Automotive Diagnostic Command Set Toolkit

2023-05-16

Contents

October 2020, 372140M-01

This help provides instructions for using the Automotive Diagnostic Command Set Toolkit and contains information about installation, configuration, and troubleshooting. It also includes the Automotive Diagnostic Command Set Toolkit LabVIEW and C API reference.

Introduction

Diagnostics involve remote execution of routines, or services, on ECUs. To execute a routine, you send a byte string as a request to an ECU, and the ECU usually answers with a response byte string. Several diagnostic protocols such as KWP2000 and UDS standardize the format of the services to be executed, but those standards leave a large amount of room for manufacturer-specific extensions. A newer trend is the emission-related legislated OnBoard Diagnostics (OBD), which is manufacturer independent and standardized in SAE J1979 and ISO 15031-5. This standard adds another set of services that follow the same scheme.

Because diagnostics were traditionally executed on serial communication links, the byte string length is not limited. For newer, CAN, LIN, or Ethernet-based diagnostics, this involves using a transport protocol that segments the arbitrarily long byte strings into pieces that can be transferred over the CAN or LIN bus, and reassembles them on the receiver side. Several transport protocols accomplish this task. The Automotive Diagnostic Command Set implements the ISO TP (standardized in ISO 15765-2) for CAN and LIN-based diagnostics, the manufacturer-specific VW TP 2.0 for CAN-based diagnostics, and the Diagnostics on IP (DoIP) transport protocol (standardized as ISO 13400) for Ethernet-based diagnostics.

Note The Automotive Diagnostic Command Set is designed for CAN, LIN, or Ethernet-based diagnostics only. Diagnostics on serial lines (K-line and L-line) or FlexRay are not in the scope of the Automotive Diagnostic Command Set.

The following topics discuss the KWP2000 and UDS protocols:

KWP2000 (Key Word Protocol 2000)

Transport Protocol

Diagnostic Services

Diagnostic Service Format

Connect/Disconnect

GetSeed/Unlock

KWP2000 (Key Word Protocol 2000)

The KWP2000 protocol has become a de facto standard in automotive diagnostic applications. It is standardized as ISO 14230-3. KWP2000 describes the implementation of various diagnostic services you can access through the protocol. You can run KWP2000 on several transport layers such as K-line (serial) or CAN.

Transport Protocol

As KWP2000 uses messages of variable byte lengths, a transport protocol is necessary on layers with only a well defined (short) message length, such as CAN. The transport protocol splits a long KWP2000 message into pieces that can be transferred over the network and reassembles those pieces to recover the original message.

KWP2000 runs on CAN on various transport protocols such as ISO TP (ISO 15765-2), TP 1.6, TP 2.0 (Volkswagen), SAE J1939-21, and Diagnostic Over IP (ISO 13400).

Note For KWP2000, the Automotive Diagnostic Command Set supports only the ISO TP (standardized in ISO 15765-2), manufacturer-

specific VW TP 2.0 transport protocols, and Diagnostic Over IP (ISO 13400).

Diagnostic Services

The diagnostic services available in KWP2000 are grouped in functional units and identified by a one-byte code (ServiceId). The standard does not define all codes; for some codes, the standard refers to other SAE or ISO standards, and some are reserved for manufacturer-specific extensions. The Automotive Diagnostic Command Set supports the following services:

- Diagnostic Management
- Data Transmission
- Stored Data Transmission (Diagnostic Trouble Codes)
- Input/Output Control
- Remote Activation of Routine

Note Upload/Download and Extended services are not part of the Automotive Diagnostic Command Set.

Diagnostic Service Format

Diagnostic services have a common message format. Each service defines a Request Message, Positive Response Message, and Negative Response Message.

The Request Message has the ServiceId as first byte, plus additional service-defined parameters. The Positive Response Message has an echo of the ServiceId with bit 6 set as first byte, plus the service-defined response parameters.

The Negative Response Message is usually a three-byte message: it has the Negative Response ServiceId as first byte, an echo of the original ServiceId as second byte, and a ResponseCode as third byte. The only exception to this format is the negative response to an EscapeCode service; here, the third byte is an echo of the user-defined service code, and the fourth byte is the ResponseCode. The KWP2000 standard partly defines the ResponseCodes, but there is room left for manufacturer-specific extensions. For some of the ResponseCodes, KWP2000 defines an error handling procedure. Because both positive and negative responses have an echo of

the requested service, you can always assign the responses to their corresponding request.

