WORLDWIDE ENGINEERING STANDARDS

General Specification
Electrical Function

GMW3110

General Motors Local Area Network Enhanced Diagnostic Test Mode Specification

1 Introduction

Note: Nothing in this standard supersedes applicable laws and regulations.

Note: In the event of conflict between the English and domestic language, the English language shall take precedence.

General Motors in-vehicle Local Area Network (GMLAN) is a family of serial communication buses (subnets) which enable Electronic Control Units (ECU) or nodes)) to communicate with each other, or with a diagnostic tester.

GMLAN supports three bus types; dual wire high speed controller area network (HS-CAN) buses, dual wire mid speed (MS-CAN) buses, and single wire low speed (LS-CAN) buses.

High speed buses (500 kilobits per second (kbps)) are typically used for sharing real time data such as driver commanded torque, actual engine torque, steering angle, etc. Examples of High speed buses are, but not limited to, the Primary HS bus, the Chassis Expansion bus, and the Powertrain Expansion bus.

Mid speed buses (approximately 95.2 kbps or 125 kbps) are typically used for infotainment applications (display, navigation, etc.) where the system response time demands that a large amount of data be transmitted in a relatively short amount of time, such as updating a graphics display. An example of a mid speed bus is the Primary MS bus.

Low speed buses (33.33 kbps) are typically used for operator-controlled functions where the system response time requirements are of the order of 100 to 200 ms. These buses also support high speed operation at 83.33 kbps used only during ECU reprogramming. An example of a Low speed bus is the Primary LS bus.

The decision to use a particular bus in a given vehicle depends upon how the feature/functions are partitioned among the different ECUs in that vehicle. There may be more than one network of a particular bus type in a given vehicle.

GMLAN buses use the Controller Area Network (CAN) communications protocol. Data is packaged into CAN messages, which are segmented into CAN frames. Each CAN frame includes header data (also known as the CAN Identifier, or CANId), and a maximum of eight (8) data bytes. A message may be comprised of a single frame, or multiple frames depending on the number of data bytes which defines the complete message. Data link arbitration occurs only over the header, or CANId, portion of a frame.

Note: See reference documents listed in Section 2 for a detailed definition of the CAN Link.

The GMLAN Enhanced Diagnostic Test Mode Specification is a GMW (GM Worldwide) document (also called GM Corporate) which defines diagnostic requirements for vehicle nodes and diagnostic testers.

1.1 Scope. This specification establishes the Enhanced Diagnostics strategy for the GM In-Vehicle Local Area Network Subsystem, GMLAN. This strategy is required to be implemented by any node diagnosed on any of the GMLAN subnets. The strategy shall be tolerated by nodes connected to GMLAN subnets which are diagnosed via another means.

1.2 Mission/Theme. The GMLAN diagnostic strategy shall provide a reliable, effective, and flexible means to diagnose ECUs and systems on vehicles equipped with GMLAN buses. The diagnostic services provided shall operate the same on all ECUs independent of subnet type, except where the diagnostic service affects the baud rate of the subnet.

The GMLAN Enhanced Diagnostic Test Mode Specification has been targeted to the following objectives:

- Functional Based Diagnostic Approach to meet distributed system functionality over multiple ECUs and in-vehicle subnets.
- Meet diagnostic requirements of in-vehicle Virtual Networks and Virtual Devices.
- Harmonization of GM J2190 and GM KWP2000 services as one global GM Corporate Standard.
- Reduction of total number of diagnostic services (test modes) and utilization of minimum amount of CAN Identifiers for Diagnostics.
- Optimized performance for testing and programming (e.g., parallel testing of ECUs on multiple subnets).
- Clear split by concept between dynamic real-time data retrieval and multiple frame static data messages (non time critical data).

1.3 Classification. Not applicable.

2 References

Note: Only the latest approved standards are applicable unless otherwise specified.

2.1 External Standards/Specifications.

ISO 7498	SAE J1979
ISO/DIS 11898	SAE 1930

ISO 15031-5

ISO 15765-2

ISO 15765-4

ISO/TR 8509

SAE J2012

SAE J2178

SAE J2186

2.2 GM Standards/Specifications.

GM1737

GM J1962

GM J2190

GMW3089

GMW3104

GMW3107

GMW3122

GMW4710

GMW15862

TL.07.0054.Rxx

Note: Rxx in the specification number indicates the revision level of this document.

- GM Bill Of Process - General Assembly Programming and Test.
- GM Body Chassis Network Communication Common Diagnostic Trouble Code Specification.
- Service Programming System (SPS) Interpreter Programmers Reference Manual.
- Vehicle Theft Deterrent Subsystem Technical Specification (SSTS).

2.3 Additional References.

2.3.1 GMLAN Specific Publications. There are two types of GMLAN publications: General Specifications and Device Specifications.

2.3.1.1 General Specification. The General Specifications establish the fundamental concepts needed by a GMLAN ECU (e.g., Communication Strategies, Bus Wiring, Physical Layer requirements, etc.) For a given ECU, it is possible that not everything in the General Specifications will apply. See Figure 1.

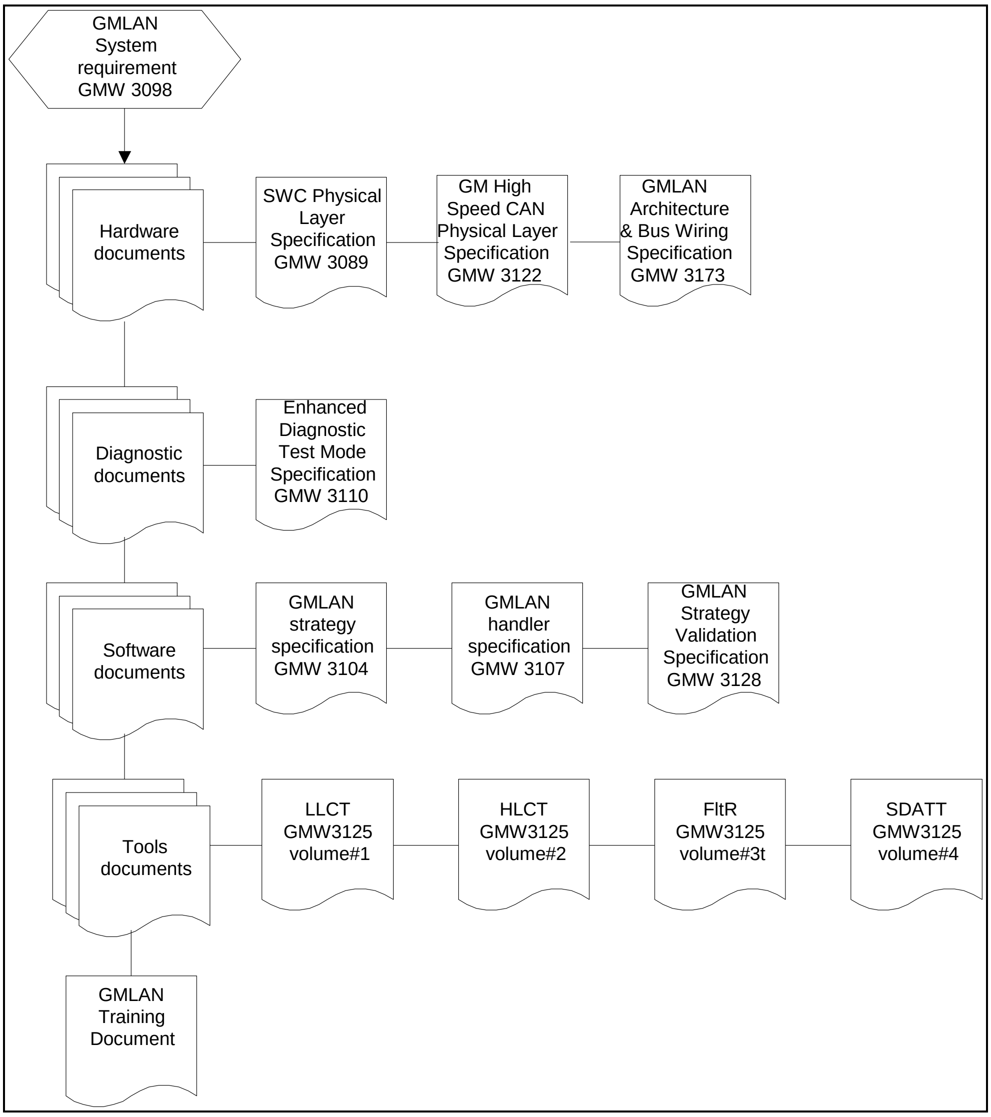

GMLAN System requirement GMW 3098
Hardware documents
SWC Physical Layer Specification GMW 3089
GM High Speed CAN Physical Layer Specification GMW 3122
GMLAN Architecture & Bus Wiring Specification GMW 3173
Diagnostic documents
Enhanced Diagnostic Test Mode Specification GMW 3110
Software documents
GMLAN strategy specification GMW 3104
GMLAN handler specification GMW 3107
GMLAN Strategy Validation Specification GMW 3128
Tools documents
LLCT GMW3125 volume#1
HLCT GMW3125 volume#2
FltR GMW3125 volume#3t
SDATT GMW3125 volume#4
GMLAN Training Document

Figure 1: GMLAN General Specifications

2.3.1.2 Device Specification. The Device Specification is ECU specific. It contains:

- References to the optional parts of General Specifications which are applicable to the ECU.
- ECU specific information (e.g., Diagnostic Services required for implementation).

2.3.2 Other Publications.

OSEK/COM/VDX	Communication Specification Version 2.2.2
Bosch	CAN Specification 2.0a/2.0b

3 Diagnostic Service Documentation Structure And Convention

Each diagnostic service is documented and defined in this specification within a specific subsection of Section 8 - Diagnostic Services (Test Modes) Definiton. The diagnostic service definitions are structured as follows:

1. A brief summary of the diagnostic service intent.
2. A Service Description section which provides text describing the general functionality of the diagnostic service and how it should be used (and implemented).

3. A Request Message Definition section which includes:

- Tables to define the structure of the request message.
- A sub-function parameter definition section with text and table descriptions of any sub-level functions within the diagnostic service.
- A request message data parameter definition section with text and table descriptions of any additional data bytes used in the request message.

4. A Response Message Definition section which includes:

- Tables to define the structure of the positive response message.
- A response message data parameter definition section with text and table descriptions of any additional data bytes used in the response message.
- A section which defines specific negative response codes supported by the test mode service.

5. A **Message Flow Example** section using tables to represent the flow of messages between the tester and target node.
6. A **Node Interface Function** section which provides:

- An interface data dictionary specific to the particular diagnostic service.
- Interface pseudo code specific to the test mode.
- Suggested verification procedures to validate proper implementation of the diagnostic service.
- A section describing any special case implications specific to test tools and their implementation of the diagnostic services.

The following provides more detailed descriptions and examples of specific diagnostic service documentation sections and or table examples.

3.1 Request Message Sub-Function Parameter ($Level) Definition. A sub-function parameter is used when a diagnostic service supports multiple levels of functionality. The sub-function parameter tells the diagnostic application which functionality is being requested. A sub-function parameter is used in a tester to ECU request message and when used, shall always occupy the first byte after the Service Identifier.

The $Level designation is included in the heading to indicate that the pseudo code shall reference the sub-function parameter using $Level as a variable.

3.2 Request Message Data Parameter Definition. A request message data parameter is used to indicate to the ECU diagnostic application which specific information is being requested (e.g., specific ECU input or output status variables, calculated variables, etc.) A request message data parameter can affect the data contained in a response message but it does not affect the functionality of the diagnostic service used to retrieve it.

Example: If a diagnostic service is used to provide ECU input statuses, output statuses, etc., then a data parameter would be passed in with the diagnostic request message to identify which inputs, outputs, etc., that the tester is requesting. A sub-function parameter would be contained in the request message for this service to tell the diagnostic application if the data is to be sent as a one time response, sent periodically, or to stop sending previously requested data.

3.3 Positive Response Message Data Parameter Definition. A data parameter in a positive message is used to provide the information to the tester that was requested via a diagnostic request message.

3.4 (Test Mode) Service Identifier (SID) Overview. Each service (test mode) shall be identified by its service Identifier (SID).

The SID must be included in each diagnostic request message.

Table 1 indicates the different ranges of service Identifier values, which are defined in ISO 15031-5/SAE J1979, Keyword Protocol 2000/GM J2190, by the vehicle manufacturer, or by system supplier.

Table 1 provides an overview about Service Identifier (SID) used by different protocols in relation to GMLAN SID assignments.

Table 1: Service (Test Mode) Identifier Overview

Service Identifier Hex Value	Service Type (bit 6)	Where Defined
00 thru 0F	Request	reserved by ISO 15031-5/SAE J1979
10 thru 1F	Request (bit 6 = 0)	reserved by GM J2190, KWP 2000 Part 2, 3 and GMLAN Enhanced Diagnostic Services
20 thru 2F		
30 thru 3E		
3F	Request (bit 6 = 0)	used by GM J2190
40 thru 4F	Positive Response	reserved by ISO 15031-5/SAE J1979
50 thru 5F	Positive Response to Services ($10 thru $3E) (bit 6 = 1)	reserved by GM J2190, KWP 2000 Part 2, 3 and GMLAN Enhanced Diagnostic Services
60 thru 6F		
70 thru 7E		
7F	Negative Response	used by GM J2190, KWP 2000, and GMLAN Enhanced Diagnostic Services
80 thru BF	Request (bit 6 = 0)	GMLAN Enhanced Diagnostic Services
C0 thru FF	Positive Response (bit 6 = 1)	GMLAN Enhanced Diagnostic Services

3.5 Request and Positive Response Message Table Structure. The description of each service includes a table which shows the appropriate structure for the request and positive response messages. These tables define which data byte(s) are applicable for the particular diagnostic service. Tables 2 and 3 are general examples of the format of these tables.

Table 2: Service Request Message Table

Data Byte	Parameter Name	Cvt Note 1	Hex Value	Mnemonic
#1	< Diagnostic Service/Test Mode Request > Service Identifier	M	zz	SIDRQ
#2	sub-function $Level (where applicable)	C	xx	LEV_
#2 or #3	data (where applicable) = [data#1	C	xx	(abv)
. . .	. . .	. . .	xx	. . .
#n	data#m]	C	xx	(abv)

Note 1: **M** = Mandatory, shall always be present.
U = User optional; the parameter shall or shall not be supplied.
C = Conditional; the presence of the parameter depends upon other parameters within the service or if the ECU requires the use of the parameter in order to fulfill other diagnostic requirements (e.g., the use of negative response code $78 in order to meet P2 timing requirements).

