timeout_T4 mcPropECU_XCP_Timeout_T5 mcPropECU_XCP_Timeout_T6 mcPropECU_XCP_Timeout_T7	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.
u32	mcPropECU_CCP_Timeout_T_std	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.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropECU_CCP_Timeout_T_pgm	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.
u32	mcPropECU_CCP_Timeout_T_mem	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.
u32	mcPropECU_CCP_Timeout_T_diag	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.
u32	mcPropECU_CCP_Timeout_T_act	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.
u32	mcPropGen_Version_Build	Returns the build number of the ECU M&C software. This number applies to Development, Alpha, and Beta phase only, and should be ignored for Release phase.
str	mcPropGen_Version_Comment	Returns a comment string for the ECU M&C software. If you received a custom release of ECU M&C from National Instruments, this comment often describes special features of the release.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropGen_Version_Comment_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropGen_Version_Comment.
u32	mcPropGen_Version_Major	Returns the major version of the ECU M&C software, such as the 1 in version 1.2.5.
u32	mcPropGen_Version_Minor	Returns the minor version of the ECU M&C software, such as the 2 in version 1.2.5.
u32	mcPropGen_Version_Update	Returns the update version of the ECU M&C software, such as the 5 in version 1.2.5.
str	mcPropIPAddress	Returns the IP address for XCP on Ethernet (TCP or UDP) as a string.
u32	mcPropIPAddress_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropIPAddress.
u16	mcPropIPPort	Returns the IP port for XCP on Ethernet (TCP or UDP).
u32	mcPropMeas_Address	Returns the address of the selected Measurement in the memory of the control unit.
u32	mcPropMeas_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	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).

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropMeas_IsVirtual	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.
f64	mcPropMeas_Maximum	Returns the maximum value of the Measurement.
f64	mcPropMeas_Minimum	Returns the minimum value of the Measurement.
u32	mcPropMeas_ReadOnly	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.
str	mcPropMeas_Unit	Returns the unit string defined for this Measurement in the A2L database.
u32	mcPropMeas_Unit_Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropMeas_Unit.
f64	mcPropMeas_Scale_Factor	Returns the scaling factor defined for this Measurement in the A2L database.
f64	mcPropMeas_Scale_Offset	Returns the scaling offset defined for this Measurement in the A2L database.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	`mcPropMeas_Scale_Type`	Returns the scaling type defined for this Measurement in the A2L database. **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. **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.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
		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.
u32	`mcPropMeas_Scale_TextValues_Size`	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).
str	`mcPropMeas_Scale_TextValues`	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.
u32	`mcPropOptCmd_ACTION_SERVICE`	Returns whether the ECU supports the optional CCP Command ACTION_SERVICE.
u32	`mcPropOptCmd_BUILD_CHKSUM`	Returns whether the ECU supports the optional CCP Command BUILD_CHKSUM.
u32	`mcPropOptCmd_CLEAR_MEMORY`	Returns whether the ECU supports the optional CCP Command CLEAR_MEMORY.
u32	`mcPropOptCmd_DIAG_SERVICE`	Returns whether the ECU supports the optional CCP Command DIAG_SERVICE.
u32	`mcPropOptCmd_DNLOAD_6`	Returns whether the ECU supports the optional CCP Command DNLOAD_6.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
u32	mcPropOptCmd_GET_ACTIVE_CAL_PAGE	Returns whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_PAGE.
u32	mcPropOptCmd_GET_S_STATUS	Returns whether the ECU supports the optional CCP Command GET_S_STATUS.
u32	mcPropOptCmd_GET_SEED	Returns whether the ECU supports the optional CCP Command GET_SEED.
u32	mcPropOptCmd_MOVE	Returns whether the ECU supports the optional CCP Command MOVE.
u32	mcPropOptCmd_PROGRAM	Returns whether the ECU supports the optional CCP Command PROGRAM.
u32	mcPropOptCmd_PROGRAM_6	Returns whether the ECU supports the optional CCP Command PROGRAM_6.
u32	mcPropOptCmd_SELECT_CAL_PAGE	Returns whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.
u32	mcPropOptCmd_SET_S_STATUS	Returns whether the ECU supports the optional CCP Command SET_S_STATUS.
u32	mcPropOptCmd_SHORT_UP	Returns whether the ECU supports the optional CCP Command SHORT_UP.
u32	mcPropOptCmd_START_STOP_ALL	Returns whether the ECU supports the optional CCP Command START_STOP_ALL.
u32	mcPropOptCmd_TEST	Returns whether the ECU supports the optional CCP Command TEST.
u32	mcPropOptCmd_UNLOCK	Returns whether the ECU supports the optional CCP Command UNLOCK.