Connect/Disconnect

KWP2000 expects a diagnostic session to be started with StartDiagnosticSession and terminated with StopDiagnosticSession. However, StartDiagnosticSession has a DiagnosticMode parameter that determines the diagnostic session type. Depending on this type, the ECU may or may not support other diagnostic services, or operate in a restricted mode where not all ECU functions are available. The DiagnosticMode parameter values are manufacturer specific and not defined in the standard.

For a diagnostic session to remain active, it must execute the TesterPresent service periodically if no other service is executed. If the TesterPresent service is missing for a certain period of time, the diagnostic session is terminated, and the ECU returns to normal operation mode.

GetSeed/Unlock

A GetSeed/Unlock mechanism may protect some diagnostic services. However, the applicable services are left to the manufacturer and not defined by the standard.

You can execute the GetSeed/Unlock mechanism through the SecurityAccess service. This defines several levels of security, but the manufacturer assigns these levels to certain services.

Read/Write Memory

Use the Read/WriteMemoryByAddress services to upload/download data to certain memory addresses on an ECU. The address is a three-byte quantity in KWP2000 and a five-byte quantity (four-byte address and one-byte extension) in the calibration protocols.

The Upload/Download functional unit services are highly manufacturer specific and not well defined in the standard, so they are not a good way to provide a general upload/download mechanism.

Measurements

Use the ReadDataByLocal/CommonIdentifier services to access ECU data in a way similar to a DAQ list. A Local/CommonIdentifier describes a list of ECU quantities that are then transferred from the ECU to the tester. The transfer can be either single value or periodic, with a slow, medium, or fast transfer rate. The transfer rates are manufacturer specific; you can use the SetDataRates service to set them, but this setting is manufacturer specific.

Note The Automotive Diagnostic Command Set supports single-point measurements.

Diagnostic Trouble Codes

A major diagnostic feature is the readout of Diagnostic Trouble Codes (DTCs). KWP2000 defines several services that access DTCs based on their group or status.

Input/Output Control

KWP2000 defines services to modify internal or external ECU signals. One example is redirecting ECU sensor inputs to stimulated signals. The control parameters of these commands are manufacturer specific and not defined in the standard.

Remote Activation of a Routine

These services are similar to the ActionService and DiagService functions of CCP. You can invoke an ECU internal routine identified by a Local/CommonIdentifier or a memory address. Contrary to the CCP case, execution of this routine can be asynchronous; that is, there are separate Start, Stop, and RequestResult services.

The control parameters of these commands are manufacturer specific and not defined in the standard.

External References

For more information about the KWP2000 Standard, refer to the ISO 14230-3 standard.

UDS (Unified Diagnostic Services)

The UDS protocol has become a de facto standard in automotive diagnostic applications. It is standardized as ISO 14229. UDS describes the implementation of various diagnostic services you can access through the protocol.

As UDS uses messages of variable byte lengths, a transport protocol is necessary on layers with only a well defined (short) message length, such as CAN or LIN. The transport protocol splits a long UDS message into pieces that can be transferred over the network and reassembles those pieces to recover the original message.

UDS runs on CAN, LIN, and Ethernet on various transport protocols.

Note The Automotive Diagnostic Command Set supports only the ISO TP (standardized in ISO 15765-2), manufacturer-specific VW TP 2.0 transport protocols, and Diagnostic Over IP (ISO 13400).

Diagnostic Services

The diagnostic services available in UDS are grouped in functional units and identified by a one-byte code (ServiceId). Not all codes are defined in the standard; for some codes, the standard refers to other standards, and some are reserved for manufacturer-specific extensions. The Automotive Diagnostic Command Set supports the following services:

- Diagnostic Management
- Data Transmission
- Stored Data Transmission (Diagnostic Trouble Codes)
- Input/Output Control
- Remote Activation of Routine

For UDS on LIN, a slave node must support a set of ISO 14229-1 diagnostic services such as:

- Node identification (reading hardware and software version, hardware part number, and diagnostic version)

- Reading data parameters (reading ECU internal values such as oil temperature and vehicle speed)
- Writing parameter values if applicable

Note For more information about the LIN Diagnostic service implementations, refer to the **LIN Specification Package**, Revision 2.2, from the LIN Consortium.