Table 3: Positive Response Message Table

Data Byte	Parameter Name	Cvt Note 1	Hex Value	Mnemonic
#1	< Diagnostic Service Positive Response> Service Identifier/Message Number	M	zz	SIDPR/Msg#
#2	sub-function $Level echo (only where applicable)	C	xx	LEV_
#2 or #3	data (where applicable) = [data#1	C	xx	(abv)
. . .	. . .	. . .	xx	. . .
#n	data#m]	C	xx	(abv)

Note 1: **M** = Mandatory, shall always be present.
U = User optional; the parameter shall or shall not be supplied.
C = Conditional; the presence of the parameter depends upon other parameters within the service or if the ECU requires the use of the parameter in order to fulfill other diagnostic requirements (e.g., the use of negative response code $78 in order to meet P2 timing requirements).

3.6 Node Interface Function Symbol, Pseudo Code, and Data Dictionary Definition.

3.6.1 Symbol Definition. Table 4 contains a list of all of the symbols used in the pseudo code found in the ECU Interface Section of each diagnostic service.

Table 4: Symbol Definition

Symbol	Definition
YES, TRUE, ENABLE, ACTIVE	Symbols used to increase readability. YES, TRUE, ENABLE, and ACTIVE can be replaced with a value of 1.
NO, FALSE, DISABLE, INACTIVE	Symbols used to increase readability. NO, FALSE, DISABLE, and INACTIVE can be replaced with a value of 0.
&	Bit-wise logical AND
AND	Logical AND
|	Bit-wise logical OR
OR	Logical OR
NOT	Logical negation
MOD	Modula Operation (e.g., z = x MOD y results in z set to the remainder of x/y)
←	Assignment
=	Equal to
!=	Not equal to
>	Greater than
<	Less than
≥	Greater than or equal to
≤	Less than or equal to
+, -, *, /	Addition, Subtraction, Multiplication, and Division
>>	Logical bit-wise shift right

Symbol	Definition
<<	Logical bit-wise shift left
$	Prefix for Hexadecimal Number
b	Suffix for Binary Number
/*	Prefix for a Comment String
*/	Suffix for comment string of more than one line

3.6.2 Pseudo Code Definition. The ECU Interface functions are presented in pseudo code format. This is to reduce the possibility of multiple interpretations of a diagnostic service (test mode). The indentation practices in the example functions below shall be followed to enhance readability.

3.6.2.1 Pseudo Code Syntax.

3.6.2.1.1 Decision Statement. Decision statements use the keywords IF, THEN, ELSE, ELSE IF and ENDIF to control the flow of execution. Included in the parenthesis is a relational expression which can be evaluated as either TRUE or FALSE. The flow of execution through decision statements is based on the results of the evaluation of the relational expression(s) contained within them. An example is provided below:

```
IF (relational expression #1) THEN
body when expression #1 is TRUE
ELSE IF (relational expression #2) THEN
body when expression #2 is TRUE
ELSE
body when expressions #1 and #2 are FALSE
ENDIF
```

Note: Every IF statement shall have a corresponding ELSE, ELSE IF, or ENDIF statement. Below is an example of pseudo code which uses two variables x and y.

```
IF (x > y) THEN x = x+2   /* body when x > y */
ELSE IF (x = y) THEN
    IF (x!= 0)            /* body when x = y */
    x = y/x               /* body when x = y and also body when x!= 0 */
    ENDIF                 /* body when x = y */
ELSE
    x = x*y               /* body when x < y */
ENDIF
```

3.6.2.1.2 Select Statement. Select statements are useful when a different body of code is to execute based on a series of relational expressions. The select statement uses the keywords SELECT FIRST and ENDSELECT to mark the beginning and ending of the select statement. When this statement is encountered, the relational expression in parenthesis next to the first WHEN keyword is evaluated. If this relational expression evaluates TRUE, then the body after the first WHEN keyword is executed. The body portion is indented one level and ends at the last line before the next WHEN keyword. If the first relational expression evaluates FALSE, the relational expression after the next WHEN keyword is evaluated. If this relational expression evaluates TRUE, then the body associated with that WHEN keyword is executed. If none of the relational expressions following WHEN keywords evaluate to TRUE, then the body following the OTHERWISE keyword is executed. After executing the body section of code following any WHEN or OTHERWISE keyword, the select statement is exited and the code begins executing after the ENDSELECT keyword. An example of this type of statement is included below:

```
SELECT FIRST
WHEN (first relational expression)
   body when first expression is TRUE
WHEN (second relational expression)
```

```
    body when second expression is TRUE

OTHERWISE
    body when all expressions are FALSE
ENDSELECT
```

3.6.2.1.3 Loop Statement. Loop statements are useful when it is desirable to execute a body of code in a repetitive fashion until an exit criteria is established. This type of statement is typically used when performing operations on arrays of data or data structures as it provides an easy mechanism to index through the array. Loop statements use the keywords FOR and ENDFOR to mark the beginning and ending points of the loop statement.

Included below is an example of a loop statement:

```
FOR (target value1 TO value2 BY value3)
    body to be performed for each value
ENDFOR
```

The first time the code encounters the keyword FOR, the variable **target** is assigned a value defined by **value1**. On the right of the keyword TO is a variable called **value2** which is compared with the variable **target** as a relational expression to determine if it is ok to exit the repetitive task. If the relational expression evaluates TRUE, then the body is executed. After executing the body of the loop, the variable **target** is incremented by the value of the variable after the keyword BY (in this case **value3**) and the relational expression is re-evaluated. The body will continue to be executed as long as the relational expression evaluates to TRUE. Once the relational expression evaluates FALSE, the loop is exited and control continues with the next statement after the ENDFOR.

Note: The convention used throughout this specification for handling the case where value1 equals value2 is that the expression is considered TRUE and the body code shall be executed.

3.6.2.1.4 WHILE Statement. While statements are used to execute a body of code as long as a relational expression evaluates TRUE. The statement uses the keywords WHILE and ENDWHILE to mark the beginning and ending of this type of statement. When the pseudo code hits a line with a WHILE keyword, the relational expression in parenthesis after the WHILE is evaluated. If the relational expression evaluates TRUE, the body is executed. The body is indented one level and ends with the last statement above the ENDWHILE keyword.

```
WHILE (relational expression)
    body when expression is TRUE
ENDWHILE
```

3.6.2.1.5 REPEAT Statement. Repeat statements are very similar to while statements except that the relational expression evaluated to determine when to stop executing the body code is not checked until after the body is executed. The body code will always be executed at least one time. The keywords REPEAT and UNTIL mark the beginning and ending of this type of statement. An example of this statement is included below:

```
REPEAT
    body when expression is TRUE (performed at least once)
UNTIL (relational expression)
```

3.6.2.1.6 CALL Statement. A CALL statement is used to invoke a procedure/subroutine. The procedure/subroutine may be defined in the description of another diagnostic service within this specification, or referenced from another specification (e.g., GM LAN Handler Specification). Upon completion of the procedure/subroutine, control is returned to the point immediately after the CALL statement. An example is included below:

```
IF (relational expression) THEN
    CALL function( )
ENDIF
```

3.6.3 Common/Global Pseudo Code Data Dictionary. Following the pseudo code is a data dictionary that defines the variables and flags used in the pseudo code. Several diagnostic services share common variables or flags with other diagnostic services. The table below contains the list of the common or global pseudo code variables and flags used in the Node Interface Data Dictionary section of each diagnostic service. The common/global variables will be included in the data dictionary of each diagnostic service but reference this section of the document for further details. See Table 5.

Table 5: Common/Global Pseudo Code Data Dictionary Global Variables

Variable/Meaning	**Power Up - S/W Reset Initial Value**	**Values**
message_data_length This variable represents the length of the message received as determined by the network/transport layer PCI.DL_High nibble and PCI.DL_Low byte (if the message is more than one frame). This correlates to the Service Data Unit (SDU) <Length> embedded in the N_USData.Ind service primitive as defined in ISO 15765-2 and OSEK/COM.	N/A	1 to 4095
message_address_type This variable represents the type of addressing used in the request message as determined by the network layer.	N/A	FUNCTIONAL/PHYSICAL
Dev_Cntrl_Active This flag used to indicate whether or not device control is currently active.	NO	YES/NO
request_DeviceControl_exit This flag is used in the DeviceControl logic to cancel active device control functions if a TesterPresent ($3E) timeout occurs or if a ReturnToNormalMode ($20) request is received.	NO	YES/NO
Diag_Services_Disable_DTCs This flag is used by the device to determine if any diagnostic service has been enabled which requires that Diagnostic Trouble Codes (DTC) be inhibited.	FALSE	TRUE/FALSE
DTCs_Enabled_In_Device_Control This flag is used to determine whether or not the service $AE (DeviceControl) will request DTCs to disable when device control is active. When this flag is equal to NO, activating service $AE would inhibit the setting of DTCs. Service $10 (InitiateDiagnosticOperation) can be used to change the state of this flag if it is requested prior to activating device control.	NO	YES/NO
Security_Access_Unlocked Flag set in service $27 (SecurityAccess) which indicates if the device is unlocked. If an ECU does not support the $27 Security service, then this flag shall default to TRUE for the purposes of the pseudo code for the other modes.	FALSE	TRUE/FALSE
manufacturers_enable_counter This is a one byte variable used when determining the security status of a node which implements the SecurityAccess ($27) service. Refer to 9.3.2.6.2 for a description of this variable.	N/A	$00 thru $FF

Variable/Meaning	Power Up - S/W Reset Initial Value	Values
vulnerability_flag The vulnerability_flag is a one byte variable in the Node's calibration area which is used when determining the security status of a node which implements the SecurityAccess ($27) service. Refer to 9.3.2.6.1 for a description of this variable.	N/A	$00 thru $FF
TesterPresent_Timer_State This status flag is set by any service which requires the tool to send TesterPresent ($3E) messages to remain active. While TesterPresent_Timer_State = ACTIVE, the diagnostic application checks the TesterPresent_Timer. Mode $3E messages are used to reset this timer. If a Mode $3E message is not received every $P3_C$ ms, the timer expires and causes the device to exit Diagnostic Mode.	INACTIVE	ACTIVE/INACTIVE
normal_message_transmission_status This is a flag that is set in the $28 service when normal communications have been shut off and DTCs are disabled.	ENABLED	ENABLED/DISABLED
TesterPresent_Timer TesterPresent ($3E) service request messages are used to reset this timer. When active, if a TesterPresent ($3E) request message is not received within the $P3_C$ timing window, the timer expires and shall cause the node to execute the equivalent of a ReturnToNormalMode ($20) service.	0	$00 thru $P3_{Cmax}$
DTC_send_on_change_flag This flag is used to indicate that the send-on-change reporting of DTC count information is currently active. This flag is activated as soon as the tester requests service $A9 message $82 with a non-zero send-on-change status mask. It is then used by the steady-state DTC count reporting function to enable updated count information to be calculated and reported back to the tester (if a count change is detected).	$00	$01 = send on change active $00 = send on change inactive
programming_mode_active This flag is set in service $A5 and is used to keep track of whether or not a programming event has been enabled.	NO	YES/NO
programming_mode_entry_ok This is a flag which is set to YES once the tool has sent a request to verify that all devices are capable of entering a programming event at this time. This flag must be set to YES to enter programming mode.	NO	YES/NO
high_speed_mode_entry_ok This is a flag which is set to YES once the tool has sent a request to verify that all devices are capable of entering a high speed programming event at this time. This flag must be set to YES to enter high speed mode. This flag is only valid for devices on the low speed link.	NO	YES/NO
high_speed_mode_active This is a flag used to keep track of whether or not high speed mode of the Single Wire CAN (SWCAN) has been enabled.	NO	YES/NO

Variable/Meaning	Power Up - S/W Reset Initial Value	Values
TransferData_Allowed This is a flag set in service $34 which is used by the TransferData ($36) service to determine if the correct sequence has occurred to begin transferring data.	NO	YES/NO
diagnostic_responses_enabled This is a flag which is set to YES if the ECU knows its permanent diagnostic CANIds or after a sequence of diagnostic messages which allows an SPS_TYPE_C ECU to enable responses to diagnostic requests. See mode $A2 and programming section of this specification for more information.	See Mode $A2 pseudo code	YES/NO
DeviceContol_Security_Level This is a flag used by the device control logic in determining the criteria for device control limits exceeded. If security is accessed, then the assembly plant device control restrictions are used. Otherwise, the restrictions imposed on service tools are used. Assembly plant device control restrictions are not required in all modules. If an ECU does not utilize assembly plant restrictions, then this variable defaults to SERVICE for the purposes of the pseudo code.	See Mode $27 pseudo code	ASSEMBLY_ PLT/SERVICE
Security_Access_Allowed Indication that a valid security code as defined in the Vehicle Theft Deterrent SSTS has been received.	FALSE	TRUE/FALSE

4 Diagnostic Strategy, Service Overview, and Implementation Rules and Requirements

This section provides implementation rules which shall be followed by the Subsystem Leadership Teams (SSLT) of each platform organization. This section defines diagnostic message strategy, message identification, message addressing, and fundamental byte assignments for message frame types.

4.1 Diagnostic Message Strategy - USDT and UUDT Messages. The Enhanced Diagnostic Test Mode Services utilize both Unacknowledged Segmented Data Transfer (USDT) and Unacknowledged Unsegmented Data Transfer (UUDT) messaging formats.

The USDT format refers to those messages which may be segmented into multiple CAN transmission frames, depending on the number of data bytes transmitted.

UUDT format refers to those messages which shall always be a single frame response.

All diagnostic request messages and negative response messages use the USDT format, whereas positive response messages may be either USDT or UUDT format. Paragraph 4.2 contains a table which specifies the format of the positive response messages for each diagnostic service.

The use of UUDT and USDT CAN Identifiers allows for interleaving of single frame responses (such as periodic data transmissions) with a multi-frame response (usually a one time data transmission) from the same node.