Table 6-4. Values for PropertyID (Continued)

Data Type	Name	Description
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.
str	mcPropGroup_Comment	Returns the comment of the selected Group.
u32	mcPropGroup_Comment_ Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropGroup_Comment.
str	mcPropFunction_Comment	Returns the comment of the selected Function.
u32	mcPropFunction_Comment_ Size	Returns the number of bytes to be allocated if you call mcGetProperty with the parameter mcPropFunction_Comment.

mcMeasurementCreate

Purpose

Creates a Measurement object in memory.

Format

```
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.

DataType	Data Format
0	Unsigned byte
1	Signed byte
2	Unsigned word
3	Signed word
4	Unsigned long
5	Signed long
6	Float 32

`DataSize` Sets the size of the measurement data and corresponds to the selected `DataType`.

Data Format	DataSize
Unsigned byte	1
Signed byte	1
Unsigned word	2
Signed word	2
Unsigned long	4
Signed long	4
Float 32	4

`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

```
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

```
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
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

```
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

```
mcTypeStatus        mcProgramStart(
                        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, `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

Data Type	Name	Description
u32	mcPropCANBaudRate	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.
u8	mcPropCANTermination	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.)
u32	mcPropCmd_EXCHANGE_ID	Sets whether or not the EXCHANGE_ID command should be suppressed during connection to the ECU.
u32	mcPropCROID	Sets the CRO CAN Identifier (**C**ommand **R**eceive **O**bject) for CCP or XCP on CAN which is used to send commands and data from the host to the slave device.
u32	mcPropDTOID	Sets the DTO CAN Identifier (**D**ata **T**ransfer **O**bject) for CCP or XCP on CAN which is used to send commands and data from the slave device to the host.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
u32	mcPropECU_BaudRate	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 8000*XXYY* 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 mcSetProperty.
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)

Data Type	Name	Description
u32	`mcPropECU_CRO_ID`	Sets the CAN identifier for the CRO ID (**C**ommand **R**eceive **O**bject), which is used to send commands and data from the host to the slave device.
u32	`mcPropECU_DTO_ID`	Sets the DTO ID (**D**ata **T**ransmission **O**bject) which is used by the ECU to respond to CCP commands and send data and status information to the CCP master.
u32	`mcPropECU_MasterID`	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.
str	`mcPropECU_SeedChkDll Path`	Determines the directory where the ECU M&C Toolkit expects to find the Seedkey or Checksum DLL. If the property is an empty string (default), the ECU M&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.
str	`mcPropECU_SeedKey_Cal`	Sets the filename of the SeedKey DLL used for Calibration purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
str	`mcPropECU_SeedKey_DAQ`	Sets the filename of the SeedKey DLL used for DAQ purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
str	`mcPropECU_SeedKey_Prog`	Sets the filename of the SeedKey DLL used for programming purposes. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.
str	`mcPropECU_SeedKey_XCP`	Sets the filename of the SeedKey DLL for XCP. For Remote Seedkey access (refer to the LabVIEW examples), set the name to *RSK:<server ip address>,<port>*.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
str	mcPropECU_LogFileName	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.
u8	mcPropECU_Single_Byte_DAQ_Lists	Sets the ECU to support single-byte or multi-byte DAQ list entries.
u32	mcPropECU_Station Address	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.
u32	mcPropECU_TimingFactor	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.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
u32	mcPropECU_XCP_Timeout_T1 mcPropECU_XCP_Timeout_T2 mcPropECU_XCP_Timeout_T3 mcPropECU_XCP_Timeout_T4 mcPropECU_XCP_Timeout_T5 mcPropECU_XCP_Timeout_T6 mcPropECU_XCP_Timeout_T7	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.
u32	mcPropECU_CCP_Timeout_T_std	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.
u32	mcPropECU_CCP_Timeout_T_pgm	Sets 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.
u32	mcPropECU_CCP_Timeout_T_mem	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.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
u32	mcPropECU_CCP_Timeout_T_diag	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.
u32	mcPropECU_CCP_Timeout_T_act	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.