Diagnostic Service Format

Diagnostic services have a common message format. Each service defines a Request Message, a Positive Response Message, and a Negative Response Message. The general format of the diagnostic services complies with the KWP2000 definition; most of the Service Ids also comply with KWP2000. The Request Message has the ServiceId as first byte, plus additional service-defined parameters. The Positive Response Message has an echo of the ServiceId with bit 6 set as first byte, plus the service-defined response parameters.

Note Some parameters to both the Request and Positive Response Messages are optional. Each service defines these parameters. Also, the standard does not define all parameters.

The Negative Response Message is usually a three-byte message: it has the Negative Response ServiceId (0x7F) as first byte, an echo of the original ServiceId as second byte, and a ResponseCode as third byte. The UDS standard partly defines the ResponseCodes, but there is room left for manufacturer-specific extensions. For some of the ResponseCodes, UDS defines an error handling procedure.

Because both positive and negative responses have an echo of the requested service, you always can assign the responses to their corresponding request.

External References

For more information about the UDS Standard, refer to the ISO 15765-3 standard.

OBD (On-Board Diagnostic)

On-Board Diagnostic (OBD) systems are present in most cars and light trucks on the road today. On-Board Diagnostics refer to the vehicle's self-diagnostic and reporting capability, which the vehicle owner or a repair technician can use to query status information for various vehicle subsystems.

The amount of diagnostic information available via OBD has increased since the introduction of on-board vehicle computers in the early 1980s. Modern OBD implementations use a CAN communication port to provide real-time data and a standardized series of diagnostic trouble codes (DTCs), which identify and remedy malfunctions within the vehicle. In the 1970s and early 1980s, manufacturers began using electronic means to control engine functions and diagnose engine problems. This was primarily to meet EPA emission standards. Through the years, on-board diagnostic systems have become more sophisticated. OBD-II, a standard introduced in the mid 1990s, provides almost complete engine control and also monitors parts of the chassis, body, and accessory devices, as well as the car's diagnostic control network. The newest standard was introduced in 2012 as WWH-OBD.

The On-Board Diagnostic (OBD) standard defines a minimum set of diagnostic information for passenger cars and light and medium-duty trucks, which must be exchanged with any off-board test equipment.

Installation and Configuration

The following topics discuss the installation and configuration of the ECU M&C Toolkit for Microsoft Windows.

- License Manager
- LabVIEW Real-Time (RT) Configuration

LabVIEW Real-Time (RT) Configuration

LabVIEW Real-Time (RT) combines easy-to-use LabVIEW programming with the power of real-time systems. When you use an NI PXI controller as a LabVIEW RT system, you can install a PXI CAN card and use the NI-CAN APIs to develop real-time applications. As with any NI software library for LabVIEW RT, you must install the Automotive Diagnostic Command Set software to the LabVIEW RT target using the Remote Systems branch in MAX. For more information, refer to the LabVIEW RT documentation.

After you install the PXI CAN cards and download the Automotive Diagnostic Command Set software to the LabVIEW RT system, you must verify the installation.

Application Development

The following topics explain how to develop an application using the Automotive Diagnostic Command Set API.

Choosing the Programming Language

LabVIEW

LabWindows™/CVI™

Visual C++ 6

Other Programming Languages

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

Choosing the Programming Language

The programming language you use for application development determines how to access the Automotive Diagnostic Command Set APIs.

LabVIEW

LabWindows/CVI

Visual C++ 6

Other Programming Languages

LabVIEW

Automotive Diagnostic Command Set functions and controls are in the LabVIEW palettes. In LabVIEW, the Automotive Diagnostic Command Set palette is in the Addons palette.

The Automotive Diagnostic Command Set API for LabVIEW section of this help describes each LabVIEW VI for the Automotive Diagnostic Command Set API.