4.2 Diagnostic Service Table Overview. Table 6 gives a fundamental overview of the diagnostic services available and when each service is required to be implemented by an ECU. The table also indicates which message format (USDT or UUDT) shall be supported for the positive response message(s) for each service.

The **Diagnostics** column indicates which diagnostic services apply for all ECUs (programmable or non-programmable) in order to support vehicle assembly plant diagnostics, field service diagnostics, or ECU programming (even if a particular node is not programmable). For programmable ECUs, this column is applied when the ECU is in a fully programmed state. An **M** in this column indicates that the service is mandatory and shall be implemented by all ECUs as previously defined. A **C** in this column indicates that the need for a given service (including specific applicable sub-functions) is to be agreed upon by the Design Release Engineer (DRE), the appropriate service representative(s), and the appropriate manufacturing representative(s). A **U** in this column indicates that the DRE can decide whether or not to implement the service in an ECU. A **N/A** in this column indicates that this service is only required for programming and the need to support the service shall be based upon the contents of the SPS column.

The **Programming (SPS) and Boot Memory** column applies to ECUs which are programmable via the SPS system. An **M** in this column means that all ECUs which are programmable via the SPS system shall implement the service and that the service shall be available even if the ECU only contains boot software. A **C** in this column indicates that a service may be required for programming based on how the ECU supplier has developed their utility file. If the diagnostic service is used in the utility file, then it shall be supported in boot software. An **N/A** in this column indicates that the service is not applicable (or not required) in order to program the ECU.

Note: Reference the ECU Programming section of this specification for more information on boot software.

Note: Reference the ECU Programming section of this specification for more information about utility files.

Note: All diagnostic services listed below shall be supported by the node regardless of addressing method used in the request message, as long as the address is valid for that particular node. Refer to 4.5 for more information on message addressing. For tester limitations on request methods, refer to the tester implication section of each specific diagnostic service.

Table 6: Required Diagnostic Services Overview

		Supported in:			
Diagnostic Service Name	**REQ SID**	**Diagnostics**	**Programming**	**Boot**	**CANId**
(Test Mode)	**(hex)**		**(SPS)**	**Memory**	**Type**
ClearDiagnosticInformation	04	M_1	N/A		USDT
InitiateDiagnosticOperation	10	C_1	C_1		USDT
ReadFailureRecordData	12	C	N/A		USDT
ReadDataByIdentifier	1A	M	M_2		USDT
ReturnToNormalOperation	20	M	M		USDT
ReadDataByParameterIdentifier	22	U	N/A		USDT
ReadMemoryByAddress	23	U	N/A		USDT
SecurityAccess	27	C_2	M_3		USDT
DisableNormalCommunication	28	M	M		USDT
DynamicallyDefineMessage	2C	C	N/A		USDT
DefinePIDByAddress	2D	U	N/A		USDT
RequestDownload	34	N/A	M		USDT
TransferData	36	N/A	M		USDT
WriteDataByIdentifier	3B	C	M_4		USDT
TesterPresent	3E	M	M		USDT
ReportProgrammingState	A2	N/A	M		USDT
ProgrammingMode	A5	M	M		USDT

		Supported in:			
Diagnostic Service Name	**REQ SID**	**Diagnostics**	**Programming**	**Boot**	**CANId**
(Test Mode)	**(hex)**		**(SPS)**	**Memory**	**Type**
ReadDiagnosticInformation	A9	M_5	N/A		UUDT
ReadDataByPacketIdentifier	AA	C	N/A		UUDT
DeviceControl	AE	C	N/A		USDT

Where:

M_1 = This service shall be supported by all ECUs which log DTCs.

M_2 = Boot Memory is only required to support the Data Identifiers (DIDs) which are necessary to program the ECU when the ECU only contains boot software.

M_3 = Mandatory for all programmable nodes which are Emission, Safety, or Theft related, or otherwise required by law.

M_4 = This service is required in boot if it is necessary to write information (e.g., option content) into an ECU before the software reset at the end of the programming event.

M_5 = This service shall be mandatory for all ECUs which log DTCs. For applicable ECUs, support of sub-function parameter $81 (readStatusOfDTCByStatusMask) shall be mandatory, and the functionality provided by the other sub-functions shall be agreed upon by the DRE, service and manufacturing representatives.

C_1 = The wakeUpLinks (sub-function parameter = $04) may be required for gateway devices that are connected to a subnet that supports a wake-up feature that is not available to the tester (e.g., wake-up wire not brought out to the Diagnostic Link Connector (DLC)). If the gateway device is SPS programmable and required to support the wakeUpLinks sub-function parameter, then it shall also support the wakeUpLinks sub-function in boot software. Which sub-functions of this service that an ECU is required to support must be agreed upon by the DRE, service and manufacturing representatives.

C_2 = Certain sub-function parameters may be supported on non-programmable ECUs to support manufacturing needs of the ECU supplier or the vehicle assembly plant.

4.3 Diagnostic Communication Implementation Rules. In applying the rules which follow, a diagnostic request is considered in process, until the node has fulfilled one of the following (depending on the specific service requested):

- The ECU sent the last consecutive frame of an USDT multi-frame positive response.
- The ECU sent an USDT negative response message indicating there is no positive response forthcoming (response code other than $78).
- The ECU sent an USDT single frame positive response message.
- The first (or only) UUDT response is received (for services $AA and $A9).

Note: Application timing requirements for UUDT diagnostic responses provided via services $A9 and $AA are specified in each respective service within this specification. Reference the descriptions of these services for additional detail.

4.3.1 Tester Rules.

1. A tester is not allowed to send a physically addressed request any time a previous physically addressed request is in process.
2. A tester is not allowed to send a physical request any time a functional request has been received that will result in a response, and the response to the functional request is in process.
3. A tester is not allowed to send a functional request that results in a response if a previous physically addressed request is in process.

4. A tester is not allowed to send a functional request that results in a response if a previous functionally addressed request is in process.
5. After sending a functionally addressed request which will result in a response, a tester is not allowed to send a second functionally addressed request that does **not** result in a response (e.g., TesterPresent request) until at least 30 ms has elapsed after sending the first request.
6. A tester shall wait a minimum of 30 ms after sending a functionally addressed request that does not result in a response before sending another functionally addressed request message.
7. A tester can send a functionally addressed request message that does not result in a response (e.g., functional tester present) at any point in time during which the ECU is in the process of receiving or responding to a physically addressed diagnostic request. (In other words the functionally addressed tester present request may even interleave the transmission of frames of a multi-frame physical request message or a multi-frame physical response message.)
8. A tester may also send a diagnostic physical request immediately following a diagnostic functional request as long as the functional request does not require a response.

Table 7 provides an additional overview about the required handling in the tester also taking into consideration the node specific rules given in paragraph 4.3.2. The **Previous Request** column refers to the last diagnostic message sent by the tester and **New Request** column refers to the diagnostic message that the tester is attempting to send next.

Table 7: Tester Communication Implementation Rules

New Request Addressing Type	Previous Request Addressing Type	Handling in the Tester
Physical	Physical	Wait until the completion of the previous physical request (positive response, or negative response other than $78) before transmitting a new physical request.
Physical	Functional with response	Wait until the completion of the previous functional request that requires a response (positive response, or negative response other than $78) before transmitting the physical request.
Physical	Functional without response	The tester is allowed to transmit the physical request immediately (e.g., functional TesterPresent followed by any physical request).
Functional without response	Physical	The tester is allowed to transmit the functional request that does not require a response immediately (e.g., any physical request followed by a functional TesterPresent).
Functional without response	Functional with response	Wait 30 ms after the transmission of the functional request that requires a response before transmitting the functional request that does not require a response.
Functional without response	Functional without response	Wait 30 ms after the transmission of the functional request that does not require a response before transmitting the next functional request that does not require a response.
Functional with response	Physical	Wait until the completion of the previous physical request (positive response, or negative response other than $78) before transmitting the functional request that requires a response.

New Request Addressing Type	Previous Request Addressing Type	Handling in the Tester
Functional with response	Functional with response	Wait until the completion of the previous functional request that requires a response (positive response, or negative response other than $78) before transmitting the next functional request that requires a response.
Functional with response	Functional without response	Wait 30 ms after the transmission of the functional request that does not require a response before transmitting the functional request that requires a response.

4.3.2 Node (ECU) Rules.

1. The node shall be able to receive and process all tester requests which comply with the above rules.
2. If a polling rate is used by an ECU to service CAN controller message object(s), then the polling rate for diagnostic message object(s) shall be < 30ms.

Note: The above rule provides the tester a fixed maximum wait time between requests to avoid overwriting diagnostic requests in the ECUs receive message object(s). This wait time does not apply to the case where a physical request is followed by a functional request that does not require a response (e.g., physical request followed by a functional TesterPresent) or the case where a functional request that does not require a response is followed by a physical request (e.g., functional TesterPresent followed by a physical request) – refer to paragraph 4.3.1.

Note: There is also a requirement for devices on the Low Speed GMLAN subnet to be able to transition into high speed operation during programming within 30 ms of receiving a request to do so. Therefore polling rates for these devices must already be less than 30 ms. Refer to the programming section of this specification and diagnostic services $A5 and $20 for additional information.

4.3.3 Special Considerations for Nodes Operating as an On-board Test Device.

- Any node on a GMLAN link operating as an on-board test device (using the diagnostic request CANids identified in this document) shall adhere to all rules applicable to an off-board tester as defined in section 4.3.1. In addition, an on-board test device shall take steps to ensure it does not interfere with an off-board test device. To accomplish this, the on-board test device shall monitor the data link (all subnets) for any of the reserved ranges of diagnostic CANids (this includes the On Board Diagnostic/European On Board Diagnostic (OBD/EOBD) CANids on the high speed link) for a period of at least 5.1 s prior to the on-board test device transmitting a diagnostic request. If the device detects diagnostic messages on the link, it shall wait until there is at least 5.1 s of diagnostic inactivity on the link before transmitting any diagnostic request. Once an on-board test device begins a diagnostic session, it shall continue to monitor the bus for diagnostic requests made by an off-board tester. If the on-board tester detects a diagnostic request that it did not make then it shall wait for a minimum of 5.1 s of diagnostic inactivity before making another diagnostic request. If the on-board test device sends a diagnostic request which results in a bus off condition then the on-board tester must monitor the bus for no diagnostic communications for a minimum of 5.1 s following the recovery from the bus off condition. An on-board tester shall also wait a minimum of 5.1 s after a programming session has concluded before initiating any diagnostic request.

Note: The programming session is concluded with either a $P3_C$ timeout or a service $20 request at the end of utility file part 1. Refer to Section 9 of this specification for more on the programming process.

Note: The 5.1 s minimum that an on-board tester must wait is derived from the $P2_{CT}$* value defined in paragraph 6.2.2.2 of this specification.

4.3.4 Special Considerations for Nodes Operating On Multiple GMLAN Subnets.

- Gateways shall not transmit diagnostic messages across subnets. Any diagnostic message which is targeted for all GMLAN subnets must be transmitted by the tester on each specific subnet. Any diagnostic message which is targeted to a functional system that spans one or multiple GMLAN subnets must be transmitted by the tester on each subnet where nodes of the addressed functional system reside on.

- A Multiple Subnet Signal Consumer (MSSC) shall process diagnostic services supported on multiple subnets individually. For example, a service $28 received on one subnet would only stop normal communications on the subnet which it is received. The ECU would continue normal communications messages (if otherwise active) on other subnet(s) to which it is attached. Another example of this would be that a tester present request received on a subnet would not reset the P3$_C$ timer on another subnet.
- A MSSC device shall support simultaneous programming events on each of the mulitple subnets to which it is connected. A programming event on one subnet shall not impact or preclude initiating a programming event on another subnet. This requires processing the following services separately on each subnet: service $20, service $28, service $A5, service $1A, and service $3E.

Note: See service $A5 for more information on enabling programming events.

Note: A node shall support the DIDs required for programming on both subnets. Reference the programming chapter within this specification for specific requirements. The other DIDs needed for diagnostics may only be supported on one subnet.

- A MSSC device shall be capable of being diagnosed fully from at least one subnet to which it is connected.
- A MSSC shall document in a Component Technical Specification (CTS), Subsystem Technical Specification (SSTS), or other specification referenced in a CTS or SSTS the diagnostic services that are supported on each subnet.
- A programmable MSSC device shall only be capable of being programmed from one subnet to which it is connected and the $A2 service shall only be supported on this subnet. This subnet that is used for programming should be the subnet with the higher baud rate capability.

Note: A programmable MSSC device is one that utilizes the SPS system (utility file concept) for reprogramming. Refer to Section 9 of this specification for more details.

- If a MSSC receives diagnostic requests on both subnets simultaneously, it must respond to each request within the P2$_C$ time, however depending on the service requested and the capabilities of the node one of the following must happen:

1. For services which the MSSC node supports on both subnets, the MSSC node sends either:

- The positive response on each subnet within P2$_{C.}$
- **OR** the MSSC node sends the positive response on one subnet while sending a negative response $7F $RequestServiceId $78 message on the second subnet within P2$_C$ (indicating a positive response is forthcoming), followed by the positive response after transmitting the positive response on the first subnet. The exceptions are modes $20, $28, and $A5 where the $7F $RequestServiceId $78 is not allowed.

  Note: For the special condition of receiving a service $20 ReturntoNormalMode request on multiple subnets **while in programming mode**, the MSSC shall perform the required reset (see section 8.5.1) after the first received service $20 request. It is the tester's responsibility to coordinate the service $20 requests on the multiple links so that the link, which requires a baudrate switch, receives the service $20 request first if the tester is connected to multiple subnets.

2. For diagnostic services only supported on one subnet (e.g., service $AA), the MSSC sends the positive response on the subnet supporting the service while sending a negative response reject message (e.g., $7F $RequestServiceId $11 for service not supported) on the other subnet (a negative response is only sent if the request is physically addressed).

4.4 Message Identification - Diagnostic CAN Identifiers (CAN Identifiers). Diagnostic CAN Identifiers (CAN Identifiers) are required in order to separate diagnostic messages from Normal Communication messages. The diagnostic CANId in a request message also determines the type of addressing used for that message, either physically addressed (targeted to only a single node) or functionally addressed (targeted to one or multiple nodes).