str	mcPropIPAddress	Sets the IP address for XCP on Ethernet (TCP or UDP) as a string.
u16	mcPropIPPort	Sets the IP port for XCP on Ethernet (TCP or UDP).
u32	mcPropOptCmd_ACTION_SERVICE	Sets whether the ECU supports the optional CCP Command ACTION_SERVICE.
u32	mcPropOptCmd_BUILD_CHKSUM	Sets whether the ECU supports the optional CCP Command BUILD_CHKSUM.
u32	mcPropOptCmd_CLEAR_MEMORY	Sets whether the ECU supports the optional CCP Command CLEAR_MEMORY.
u32	mcPropOptCmd_DIAG_SERVICE	Sets whether the ECU supports the optional CCP Command DIAG_SERVICE.
u32	mcPropOptCmd_DNLOAD_6	Sets whether the ECU supports the optional CCP Command DNLOAD_6.
u32	mcPropOptCmd_GET_ACTIVE_CAL_PAGE	Sets whether the ECU supports the optional CCP Command GET_ACTIVE_CAL_PAGE.
u32	mcPropOptCmd_GET_S_STATUS	Sets whether the ECU supports the optional CCP Command GET_S_STATUS.
u32	mcPropOptCmd_GET_SEED	Sets whether the ECU supports the optional CCP Command GET_SEED.
u32	mcPropOptCmd_MOVE	Sets whether the ECU supports the optional CCP Command MOVE.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
u32	mcPropOptCmd_PROGRAM	Sets whether the ECU supports the optional CCP Command PROGRAM.
u32	mcPropOptCmd_PROGRAM_6	Sets whether the ECU supports the optional CCP Command PROGRAM_6.
u32	mcPropOptCmd_SELECT_CAL_PAGE	Sets whether the ECU supports the optional CCP Command SELECT_CAL_PAGE.
u32	mcPropOptCmd_SET_S_STATUS	Sets whether the ECU supports the optional CCP Command SET_S_STATUS.
u32	mcPropOptCmd_SHORT_UP	Sets whether the ECU supports the optional CCP Command SHORT_UP.
u32	mcPropOptCmd_START_STOP_ALL	Sets whether the ECU supports the optional CCP Command START_STOP_ALL.
u32	mcPropOptCmd_TEST	Sets whether the ECU supports the optional CCP Command TEST.
u32	mcPropOptCmd_UNLOCK	Sets whether the ECU supports the optional CCP Command UNLOCK.
u8	mcPropPGM_AccessMethod	Selects 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_CompressionMethod	Selects the selected compression method used for mcProgram. 0—Data is uncompressed (default). 0x80...0xFF—User defined.

Table 6-5. ECU-Specific Value Types for the PropertyID Input Value (Continued)

Data Type	Name	Description
u8	mcPropPGM_Encryption Method	Selects the selected encryption method used for mcProgram. 0—Data is not encrypted (default). 0x80...0xFF—User defined.
u8	mcPropPGM_Programming Method	Selects the selected programming method used for mcProgram. 0—Sequential programming (default). 0x80...0xFF—User defined.

Table 6-6. DAQ-Specific Value Types for the PropertyID Input Value

Data Type	Name	Description
u32	`mcPropDAQ_DTO_ID`	Sets the DTO ID (**D**ata **T**ransmission **O**bject) which is used by the ECU to respond to send data from the DAQ lists to the CCP master.
str	`mcPropDAQ_EventChannelName`	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.
i32	`mcPropDAQ_Mode`	Sets the mode of an M&C Measurement task. 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 `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. Use `mcDAQRead` 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 `mcDAQRead` is called.
u16	`mcPropDAQ_Prescaler`	Sets the Prescaler, which reduces the desired transmission frequency of the associated DAQ list.

Characteristic-Specific Properties

Table 6-7. Characteristic-Specific Value Types for the PropertyID Input Value

Data Type	Name	Description
double[]	mcPropChar_X_Axis	Sets the X-axis values on which the Characteristic is defined. The Characteristic dimension must be at least 1.
double[]	mcPropChar_Y_Axis	Sets the Y-axis values on which the Characteristic is defined. The Characteristic dimension must be 2.
u32	mcPropChar_ByteOrder	Sets the specified byte order of the selected Characteristic: 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-Specific Properties

Table 6-8. Measurement-Specific Value Types for the PropertyID Input Value

Data Type	Name	Description
u32	mcPropMeas_ByteOrder	Sets the specified byte order of the selected Measurement: 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.

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 6-9. Description of Return Codes

Status Code	Definition
Negative	Error—Function did not perform expected behavior.
Positive	Warning—Function performed as expected, but a condition arose that may require attention.
Zero	Success—Function completed successfully.

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.

```
status= ncDatabaseOpen ("TestDataBase.A2L", &MyDbHandle);
PrintStat (status, "mcOpenDatabase");
```

where the function `PrintStat` has been defined at the top of the program as:

```
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:

```
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

```
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` — `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

`Mode` — `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` — `Segment` specifies the selected logical data segment number.