To access the VI reference from within LabVIEW, press <Ctrl-H> to open the Help window, click the appropriate Automotive Diagnostic Command Set VI, and follow

the link. The Automotive Diagnostic Command Set software includes a full set of LabVIEW examples. 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 Automotive Diagnostic Command Set function panel is in **Libraries»Automotive Diagnostic Command Set**. As with other LabWindows/CVI function panels, the Automotive Diagnostic Command Set function panel provides help for each function and the ability to generate code. The Automotive Diagnostic Command Set API for C section of this help describes each Automotive Diagnostic Command Set API function. You can access the reference for each function directly from within the function panel. The Automotive Diagnostic Command Set API header file is `nidiagcs.h`. The Automotive Diagnostic Command Set API library is `nidiagcs.lib`. The toolkit software includes a full set of LabWindows/CVI examples. The examples are in the LabWindows/CVI `\samples\Automotive Diagnostic Command Set` directory. Each example includes a complete LabWindows/CVI project (`.prj` file). The example description is in comments at the top of the `.c` file.

Visual C++ 6

The Automotive Diagnostic Command Set 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 Automotive Diagnostic Command Set API, include the `nidiagcs.h` header file in the code, then link with the `nidiagcs.lib` library file. The 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, as follows:

```
#include "nidiagcs.h"
```

For C++ applications (files with a `.cpp` extension), define `__cplusplus` before including the header, as follows:

```
#define __cplusplus

#include "nidiagcs.h"
```

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

The Automotive Diagnostic Command Set API for C section of this help describes each function.

On Windows Vista (with Standard User Account), the typical path to the C examples folder is `\Users\Public\Documents\National Instruments\Automotive Diagnostic Command Set\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\Automotive Diagnostic Command Set\Examples\MS Visual C`.

Each example is in a separate folder. The example description 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>** `GetDTCs.c` **<LibDir>**`\nidiagcs.lib`

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

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

Other Programming Languages

The Automotive Diagnostic Command Set software does not provide formal support for programming languages other than those described in the preceding sections. If the programming language includes a mechanism to call a Dynamic Link Library (DLL), you can create code to call Automotive Diagnostic Command Set functions. All functions for the Automotive Diagnostic Command Set API are in `nidiagcs.dll`. If the programming language supports the Microsoft Win32 APIs, you can load pointers to Automotive Diagnostic Command Set functions in the application. The following section describes how to use 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 shows how to call Win32 `LoadLibrary` to load the Automotive Diagnostic Command Set API DLL:

```
#include <windows.h>
#include "nidiagcs.h"
HINSTANCE NiDiagCSLib = NULL;
NiMcLib = LoadLibrary("nidiagcs.dll");
```

Next, the application must call the Win32 `GetProcAddress` function to obtain a pointer to each Automotive Diagnostic Command Set function the application uses. For each function, you must declare a pointer variable using the prototype of the function. For the Automotive Diagnostic Command Set function prototypes, refer to the Automotive Diagnostic Command Set API for C section of this help. Before exiting the application, you must unload the Automotive Diagnostic Command Set DLL as follows:

```
FreeLibrary (NiDiagCSLib);
```

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 Automotive Diagnostic Command Set does not include FPGA bitfiles for any FPGA target. Refer to the LabVIEW FPGA Module documentation for more information about creating FPGA VIs and bitfiles for an FPGA target.

The default FPGA VI is sufficient for a basic Automotive Diagnostic Command Set application. However, in some situations you may need to modify the existing FPGA code to 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 ADCS 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 ADCS 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).