4.4.1 Diagnostic CANId Definitions. Each GMLAN node shall support the following types of diagnostic CAN Identifiers unless otherwise stated in the descriptions below:

- **Physical Request CAN Identifier (USDT):** This CAN Identifier shall be used to transmit physically addressed single frame or multiple frame request messages to a single ECU utilizing the Network USDT protocol with normal addressing. This CAN Identifier shall also be used by the tester to send FlowControl frames to an ECU if the transmission of data from an ECU to the tester requires a multiple frame transmission.

- **AllNode Functional Request CAN Identifier (USDT):** This CAN Identifier shall be used to transmit functionally addressed single frame request messages to a functional system which consists of one or multiple ECUs utilizing the Network USDT Protocol, with extended addressing (EA). All nodes on the subnet where the request was sent shall receive this CANId and evaluate the extended address in order to determine if the request is valid for that specific ECU. The extended address represents a functional system address (see the section on Functional System Assignments for more information).

Note: Only nodes on the subnet where the request was sent shall receive this message. This is because the diagnostic strategy does not allow gateways to pass diagnostic messages on to other subnets.

- **Physical USDT Response CAN Identifier:** This CAN Identifier shall be used to transmit physically addressed single frame or multiple frame response messages to the tester utilizing the Network USDT protocol with normal addressing. This CAN Identifier shall also be used by the ECU to send FlowControl frames to the tester if the transmission of data from the tester requires a multiple frame transmission.
- **Physical UUDT Response CAN Identifier:** This CAN Identifier shall be used to transmit physically addressed single frame UUDT response messages to the tester.
- **OBD/EOBD CAN Identifiers:** These CAN Identifiers shall be supported only by emission related devices. A specific range of Identifiers are reserved for OBD/EOBD purposes and unused identifiers in this range shall not be used for other purposes. The range of reserved OBD/EOBD identifiers is shown in Table 8. OBD/EOBD CAN identifiers may also be optionally used by emission related ECUs to support enhanced diagnostics. Refer to paragraph 4.4.4.1 for further details.

4.4.2 CAN Identifier Memory Map Model. The following requirements apply to the GMLAN diagnostic CAN Identifiers used by all ECUs with the exception of any emission related ECU(s) that have opted to use their OBD/EOBD CAN Identifiers to support enhanced diagnostics (refer to paragraph 4.4.4.1 for rules regarding the optional assignment of OBD/EOBD CAN Identifiers by emissions related ECUs for enhanced diagnostics):

- The physical request CAN Identifiers shall be higher priority than the UUDT response CAN Identifiers to be able to shut down a node which is sending periodic messages.
- The UUDT response CAN Identifiers shall be higher priority than the USDT response CAN Identifiers to allow the periodic messages to be sent prior to a multiple frame response message which is transmitted to the tester.
- Since the priority of the CANId is inversely proportional to its numerical value (i.e., the lower the CANId number, the higher the priority), the following is true:

USDT_RequestCANId < UUDT_ResponseCANId < USDT_ResponseCANId

Figure 2 gives an overview of the reserved GMLAN diagnostic CAN Identifier ranges to provide a better understanding of where the GMLAN diagnostic CAN Identifiers are located. It also shows some of the GMLAN CAN Identifiers reserved for normal communication, such as Virtual Network Management Frames (VNMF).

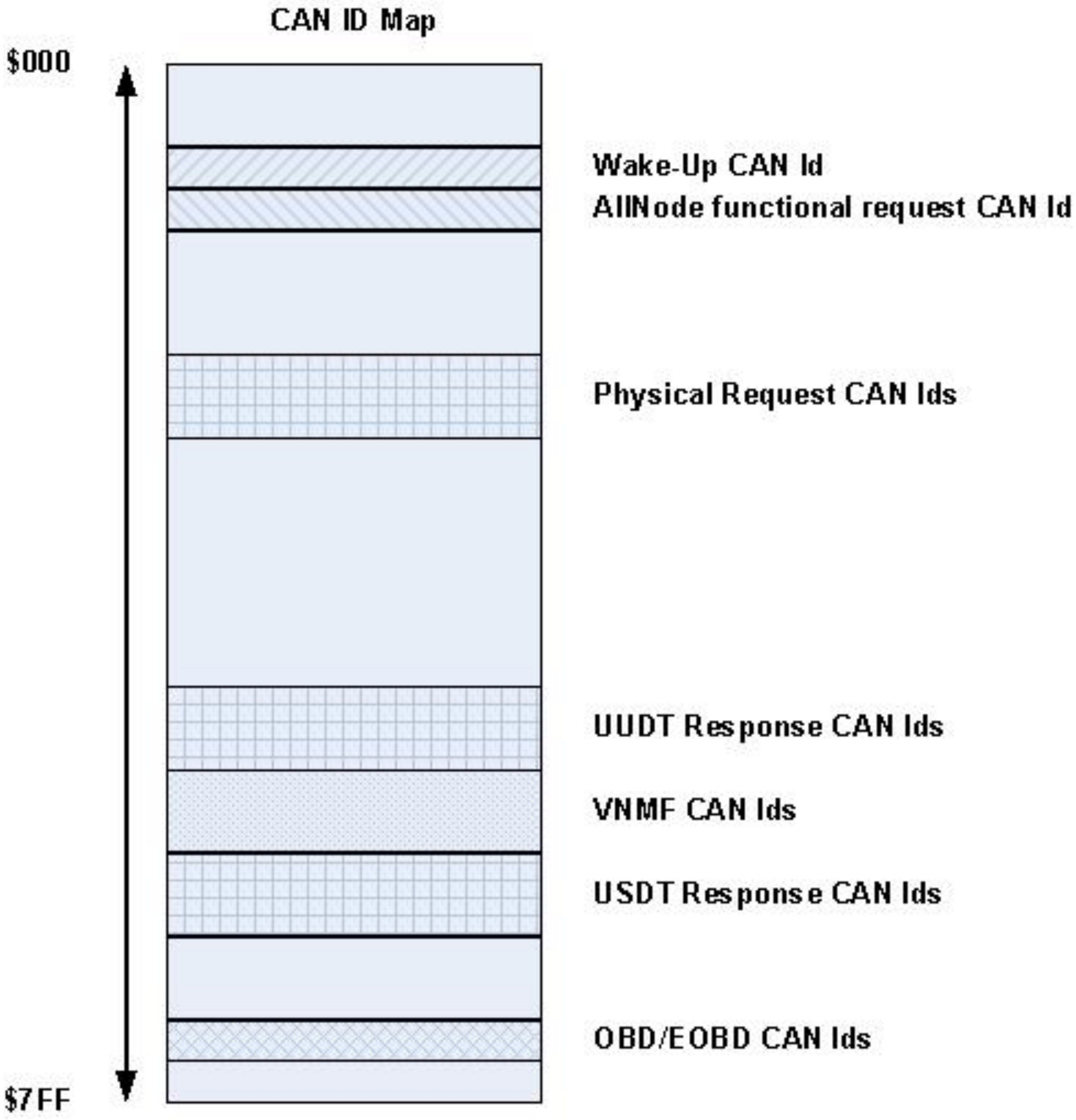


Figure 2: CAN Identifier Memory Map Model

4.4.3 CAN Identifier Table. Table 8 defines the ranges for the GMLAN diagnostic CAN Identifiers to be reserved for each GMLAN sub-network.

Table 8: Diagnostic CAN Identifier Assignments

Description	**Addressing**		
Identifier Type	**Address Type**	**Address Scheme**	**CAN Identifier**
Wake-Up [Note 1]	-	-	$100
AllNode request	Functional	Extended	$101
Diagnostic request - reserved	-	Extended	$102
USDT request - reserved	Physical	Normal	$240

Description	Addressing		
Identifier Type	Address Type	Address Scheme	CAN Identifier
USDT request	Physical	Normal	$241 thru $25F
UUDT response - reserved	Physical	Normal	$540
UUDT response	Physical	Normal	$541 thru $55F
UUDT response (optional for emissions-related ECUs only)	Physical	Normal	$5E8 thru $5EF
USDT response - reserved	Physical	Normal	$640
USDT response	Physical	Normal	$641 thru $65F
OBD/EOBD	Functional/physical	Normal	$7DF thru $7EF

Note 1: Defined by GMLAN Message Strategy Specification.

4.4.4 Rules For CAN Identifier Assignment. The CAN Identifiers of a node shall be chosen in a way that the lower byte of each node CAN Identifier (physical request, USDT response, UUDT response) can be interpreted as a unique offset relative to the diagnostic CAN Identifier range (for a given subnet). For example, the relative offset of a node with the USDT response CAN Identifier $646 is $46. This relative offset can be used to calculate all other diagnostic CAN Identifiers of this node: UUDT = $546, Phys. Request = $246. The tester only needs to exchange the most significant nibble of the 11-bit CAN Identifier to the one used for the CAN Identifier ranges. The most significant nibble of the 11-bit CAN Identifier can be interpreted in the following way (which does not mean that the whole range of 256 CAN Identifiers is reserved for GMLAN diagnostic).

- $**2**xx = physical request CAN Identifier.
- $**5**xx = UUDT response CAN Identifier.
- $**6**xx = USDT response CAN Identifier.

Note: Emission-related ECUs may use optional OBD CANid assignments. Refer to paragraph 4.4.4.1.

Example: Three (nodes) connected to the LS-CAN sub-network (Table 9).

Table 9: CAN Identifier Assignments Example [Note 1]

ECU Diagnostic Address (physical node Id)	PhysicalReqId	USDTRespId	UUDTRespId
$22 $42 $11	$24A $243 $25E	$64A $643 $65E	$54A $543 $55E

Note 1: PhysicalReqId = Physical Request CAN Identifier; **USDTRespId** = USDT Response CAN Identifier; **UUDTRespId** = UUDT Response CAN Identifier

Note: There is no numerical relationship between the ECU Diagnostic Address and the Diagnostic CAN Identifiers. The Diagnostic address shall be unique for every ECU on the vehicle but the diagnostic CAN Identifiers are only required to be unique on a given subnet. This means that two devices on different subnets may use the same set of Diagnostic CAN Identifiers but their Diagnostic addresses are different.

4.4.4.1 Optional CAN Identifier Assignments for Emission Related ECUs. Table 10 details the usage of the 11-bit OBD/EOBD reserved CAN Identifiers. Each GMLAN emission related ECU that supports legislated diagnostics (as defined in SAE J1979/ISO 15031-5) with 11-bit CAN Identifiers must support the $7DF functional request CANId, one of the eight reserved OBD/EOBD physical request CAN Identifiers, and the OBD/EOBD physical response CANId associated with the supported physical request CANId. All of the OBD/EOBD CAN Identifiers utilize the USDT protocol and normal addressing. See paragraph 4.5.1.1.1 for more information on addressing types.

Table 10: OBD/EOBD 11-Bit USDT CAN Identifier Assignments

CAN Identifier	Description
$7DF	CAN Identifier for functionally addressed request messages sent by the external test equipment.
$7E0	Physical request CAN Identifier from the external test equipment to OBD ECU #1
$7E8	Physical response CAN Identifier from OBD ECU #1 to the external test equipment
$7E1	Physical request CAN Identifier from the external test equipment to OBD ECU #2
$7E9	Physical response CAN Identifier from OBD ECU #2 to the external test equipment
$7E2	Physical request CAN Identifier from the external test equipment to OBD ECU #3
$7EA	Physical response CAN Identifier from OBD ECU #3 to the external test equipment
$7E3	Physical request CAN Identifier from the external test equipment to OBD ECU #4
$7EB	Physical response CAN Identifier from OBD ECU #4 to the external test equipment
$7E4	Physical request CAN Identifier from the external test equipment to OBD ECU #5
$7EC	Physical response CAN Identifier from OBD ECU #5 to the external test equipment
$7E5	Physical request CAN Identifier from the external test equipment to OBD ECU #6
$7ED	Physical response CAN Identifier from OBD ECU #6 to the external test equipment
$7E6	Physical request CAN Identifier from the external test equipment to OBD ECU #7
$7EE	Physical response CAN Identifier from OBD ECU #7 to the external test equipment
$7E7	Physical request CAN Identifier from the external test equipment to OBD ECU #8
$7EF	Physical response CAN Identifier from OBD ECU #8 to the external test equipment

GMLAN emission related components that need to utilize the hardware filtering capabilities provided with CAN controllers may run into a situation where the number of messages that need to be filtered cannot be handled with the number of message objects available in the CAN controller. In such circumstances, an emission related ECU may choose to use the OBD/EOBD diagnostic CAN Identifiers to also support enhanced diagnostics. This is only valid for ECUs that support OBD/EOBD diagnostics with the 11-bit header format.

Emission related ECUs utilizing the OBD/EOBD CAN Identifiers for enhanced diagnostics shall assign the enhanced diagnostic CAN Identifiers per Table 11. The range of CAN Identifiers from $5E8 thru $5EF on the high speed bus has been reserved in order to maintain the relationship between the USDT response CANId and the UUDT response CANId as outlined in paragraph 4.4.4.

Table 11: OBD CANId Assignments for Enhanced Diagnostics

OBD ECU	OBD/Enhanced Diagnostic Physical Request CANId	OBD/Enhanced Diagnostic Physical Response CANId	Enhanced Diagnostic UUDT Response CANId
1	$7E0	$7E8	$5E8
2	$7E1	$7E9	$5E9
3	$7E2	$7EA	$5EA
4	$7E3	$7EB	$5EB
5	$7E4	$7EC	$5EC
6	$7E5	$7ED	$5ED
7	$7E6	$7EE	$5EE
8	$7E7	$7EF	$5EF

Note: Usage of OBD/EOBD CAN Identifiers for enhanced diagnostics does not have any effect on the maximum number of ECUs allowed on the HSCAN network.

4.4.5 SPS Special Case ECU Programming CANId Assignments. The reserved diagnostic CAN Identifiers outlined in Table 7 are the CAN Identifiers used for diagnostics on ECUs which are either:

- Programmable via the Service Programming System (SPS) and have been completely programmed, **or**
- Fully functional ECUs which are not programmable via the Service Programming System.

The reserved diagnostic CAN Identifiers are further referenced in this document as permanent diagnostic CAN Identifiers. An ECU which is SPS programmable and has not been completely programmed (e.g., an ECU shipped to a vehicle assembly plant with boot and operational software and no calibration data), may or may not comprehend its permanent diagnostic CAN Identifiers. This may be the case where a corporate common SPS programmable ECU is used across multiple platforms and the platforms are not able to standardize the permanent diagnostic CAN Identifiers.