Output

`Page` — `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

```
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:

Type	Description
0	ASCII text
1	ASAM-MC2 filename without path and extension
2	ASAM-MC2 filename with path and extension
3	URL where the ASAM-MC2 file can be found
4	ASAM-MC2 file to upload128..255 User defined

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

```
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:

Bit Number	Flag	Description
0	STORE_CAL_REQ	REQuest to STORE CALibration data: 0—STORE_CAL_REQ mode is reset. 1—STORE_CAL_REQ mode is set.
1	Unused	—
2	STORE_DAQ_REQ	REQuest to STORE DAQ list: 0—STORE_DAQ_REQ mode is reset. 1—STORE_DAQ_REQ mode is set.
3	CLEAR_DAQ_REQ	REQuest to CLEAR DAQ configuration: 0—CLEAR_DAQ_REQ is reset. 1—CLEAR_DAQ_REQ is set.
4	Unused	—
5	Unused	—
6	DAQ_RUNNING	Data Transfer: 0—The data transfer is not running. 1—The data transfer is running.
7	RESUME	RESUME Mode: 0—The slave device is not in RESUME mode. 1—The slave device is in RESUME mode.

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:

Bit Number	Flag	Description
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	—

Bit Number	Flag	Description
6	Unused	—
7	Unused	—

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 `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

```
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` `ECURefNum` is the task reference which links to the selected ECU. This reference is originally returned from `mcECUSelectEx`.

`Mode` `Mode` describes the verification mode:

Value	Description
0	Request to start internal routine.
1	Send a Verification Value stored in `VerValue`.

`VerType` `VerType` specifies the Verification Type of the requested program verification. The Verification Type is a bit mask described below:

Verification Type	Description
0x0001	Calibration area(s) of the flash.
0x0002	Code area(s) of the flash.
0x0004	Complete flash content.
0x0008 ... 0x0080	Reserved.
0x0100 ... 0xFF00	User defined.

`VerValue` `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:

Bit	Description
0	The given page is used by the slave device application.
1	The slave device XCP driver will access the given page.
2	Unused.
3	Unused.
4	Unused.
5	Unused.
6	Unused.
7	The logical segment number is ignored. The command applies to all segments.

`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

```
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:

Bit	Description
0	Request to store calibration data in non-volatile memory.
1	Unused.
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.
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.
4	Unused.
5	Unused.
6	Unused.
7	Unused.

`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.

A

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 **C**ommand**R**eceive**O**bject (**CRO**), which is used to send commands and data from the host to the slave device; and the **D**ata**T**ransmission**O**bject (**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 **C**ommand**R**eturn**M**essages (**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:

0	1	2	3	4	5	6	7
CMD	CTR	Parameter and Data Field					

A DataTransmissionObject has a **P**acket**ID** (**PID**) as the first byte. This PID determines how the rest of the message is interpreted. CCP differentiates between three types of DTOs:

PID	Type
0x00—0xFD	Data Acquisition Message
0xFE	Event Message
0xFF	Command Return Message

Command Return Messages and Event Messages have the following structure:

0	1	2	3	4	5	6	7
PID	ERR	Parameter and Data Field					

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 **A**cquisition **M**essages (DAQ Messages or **DAM**s) have a PID in the first byte, and the rest of the message contains data. DAMs may be shorter than eight bytes:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
PID	Parameter and Data Field						

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 **O**bject**D**efinition**T**ables (**ODT**s). 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.

B

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 websites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

Symbol	Prefix	Value
m	milli	10^{-3}
k	kilo	10^{3}
M	mega	10^{6}

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.

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.

E

ECU	Electronic Control Unit—An electronic device with a central processing unit performing programmed functions with its peripheral circuitry.
ECU M&C Channel functions	The part of the ECU M&C Toolkit API that you use to read and write channels. 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&C API Read and Write functions provide access to Characteristic or Measurement channels. When a CAN message is received, ECU M&C Toolkit converts raw fields in the message into physical units, which you then obtain using the ECU M&C API Read function. When you call a ECU M&C API Write function, you provide floating-point values in physical units, which ECU M&C Toolkit converts into raw fields and transmits as a CAN message based on the CCP protocol.
ECU reference	Reference handle to a selected ECU.
ECU task	*See* ECU reference.
Event Channel	Specifies the generic signal source that effectively determines the data transmission timing.
Extended arbitration ID	A 29-bit arbitration ID. Frames that use extended IDs are often referred to as CAN 2.0 Part B (the specification which defines them).

M

Master ID	A 6-byte string identifying the CCP master device.
Measurement	*See* DAQ.
Measurement task	A collection of DAQ channels that you can read or write.
Memory Transfer Address	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.
MTA	*See* Memory Transfer Address.

O

ODT	Object Descriptor Table—A list of elements (variables) used for organization of data acquisition (DAQ).

P

PID	**P**acket**ID**—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

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X