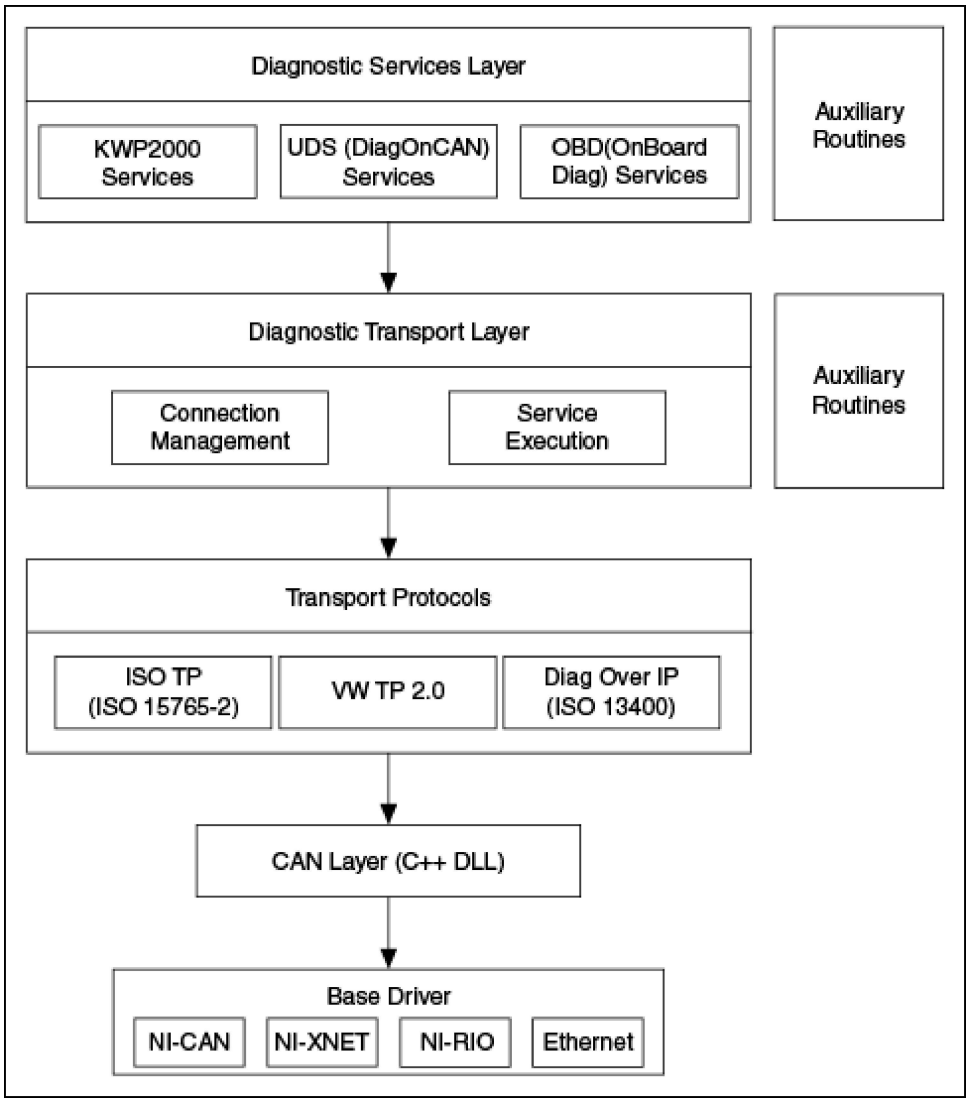
After recompiling your FPGA VI, copy the bitfile to the root directory of your CompactRIO controller and specify the bitfile in the interface name. Or copy the file to any location on the CompactRIO controller and specify an absolute path or path relative to the root for the bitfile.

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 ADCS. Refer to the **Getting Started with CompactRIO** section in the **NI-XNET Hardware and Software Help** 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.

Using the Automotive Diagnostic Command Set

The following topics explain how to use the Automotive Diagnostic Command Set:

Structure of the Automotive Diagnostic Command Set



The Automotive Diagnostic Command Set is structured into three layers of functionality:

- The top layer implements three sets of diagnostic services for the diagnostic protocols KWP2000, UDS (DiagOnCAN), and OBD (On-Board Diagnostics).
- The second layer implements general routines involving opening and closing diagnostic communication connections, connecting and disconnecting to/from an ECU, and executing a diagnostic service on byte level. The latter routine is the one the top layer uses heavily.

- The third layer implements the transport protocols needed for diagnostic communication to an ECU. The second layer uses these routines to communicate to an ECU.

All three top layers are fully implemented in LabVIEW.