Note: It is also possible for an SPS programmable ECU which is not completely programmed to be able to use its permanent diagnostic CAN Identifiers during programming. This would require that the ECU have the permanent diagnostic CAN Identifiers stored in memory which is accessible to the software executing upon boot while the ECU is not completely programmed.

In order to avoid the proliferation of part numbers on SPS programmable ECUs solely due to diagnostic CANId conflicts across platforms, special case SPS CAN Identifiers have been defined. Each ECU requiring the special case CAN Identifiers shall support a special case point-to-point USDT diagnostic request CANId (further referenced as the SPS_PrimeReq CANId) and a special case USDT diagnostic response CANId (further referenced as the SPS_PrimeRsp CANId). Table 12 defines the assignment of the special case CAN Identifiers to the ECU.

Table 12: SPS Special Case ECU CAN Identifier Assignment

	CAN Id Bit position	
	10 8	7 0
SPS_PrimeReq USDT request CAN Id	$0	ECU Diagnostic Address
SPS_PrimeRsp USDT response CAN Id	$3	ECU Diagnostic Address

The diagnostic address embedded in the special case SPS_PrimeReq and SPS_PrimeRsp diagnostic request and response CAN Identifiers shall be the physical node (ECU) address. See Appendix D for further details on ECU diagnostic addresses.

An ECU which only comprehends the special case diagnostic CAN Identifiers shall only respond to diagnostic requests once the tester enables their use. Additional information about the SPS_PrimeReq and

SPS_PrimeRsp diagnostic CAN Identifiers and the process to enable them can be found in this specification in paragraph 9.1.1 - Enabling Diagnostic Responses on SPS_TYPE_C ECUs. The special case CAN Identifiers must be enabled by the tester after normal communications have been disabled in order to prevent potential CANId conflicts between normal communications messages and diagnostic messages using the special case CAN Identifiers.

4.5 Message Addressing. Diagnostic messages may be addressed either physically (targeted to a single node) or functionally (addressed to one or multiple nodes). The type of message addressing used for diagnostic request messages is determined by the diagnostic CANId used. All diagnostic response messages will be physically addressed, using one of the reserved range of physical response CAN Identifiers (defined previously).

4.5.1 Frame Data Byte Definition Based On Address Method. GMLAN diagnostic messages shall support different combinations of addressing and segmentation methods as determined by the CANId and diagnostic service used.

4.5.1.1 General Frame Format. The following sections describe the addressing schemes and CAN frame formats including the Network Layer Protocol Control Information (PCI). The sections following the general description section describe the specific usage for GMLAN enhanced diagnostics. Detailed descriptions of the addressing schemes and the CAN frame formats can be found in ISO 15765-2 and OSEK/COM.

4.5.1.1.1 Addressing Schemes (Table 13).

Table 13: Addressing Scheme Summary

Addressing Scheme Name	Identifier	CAN frame data field							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NORMAL	CAN Id								
EXTENDED	CAN Id	EA							

4.5.1.1.2 Protocol Control Information (PCI) Formats for GMLAN. All references to Protocol Control Information (PCI) within this specification relate only to the network layer (referenced as N_PCI within ISO 15765-2). As shown in Tables 14 thru 17 below, the PCI information can be 1, 2, or 3 bytes in length depending on the frame type.

Table 14: Location of Protocol Control Information (PCI) Using Normal Addressing

<table>
<tr><th>Addressing Scheme Name</th><th>Identifier</th><th colspan="8">CAN frame data field</th></tr>
<tr><th>Description</th><th>ID</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th></tr>
<tr><td>NORMAL - SingleFrame</td><td>CAN Id</td><td>PCI</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>NORMAL - FirstFrame</td><td>CAN Id</td><td colspan="2">PCI</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>NORMAL - FlowControl</td><td>CAN Id</td><td colspan="3">PCI</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>NORMAL - ConsecutiveFrame</td><td>CAN Id</td><td>PCI</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

Table 15: Location of Protocol Control Information (PCI) Using Extended Addressing

Addressing Scheme Name	Identifier	CAN frame data field							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
EXTENDED - SingleFrame	CAN Id	EA	PCI						
EXTENDED - FirstFrame	CAN Id	EA	PCI						
EXTENDED - ConsecutiveFrame	CAN Id	EA	PCI						

Table 16: UUDT Frame Format Using Normal Addressing

Addressing Scheme Name	Identifier	CAN frame data field							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NORMAL - UUDT frame	CAN Id								

Table 17: Encoding of Protocol Control Information (PCI) Bytes of USDT Frames

USDT PCI Encoding		Protocol Control Information (PCI)									
		Byte #1 Note 1								**Byte #2**	**Byte #3**
PCI Frame	**Mnemonic**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**		
SingleFrame	SF	0	0	0	0	DL				N/A Note 2	N/A Note 2
FirstFrame	FF	0	0	0	1	DL high				DL low	N/A Note 2
ConsecutiveFrame	CF	0	0	1	0	SN				N/A Note 2	N/A Note 2
Flow Control frame	FC	0	0	1	1	FS				BS	STmin
Reserved		$40 through $FF								Reserved	Reserved

Note 1: The byte number applies to the position of the information within the PCI. For normal addressing the PCI itself starts on position #1 of the CAN frame, for extended addressing it starts on position #2 in the CAN frame. A detailed description of the PCI parameters can be found in ISO 15765-2.

Note 2: Bytes labeled as N/A may be used as data bytes in frames of that type.

Table 18 contains general PCI parameter descriptions as defined by ISO 15765-2. This specification places restrictions on the allowed ECU and tester implementation of some of these parameters. See paragraph 6.3 for detailed GMLAN implementation requirements.

Table 18: USDT PCI Parameter Definitions

PCI Parameter Name	Definition
DL	This 4-bit parameter is used to define the usable Data Length (DL) of a single frame (SF) message. DL can range from 0 to 7 or 6 since the maximum number of useable data bytes in a single frame (SF) message is 7 if normal addressing is used and 6 if extended addressing is used. The DL parameter differs from the data length code provided in each CAN frame as it only reflects the length of the usable data bytes (i.e., It is possible to pad messages to a fixed maximum length of 8 bytes even though the number of meaningful bytes is less than 8. In this case, the CAN frame data length code would reflect 8 bytes and the DL parameter would reflect the usable bytes).
DL High	This 4-bit parameter is combined with the DL Low parameter to allow 12 total bits to define the data length of a message. The 12 bits allows for a maximum of 4095 bytes in a single message. DL High occupies the 4 most significant bits of the 12 total message length bits
DL Low	This 8-bit parameter is combined with the DL high parameter to allow 12 total bits to define the data length of a message. The 12 bits allows for a maximum of 4095 bytes in a single message. DL Low occupies the 8 least significant bits of the 12 total message length bits.
SN	The Sequence Number (SN) is a 4-bit parameter which is used as a rolling counter to keep track of the successive consecutive frames of a multi-frame message. The first consecutive frame (CF) will have a SN value of 1. The SN value for each subsequent frame shall increment by 1 up to $F and then the next frame shall have a value of 0. This process shall continue ($0 thru $F, $0 thru $F) until all frames of the multi-frame message have been successfully transmitted. Flow control frames transmitted during the transmission of the multi-frame message do not affect this rolling counter.
FS	The Flow Status (FS) is a 4-bit parameter used to indicate to a sender of a multi-frame message whether it can proceed with the transmission of consecutive frames (CF). A FS value of $0 indicates ContinueToSend (ConTS) **Note 1** and means that the sender can continue to transmit the consecutive frames using the Block Size (BS) and Separation Time (ST) values from the flow control (FC) frame transmitted by the receiver. A FS value of $1 indicates a Wait (WT) condition. A wait condition means that the sender has to pause sending consecutive frames until after the receiver transmits a flow control frame with and FS value of $0 (ConTS). A FS value of $2 indicates an Oveflow (OVFLW) condition where the receiver's buffer is not large enough to receive the incoming message. This flow status indicates that reception of the message is being terminated.
BS	The Block Size (BS) is a 1-byte parameter that defines the number of consecutive frames that the sender can transmit before waiting for a flow control frame from the receiver. There shall never be a FC frame following the last CF frame of a multi-frame message.
ST_{min}	The Separation Time minimum (STmin) is a 1-byte parameter that defines the minimum time lapse between consecutive frames. The units on this parameter are 1 ms per count for values between $00 and $7F. For ST_{min} values between $F1 and $F9, the resolution is 100 µs (microseconds) per count where $F1 = 100 µs and $F9 = 900 µs. All other values are reserved.
WFT_{max} (This parameter is not allowed in GMLAN implementation)	The maximum number of Wait Frame Transmissions (WFT_{max}) parameter is used to determine the maximum number of allowed consecutive flow control frames with an FS value of WT. This parameter is local to each ECU and is not transmitted as part of a flow control frame. A WFT_{max} value of $0 indicates that the ECU is not allowed to send FC.WT frames. ISO 15765-2 restricts the maximum value of the parameter $WFT_{max.}$

Note 1: International Organization for Standardization (ISO) uses the acronym CTS for this parameter. In this specification it shall be referred to as ConTS to avoid confusion with Component Technical Specification which uses the acronym CTS in this specification.

4.5.1.2 Normal Addressing - Physical Request and Response Messages. Physically addressed request messages, USDT Response messages, and UUDT Response messages shall use a “normal addressing” frame format where the CANId (one of the reserved physical CAN Identifiers) is the only data used to indicate the target node. The following three tables show the format for diagnostic message with normal addressing.

In the tables, the following legend is used:

- SIDRQ = Service Identifier Request.
- SIDPR = Service Identifier Positive response (SID + $40).
- LEV = Sub-Function Parameter $Level (usage depends on the diagnostic service).
- MSG# = Message Number. This is a one byte value which is used to determine the contents of the remaining seven bytes of the message. (See note below.)
- Other Values in Tables 19 thru 21 are defined in the Terms and Abbreviations section of this specification.

Note: The use of a message number was chosen for UUDT messages instead of a SIDPR in certain critical diagnostic services because it allows one additional byte of data in the message (with the SIDPR approach, another data byte would be needed to determine the contents of the remaining data bytes). Refer to Diagnostic Service $A9 and $AA.

Table 19: Diagnostic Request Frame Examples with Normal Addressing

Addressing Scheme Name	**Identifier**	**CAN frame data field**							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NORMAL - SingleFrame	CAN Id	PCI	SIDRQ	LEV/ Data	Data	Data	Data	Data	Data
NORMAL - FirstFrame	CAN Id	PCI		SIDRQ	LEV/ Data	Data	Data	Data	Data
NORMAL - ConsecutiveFrame	CAN Id	PCI	Data	Data	Data	Data	Data	Data	Data

Table 20: Diagnostic USDT Response Frame Format Examples

Addressing Scheme Name	**Identifier**	**CAN frame data field**							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NORMAL - SingleFrame	CAN Id	PCI	SIDPR	Data	Data	Data	Data	Data	Data
NORMAL - FirstFrame	CAN Id	PCI		SIDPR	Data	Data	Data	Data	Data
NORMAL - ConsecutiveFrame	CAN Id	PCI	Data	Data	Data	Data	Data	Data	Data

Table 21: Diagnostic UUDT Response Frame Format Example

Addressing Scheme Name	**Identifier**	**CAN frame data field**							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
NORMAL - UUDT frame	CAN Id	MSG#	Data	Data	Data	Data	Data	Data	Data

4.5.1.3 Normal Addressing - Flow Control Frames. Table 22 shows the definition of the Flow Control frame generated by the network layer of the node (or the tester) when receiving multi-frame messages.

Table 22: Flow Control Frame Examples

Addressing Scheme Name	Identifier	CAN frame data field							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
FlowControl	CAN Id	PCI							

4.5.1.4 Extended Addressing - Functional Request Messages. Functionally addressed request messages shall always be single frame and shall use the “extended addressing” frame format, where the combination of CANId (one of the reserved Functional CAN Identifiers) and an additional data byte (extended address byte, or EA) indicates the target node(s). Usually, the extended address byte will designate a node or group of nodes operating as a specific functional sub-system. However, all resulting response messages will be formatted as physical diagnostic responses with normal addressing (USDT single or multi-frame response, or a UUDT response. See diagnostic USDT and UUDT response frame format examples above).

Diagnostic request messages using extended addressing shall always be USDT messages and have the specific frame formats shown in Table 23 (the legend used is the same as the three previous tables).

Table 23: Diagnostic Request Frame Examples with Extended Addressing

Addressing Scheme Name	Identifier	CAN frame data field							
Description	**ID**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
EXTENDED - SingleFrame	CAN Id	EA	PCI	SIDRQ	LEV/ Data	Data	Data	Data	Data

Note: See OSEK-COM specification for further details about the UUDT/USDT protocol (message formats, service primitives).

4.6 ECU Frame Padding Requirements. GMLAN ECUs shall be capable of receiving diagnostic request messages that are padded. GMLAN ECUs should be capable of receiving diagnostic messages that are not padded.

Note: Frame padding is adding additional data bytes to a transmitted frame (e.g., single frame, flow control frame, or the last frame of a multi-frame message) to ensure that the data length code in the CAN header control field is fixed to 8 bytes.

An ECU can choose to transmit enhanced diagnostic response messages with or without padding (without padding is preferred to minimize bus bandwidth). These requirements are applicable to the ECU when executing either the application software or the boot software. An ECU that does not support the reception of non-padded diagnostic request messages shall document this in a CTS, SSTS, or supplemental diagnostic specification referenced in a CTS or SSTS.

Note: This specification places no requirements on the value of the pad bytes used in either a diagnostic request or response message. However, it is recommended to use values of either $55 or $AA (alternating bit patterns).

Note: The padding requirements detailed in this section deal with frames transmitted for the enhanced diagnostic services described in Section 8 of this document, however, implementation of diagnostic services legislated by OBD/EOBD regulations for ECUs classified as “emission” devices shall comply with ISO 15765-4 which specifies that OBD diagnostic response messages must be padded.

4.7 Communication Layer Interaction. The interaction between the communication layers in the tester and the ECU(s) are guided by the conventions discussed in OSI Service Conventions (ISO/TR 8509). Each layer provides a set of service primitives, which transports a certain type of parameters and data. The service specifications of ISO 11898 and OSEK-COM do not specify any implementation requirements but provide abstract service prototypes to support data exchange with the layers.