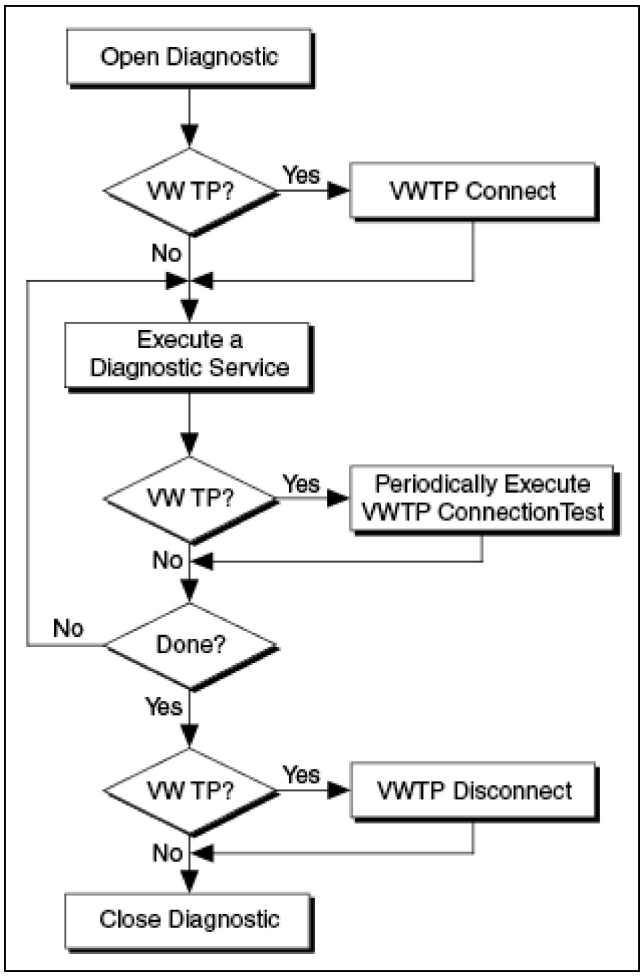
The transport protocols then execute CAN/LIN Read/Write operations through a specialized DLL for streamlining the CAN/LIN data flow, especially in higher busload situations.

Automotive Diagnostic Command Set API Structure

The top two layer routines are available as API functions. Each diagnostic service for KWP2000, UDS, and OBD is available as one routine. Also available on the top level are auxiliary routines for converting scaled physical data values to and from their binary representations used in the diagnostic services.

On the second layer are more general routines for opening and closing diagnostic communication channels and executing a diagnostic service. Auxiliary routines create the diagnostic identifiers from the logical ECU address.

General Programming Model



First, you must open a diagnostic communication link. This involves initializing the CAN/LIN port and defining communication parameters such as the baud rate. For CAN-based diagnostics, the CAN identifiers on which the diagnostic communication takes place must be defined also. No actual communication to the ECU takes place at this stage.

For the VW TP 2.0, you then must establish a communication channel to the ECU using the VWTP Connect routine. The communication channel properties are negotiated between the host and ECU.

After these steps, the diagnostic communication is established, and you can execute diagnostic services of your choice. Note that for the VW TP 2.0, you must execute the

VWTP ConnectionTest routine periodically (once per second) to keep the communication channel open.

When you finish your diagnostic services, you must close the diagnostic communication link. This finally closes the CAN or LIN port. For the VW TP 2.0, you should disconnect the communication channel established before closing.

Available Diagnostic Services

The standards on automotive diagnostic define many different services for many purposes. Unfortunately, most services leave a large amount of room for manufacturer-specific variants and extensions. NI has implemented the most used variants while trying not to overload them with optional parameters.

However, all services are implemented in LabVIEW and open to the user. If you are missing a service or variant of an existing service, you can easily add or modify it on your own.

In the C API, you can also implement your own diagnostic services using the ndDiagnosticService routine. However, the templates from the existing services are not available.

Tweaking the Transport Protocol

A set of global constants controls transport protocol behavior. These constants default to maximum performance. To check the properties of an implementation of a transport protocol in an ECU, for example, you may want to change the constants to nonstandard values using the Get/Set Property routines.

Using an XNET IP Stack

Each ADCS DoIP session can use either the native operating system IP stack or an XNET IP Stack, and parallel ADCS sessions can use any combination of IP stacks and virtual interfaces.

By default, DoIP sessions use the operating system IP stack. To use an XNET IP Stack, set the **XNET IP stack name** parameter of Open Diagnostic On IP.vi (or the

xnetStackName parameter of ndOpenDiagnosticOnIPStack) to a valid name of an XNET IP Stack.

ADCS obtains a reference to the selected XNET IP Stack. After communication is complete, the function Close Diagnostic.vi or ndCloseDiagnostic must be used to free that reference in all cases.

An XNET IP Stack allows the configuration of an NI-XNET network interface independent from other network interfaces in the system and provides advanced features such as virtual interfaces and VLANs. The XNET IP Stack must be created beforehand using the NI-XNET API. Refer to **NI-XNET Hardware and Software Help** and NI-XNET examples for information about configuring, creating, and clearing an XNET IP Stack.