4.7.1 GMLAN Communication Layer Interaction. Figures 3 and 4 shall illustrate how the communication layers for a GMLAN based system interact:

- Application Layer: GMLAN (e.g., GMW3110).
- Network Layer: OSEK-COM (USDT/UUDT Protocol).
- Data Link Layer: ISO 11898 (CAN).
- Physical Layer: GMW3122 (DW-CAN), GMW3089 (SW-CAN).

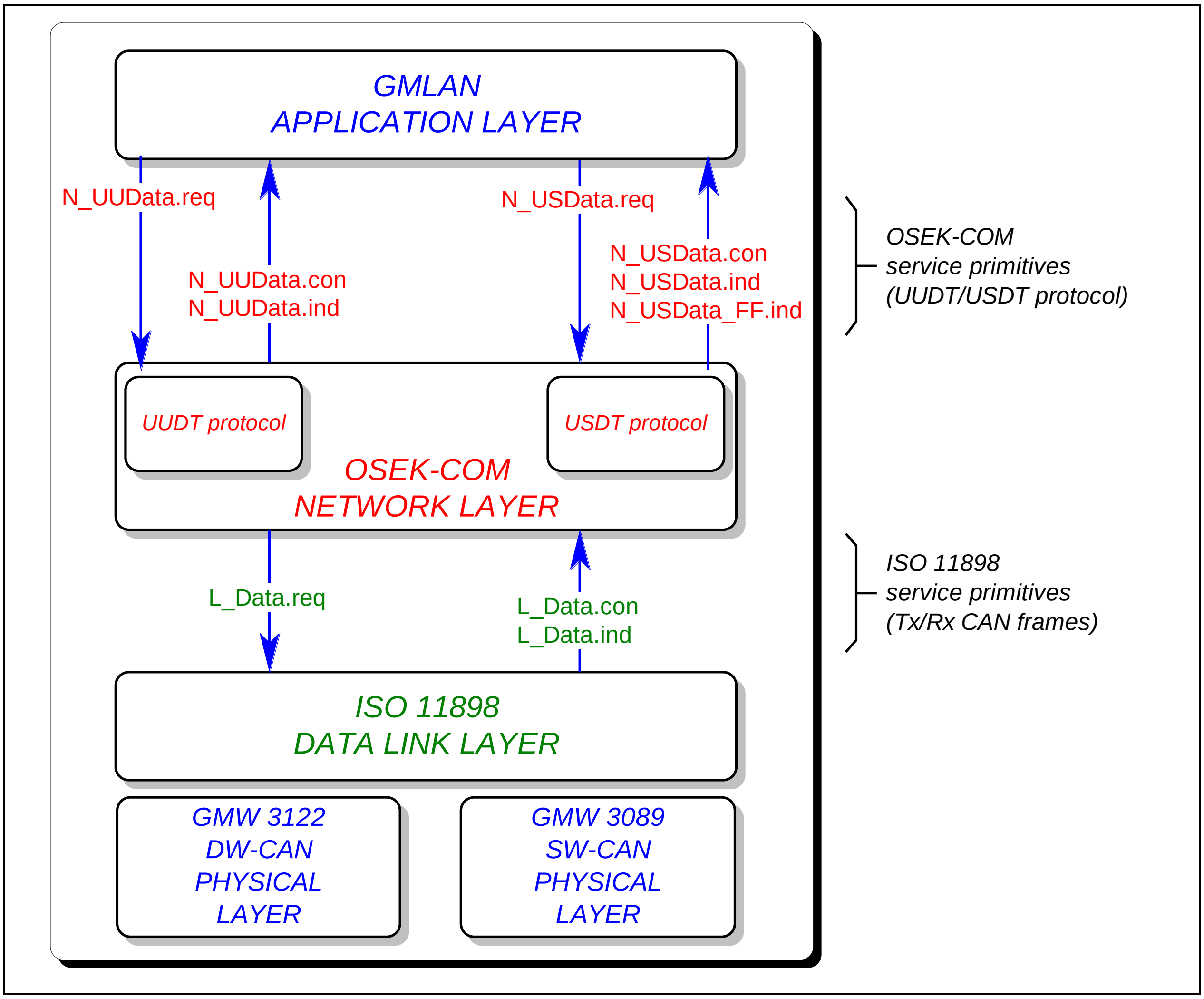


Figure 3: OSI Layer Interaction Model

A detailed description of the service primitives, and the data passed to each layer and retrieved from each layer (e.g., address information provided to/from the Network Layer) can be found in OSEK-COM (Network Layer) and ISO 11898 (CAN Data Link Layer).

ISO 11898 provides service primitives for an unacknowledged data transfer (and unacknowledged remote data request) with up to 8 additional user data bytes. ISO 11898 only allows addressing via the CAN Identifier.

OSEK-COM provides a set of service primitives for the transmission and reception of messages with up to 4095 user data bytes. OSEK-COM provides a segmentation method for messages which do not fit into a single CAN frame as provided by ISO 11898 and, in addition, multiple addressing schemes (e.g., normal addressing: CAN Identifier; extended addressing: CAN Identifier, extended address = 1st data byte of CAN frame). OSEK-COM uses the unacknowledged data transfer service primitives of ISO 11898 to transmit and receive CAN frames.

Figure 4 gives an overview of the occurrence of the ISO 11898 and OSEK-COM service primitives during a request/response sequence, where the tester transmits a SingleFrame request message to a single ECU using the USDT protocol and the ECU responds with a multiple frame response message (two (2) ConsecutiveFrames) using the USDT protocol, too.

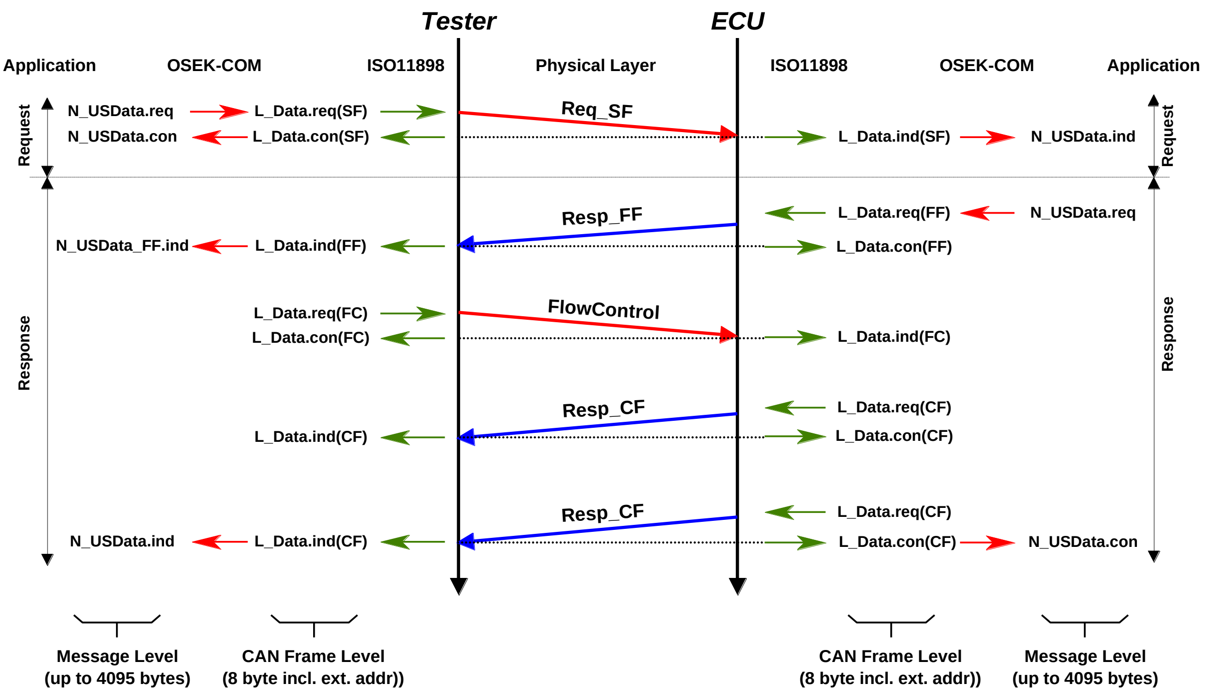


Figure 4: Service Primitives During a Request/Response Sequence in the Tester and the ECU

The service specifications of ISO 11898 and OSEK-COM do not specify any implementation requirements but provide abstract service prototypes to support data exchange with the layers. The service primitive below is an extraction of ISO 11898 and shall be treated as an example for service primitives:

Semantics of the L_Data.req primitive:

L_Data.req (IDE
IDENTIFIER
DLC
DATA)

Where:

IDE = Identifies the IDENTIFIER’s length

IDENTIFIER = Identifies the data and its priority (CAN Identifier)

DLC = Data Length Code

DATA = Data the user wants to transmit

4.8 Network Layer Buffer Requirements. The USDT protocol allows for segmented messages up to a maximum of 4095 bytes. The amount of memory that each ECU reserves for network layer support shall be based on diagnostic and normal communication needs. A SPS programmable ECU may have different network layer buffer size requirements for normal operations (e.g., normal communications and diagnostics) than it does while it is being programmed.

4.8.1 Buffer Requirements for Normal Operation and Diagnostics. The size of the network layer buffer used during normal vehicle operation (including diagnostics and excluding SPS programming) shall be optimized to minimize RAM requirements while still allowing for the reception and transmission of the largest normal communications or diagnostic message expected by that ECU.

Note: If the ReadMemoryByAddress ($23) service is supported in an ECU, then the DRE, the appropriate Service Operations engineer(s), and the appropriate Manufacturing engineer(s) must determine the maximum amount of memory required to be retrieved via a single request of this service and take this into consideration when determining the network layer buffer requirements.

The actual buffer size shall be documented in a CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents.

4.8.2 Buffer Requirements During SPS Programming. During SPS programming, an ECU shall maximize the size network layer buffer to the largest possible value while taking into account ECU RAM resources, the size of the programming algorithm to be downloaded, and the 4095 byte restriction provided by the USDT protocol.

An ECU may have a different buffer size during SPS programming than it would during normal operations. The RequestDownload ($34) service is used during a programming event to tell an ECU that it is about to be programmed. An ECU can use the receipt of this diagnostic service to free up RAM resources for programming which would otherwise be used to support normal vehicle functionality. All ECUs shall perform a software reset at the conclusion of a programming event at which time the ECU would reinitialize RAM and once again begin using the allocated memory for normal operations.

4.9 Diagnostic Message Sequence Examples.

4.9.1 General USDT and UUDT Request/Response Sequence Examples. Figure 5 shows the general structure of a request/response scheme for GMLAN. In this example, the request message is a multi-frame USDT message and the response message is a single frame USDT message. Figure 5 and Table 24 illustrate how the sequence number embedded in each consecutive frame of a multi-frame message shall be treated.

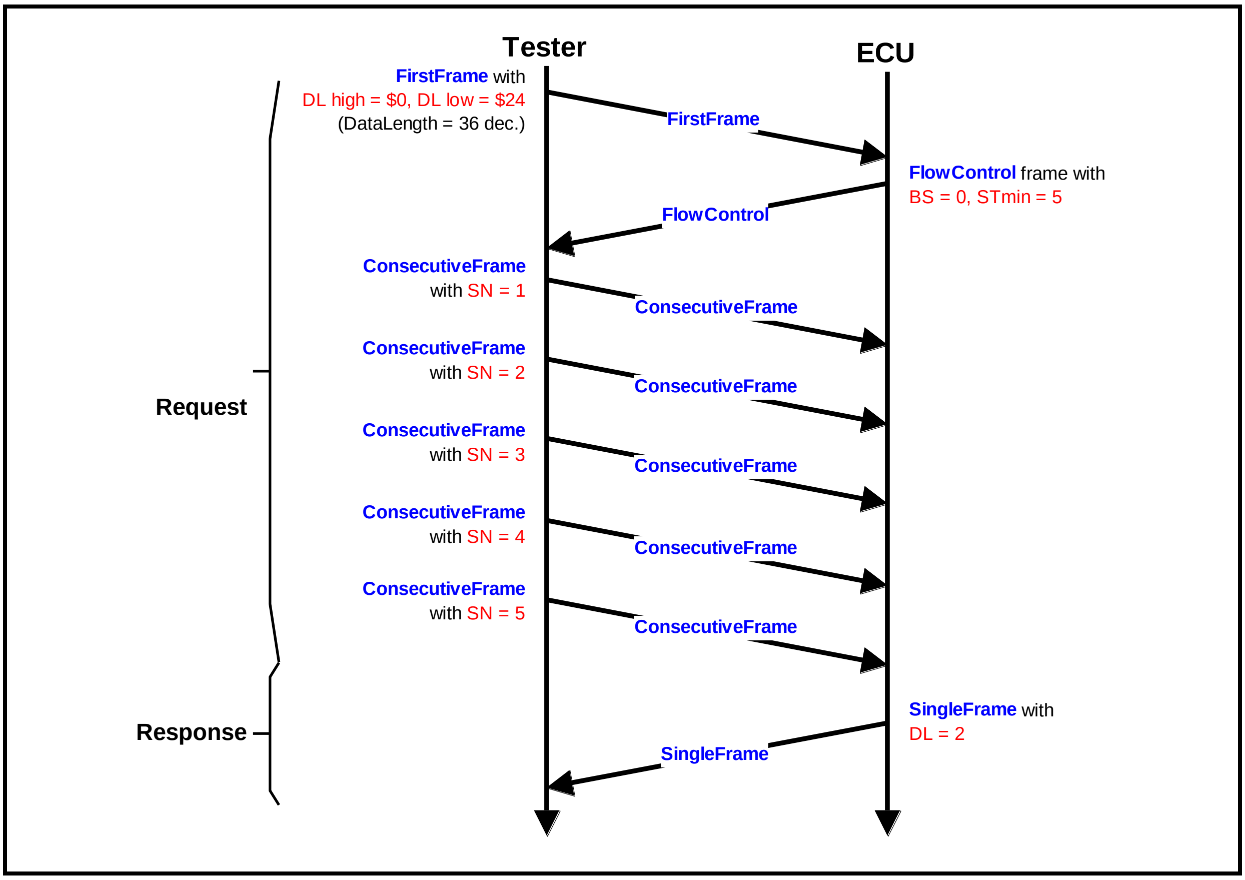


Figure 5: USDT Message Sequence Chart Example

Table 24: USDT Message Flow Example

T = Frame Sent By Tester, N = Frame Sent By Node (shaded area = PCI for USDT frames)									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
Request									
T(USDT-**FF**)	241	10	24	3B	45	01	02	03	04
N(USDT-**FC**)	641	30	00	05	---	---	---	---	---
T(USDT-**CF**)	241	21	05	06	07	08	09	0A	0B
T(USDT-**CF**)	241	22	0C	0D	0E	0F	10	11	12
T(USDT-**CF**)	241	23	13	14	15	16	17	18	19
T(USDT-**CF**)	241	24	1A	1B	1C	1D	1E	1F	20
T(USDT-**CF**)	241	25	21	22	---	---	---	---	---
Positive Response									
N(USDT-**SF**)	641	02	7B	01	---	---	---	---	---

Figure 6 and Table 25 show the general structure of a request/response scheme for GMLAN using UUDT messages to transmit the requested data (response). In this example, the request message is a single-frame USDT message and the response consists of multiple UUDT messages.

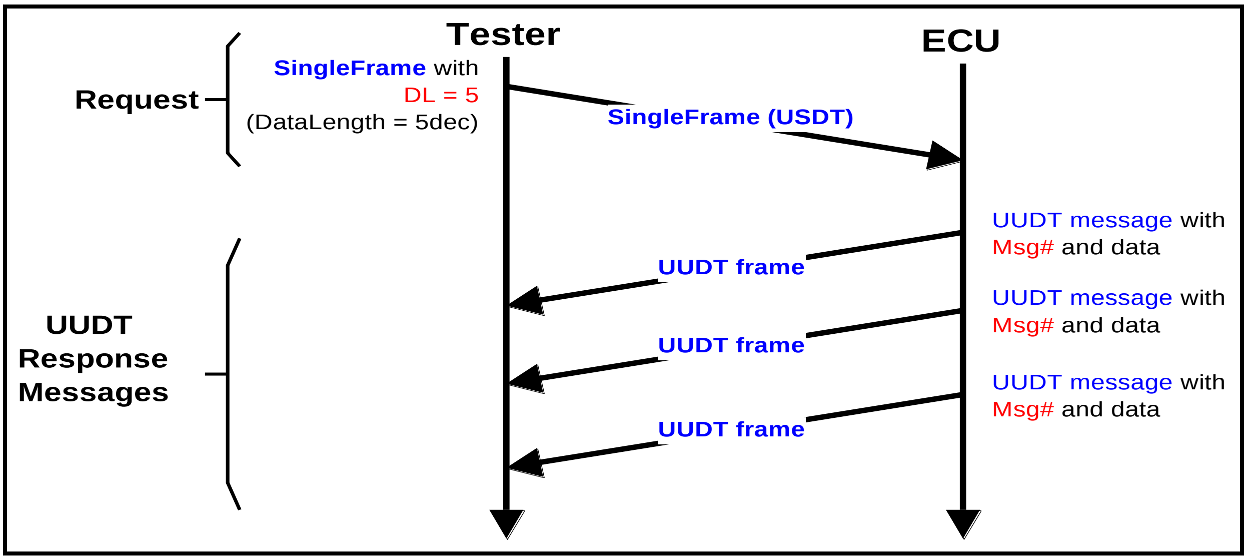


Figure 6: UUDT Diagnostic Response Message Sequence Chart Example

Table 25: UUDT Diagnostic Response Message Flow Example

T = Frame Sent By Tester, N = Frame Sent By Node (shaded area = PCI for USDT frames)									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
Request									
T(USDT-**SF**)	241	05	AA	01	22	A1	10	---	---
UUDT Diagnostic Response messages									
N(**UUDT**)	541	22	AD	12	---	---	---	---	---
N(**UUDT**)	541	10	12	7B	48	32	90	---	---
N(**UUDT**)	541	A1	98	34	45	---	---	---	---

4.9.2 Interleaving Single Frame and Multi-Frame Diagnostic Messages. The Enhanced Implementation of Diagnostic Services specification supports the concept of interleaving messages. A multiple frame message which is sent by either the tester or ECU can be interleaved by other single frame messages which must have a different frame Identifier (e.g., CAN Identifier). This concept supports the feature of periodic messages sent by one or multiple ECU(s) while another ECU sends a multiple frame response message. See Figure 7 for a graphical example of interleaving messages.

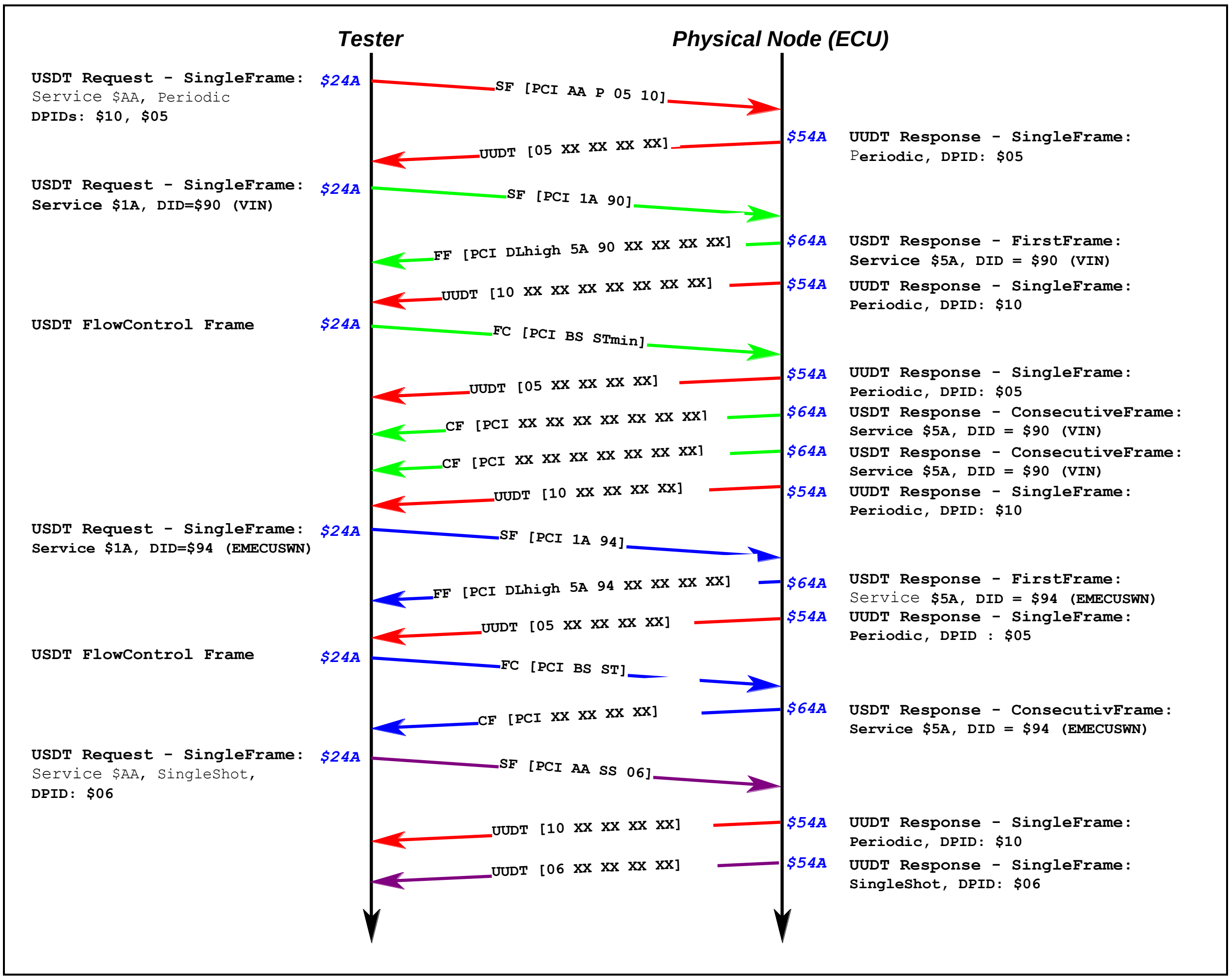

Tester
Physical Node (ECU)
USDT Request - SingleFrame:
Service $AA, Periodic
DPIDs: $10, $05
$24A
SF [PCI AA P 05 10]
$54A
UUDT [05 XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID: $05
USDT Request - SingleFrame:
Service $1A, DID=$90 (VIN)
$24A
SF [PCI 1A 90]
$64A
FF [PCI DLhigh 5A 90 XX XX XX XX]
USDT Response - FirstFrame:
Service $5A, DID = $90 (VIN)
$54A
UUDT [10 XX XX XX XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID: $10
USDT FlowControl Frame
$24A
FC [PCI BS STmin]
$54A
UUDT [05 XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID: $05
$64A
CF [PCI XX XX XX XX XX XX XX]
USDT Response - ConsecutiveFrame:
Service $5A, DID = $90 (VIN)
$64A
CF [PCI XX XX XX XX XX XX XX]
USDT Response - ConsecutiveFrame:
Service $5A, DID = $90 (VIN)
$54A
UUDT [10 XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID: $10
USDT Request - SingleFrame:
Service $1A, DID=$94 (EMECUSWN)
$24A
SF [PCI 1A 94]
$64A
FF [PCI DLhigh 5A 94 XX XX XX XX]
USDT Response - FirstFrame:
Service $5A, DID = $94 (EMECUSWN)
$54A
UUDT [05 XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID : $05
USDT FlowControl Frame
$24A
FC [PCI BS ST]
$64A
CF [PCI XX XX XX XX]
USDT Response - ConsecutivFrame:
Service $5A, DID = $94 (EMECUSWN)
USDT Request - SingleFrame:
Service $AA, SingleShot,
DPID: $06
$24A
SF [PCI AA SS 06]
$54A
UUDT [10 XX XX XX XX]
UUDT Response - SingleFrame:
Periodic, DPID: $10
$54A
UUDT [06 XX XX XX XX]
UUDT Response - SingleFrame:
SingleShot, DPID: $06

Figure 7: Interleaving of a Multiple Frame Message by Periodic/Event Driven Single Frame Messages

4.10 Functional System Assignments.

4.10.1 Definition of a Functional System. A functional system is comprised of a node, or group of nodes, which support a specific vehicle feature/functionality. If a node provides any kind of information which is necessary for a second or multiple node(s) to perform any kind of action, these nodes are a functional system. Also, if a node consumes any kind of information which is provided by another node or multiple other nodes these nodes belong to a functional system. These functional systems may also be indicated as Virtual Networks (VN) within the GMLAN system architecture. See Figure 8.

Example: An ECM provides the coolant temperature as a subfunction for the functional system A/C (Air Conditioning/Climate Control) to a second node which controls the rest of the A/C functional system. Both nodes are members of the functional system A/C (Air Conditioning/Climate Control) and must respond on a functional request message addressed to the functional system A/C.

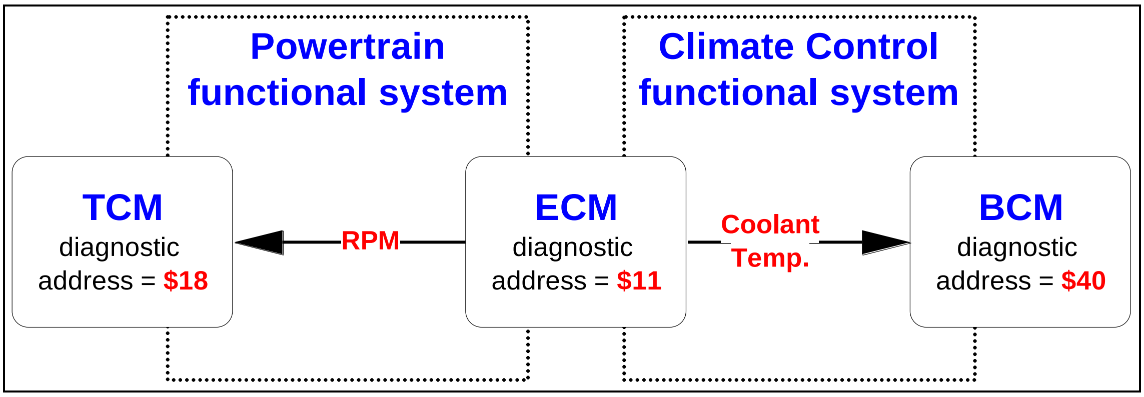

Powertrain functional system
Climate Control functional system
TCM diagnostic address = $18
RPM
ECM diagnostic address = $11
Coolant Temp.
BCM diagnostic address = $40

Figure 8: Functional Systems - Example

4.10.2 Functional System Table. Table 26 lists the standardized functional systems with their associated Extended Address (EA) for the GMLAN network.

Table 26: Functional Systems Address Assignments (Standardized Addresses)

Functional System Name	Functional System Address (EA)
Reserved	$00 thru $FC
Gateway Devices	$FD
All Functional Systems (AllNode)	$FE
Reserved	$FF

5 Diagnostics and Node Management

The following paragraphs describe the interaction between diagnostic services and Node Management. The Node Management strategy is described in the GMLAN Communication Strategy Specification (GMW3104).

Paragraph 5.1 describes the fundamental interaction between diagnostics and node management from the test tools perspective. The ECU shall meet the requirements in this section independent of the version of the strategy specification implemented in the kernel.

Paragraph 5.2 details an implementation based on version 1.5 of the GMLAN strategy specification (GMW3104).

5.1 Interaction between Diagnostics and Node Management. Each time a wake-up occurs (independent of the wake-up mechanism), a node shall transition to a communications state where it can receive, process, and respond to any diagnostic message.

Note: Reference the wake-up requirements section of this specification for information relative to the maximum time allowed after a wake-up until the node is capable of receiving, processing and responding to diagnostic messages.

A node shall remain capable of receiving, processing, and responding to diagnostic messages for a minimum of 8 s following the wake-up.

Note: This assumes the node is powered at the time of wake-up and does not lose power during the 8 s interval.

Once diagnostic communication has started (i.e., at least one diagnostic message has been received, or is in the process of being received within the first 8 s following a wake-up), the node shall remain capable of receiving, processing, and responding to diagnostic messages for a minimum of 8 s following the completion of the last diagnostic service.

Note: This assumes that the node remains powered. Refer to paragraph 5.2.4 for requirements on how to determine when a diagnostic service is completed. A node which is in the process of receiving a multi-frame diagnostic message shall be capable of receiving the complete message without the possibility of remoting off. This can be accomplished via the use of the network layer service primitives which provide first frame indications, error indications, and an indication of the reception of a complete message.

If Diagnostic Mode is activated during the course of diagnostic communication, a node shall be capable of receiving, processing and responding to diagnostic messages for a minimum of 8 s following the termination of Diagnostic Mode.

Note: Reference the Terms and Definitions section of this specification for a detailed definition of diagnostic mode.

A GMLAN node that does not support VNMFs shall be capable of receiving, processing, and responding to diagnostic messages any time that the node would otherwise be communications capable. If diagnostic functionality is required at a time when a node would not otherwise be communicating (and can be designed such that diagnostic communications can take place), then the circumstances under which diagnostics can be performed shall be documented in a CTS, SSTS or applicable diagnostic specification referenced by a CTS or SSTS.

5.2 Communication State Diagram Based On Version 1.5 of GMW3104. Figure 9 and Figure 10 were taken from GMW3104 to provide a visual aid in understanding the possible communication states that exist in a node. The COMM_INIT and COMM_KERNEL_ACTIVE states are the states where a node is either capable of communications, or in the process of initializing the hardware and software necessary to become capable of communications. A node which is not in the COMM_INIT or COMM_KERNEL_ACTIVE states is not capable of communications and must enter the COMM_INIT state (to initialize the communications hardware and software) prior to communicating with other nodes on the vehicle or an off-board tester.

The following sections of this specification describe what takes place in the COMM_INIT and COMM_KERNEL_ACTIVE states relative to diagnostics. For more detailed information on these states or the other states in the figures (reference GMW3104).

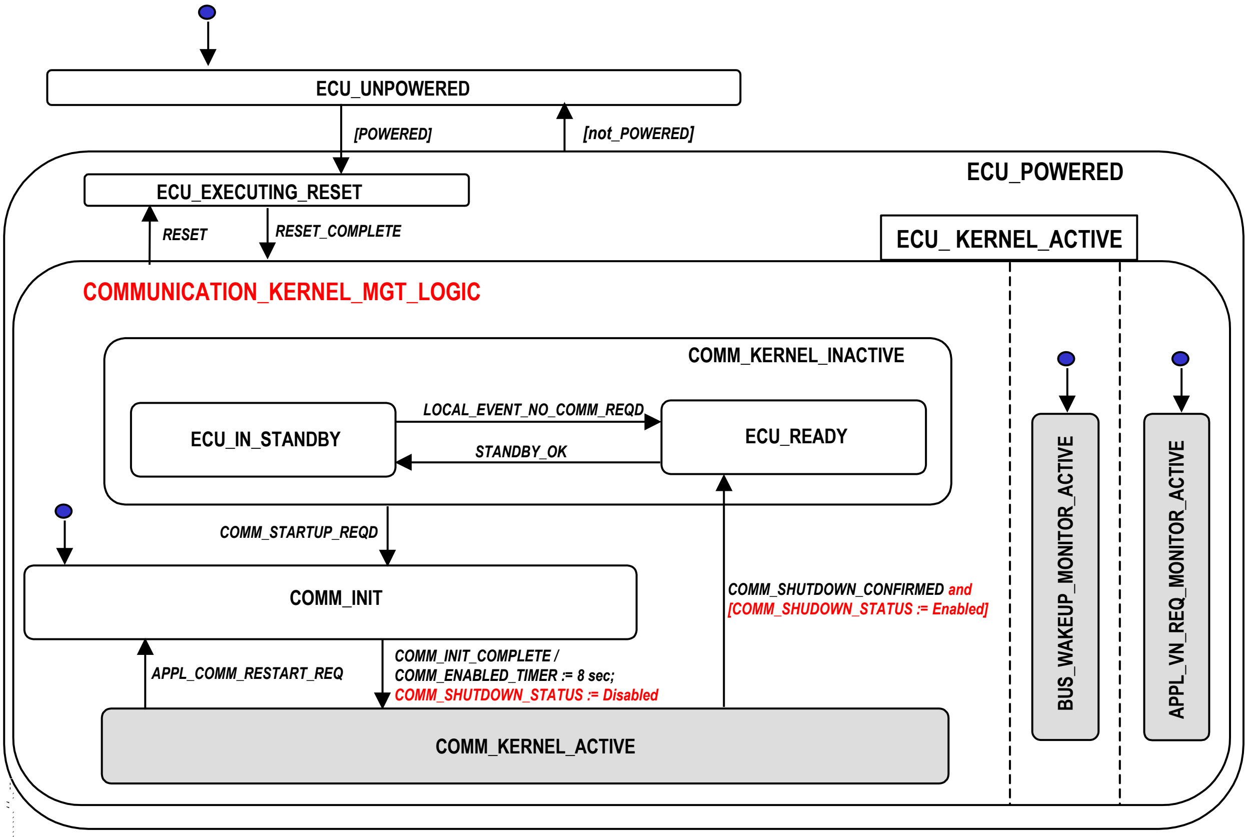


Figure 9: GMLAN Node Communication States

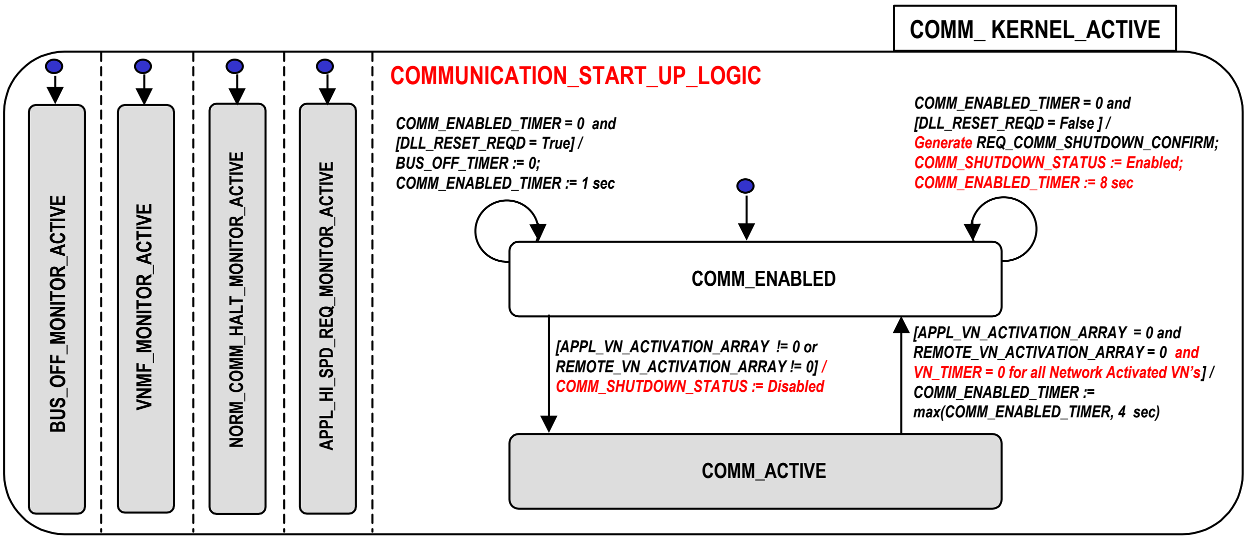

COMM_ KERNEL_ACTIVE
BUS_OFF_MONITOR_ACTIVE
VNMF_MONITOR_ACTIVE
NORM_COMM_HALT_MONITOR_ACTIVE
APPL_HI_SPD_REQ_MONITOR_ACTIVE
COMMUNICATION_START_UP_LOGIC
COMM_ENABLED_TIMER = 0 and [DLL_RESET_REQD = True] / BUS_OFF_TIMER := 0; COMM_ENABLED_TIMER := 1 sec
COMM_ENABLED_TIMER = 0 and [DLL_RESET_REQD = False] / Generate REQ_COMM_SHUTDOWN_CONFIRM; COMM_SHUTDOWN_STATUS := Enabled; COMM_ENABLED_TIMER := 8 sec
COMM_ENABLED
[APPL_VN_ACTIVATION_ARRAY != 0 or REMOTE_VN_ACTIVATION_ARRAY != 0] / COMM_SHUTDOWN_STATUS := Disabled
[APPL_VN_ACTIVATION_ARRAY = 0 and REMOTE_VN_ACTIVATION_ARRAY = 0 and VN_TIMER = 0 for all Network Activated VN's] / COMM_ENABLED_TIMER := max(COMM_ENABLED_TIMER, 4 sec)
COMM_ACTIVE

Figure 10: GMLAN Node Communication States - COMM_KERNEL_ACTIVE

5.2.1 Communications Initialization (COMM_INIT) State. This is an initialization state in which serial communications are not possible for diagnostics or normal communications. In this state a device initializes its CAN hardware and software variables used for communication. A node enters this state from either the ECU_READY or ECU_IN_STANDBY states if either of the following occurs:

1. Node control logic evaluates a local input and determines that communications with other nodes are required.
2. A bus wake-up is received. The wake-up may have been generated by a node on the vehicle or by an off-board tester.

A node enters this state from the COMM_ACTIVE state upon termination of the Normal Comm Halted handler service, and a node also enters this state following the completion of a software reset.

See GMW3104 and the GMLAN Handler specification (GMW3107) for more detail.

5.2.2 Communications Enabled (COMM_ENABLED) State. The state of a node where all communication capabilities have been established. In the Communication Enabled state a node is capable of receiving and processing Virtual Network Management Frames (VNMF) (see GMW3104 for more information on VNMF) and receiving bus wake-ups. Diagnostic messages cannot be sent or received by a node in this state. See the COMM_ACTIVE state description section for further definition of diagnostic messages.

A node remains in the **Communications Enabled** state until a Virtual Network (VN) that it supports becomes active, or until a time period elapses where no VNs have been activated.

Note: Refer to GMW3104 for more information on Virtual Network Management Frames.

If a VN that a node supports becomes active, then the device transitions to the **Communications Active** state. A node transitions to the **ECU Ready** state if a timeout occurs.

Note: GMW3104 also specifies additional requirements of the application before an ECU can transition to the ECU_READY state.

5.2.3 Communications Active (COMM_ACTIVE) State. The state of a node when it is participating in at least one active VN. The node may have either internally decided to activate a VN or it may have received a Virtual Network Management Frame (VNMF) with an activation request for a VN the node supports. The Communication Active state is entered before any frame can be transmitted.

Note: Refer to GMW3104 for more information on Virtual Network Management Frames.

A node remains in this state as long as it has at least one active VN. This VN can be an initially active VN, an application triggered VN, or a bus activated VN. Once a node has no VNs active, it can transition to the **Communication Enabled** state.

A node shall be capable of receiving, processing, and responding to diagnostic messages in this state any time the **initially active diagnostic VN** is active or the **application triggered diagnostic VN** is active.

5.2.3.1 Initially Active Diagnostic VN. GMW3104 defines initially active VNs as Virtual Networks which are automatically activated any time a bus wake-up is received. Initially active VNs remain active for a minimum of 8 s following the wake-up, or 8 s after the transmission/reception of a VNMF with the bit set in the VNMF corresponding to the initially active VN.

GMW3104 defines an initially active diagnostic VN. When the initially active diagnostic VN is active, the communications kernel enables the use of all diagnostic CAN Identifiers.

Diagnostic CAN Identifiers include the following:

1. A node's predefined diagnostic point-to-point request CAN Identifier (physical request CAN Identifier).
2. The **AllNodes** request CAN Identifier (functional request CAN Identifier) with a valid extended address (functional system address). See the section in this specification on diagnostic CAN Identifiers for more information on the **All Nodes** request CAN Identifier and the use of extended addresses.
3. A node's reserved SPS_PrimeReq CAN Identifier (this CAN Identifier is only valid for a SPS_TYPE_C ECU during a programming event. See the programming section of this specification for more details on the use of this type of CAN Identifier).
4. A valid OBD/EOBD request CAN Identifier.

5.2.3.2 Application Triggered Diagnostic VN. To ensure that a node remains in the COMM_ACTIVE state while a tester is performing diagnostic services, all nodes (which support the diagnostic services described in this specification) shall support an application triggered diagnostic VN.

The application triggered diagnostic VN shall be activated and deactivated by the node's diagnostic application layer. Since the application triggered diagnostic VN is local to each node, no Virtual Network Management Frame (VNMF) activation bit shall be transmitted in support of this VN.

The Application Layer shall use the service primitives provided by the Network Layer and additional self-generated flags to determine if an activation or deactivation of the application triggered diagnostic VN is required.

5.2.4 Enabling Diagnostic Communications. To begin performing diagnostic services, a tool sends out a wake-up request (for SWCAN this is the high voltage wake-up with CANId $100, for wake-up mechanisms on other GMLAN links, the tool uses the InitiateDiagnosticOperation service sub-function $04 - wakeUpLinks targeted to all gateway devices using a functionally addressed request with the Gateway Devices - $FD extended address). The wake-up causes ECUs to transition into (or remain in) the COMM_ACTIVE state. An ECU which was previously in the COMM_KERNEL_INACTIVE state (see Figure 9) would transition through the COMM_INIT state and through the COMM_ENABLED state (due to the initially active diagnostic VN) and end up in the COMM_ACTIVE state. The diagnostic CAN Identifiers are enabled in the kernel as long as the initially active diagnostic VN is active (8 s).

While the initially active diagnostic VN is active, the tester can activate the application triggered diagnostic VN in order to keep the ECU in the COMM_ACTIVE state while it is performing diagnostics.

Figure 11 contains a state transition diagram of the diagnostic application. The following definitions are provided to aid in understanding the requirements for transitioning into and out of the various states:

Start of Diagnostic Request:

The start of a diagnostic request is defined as reception of either the FirstFrame of a diagnostic request (as indicated by the Network Layer), or the reception of a SingleFrame diagnostic request message using the defined diagnostic CAN Identifiers.

Diagnostic Service Finished (successful or with error): A diagnostic service is considered finished (or completed) if any of the following occurs:

1. The application receives an error indication from the network layer after the reception of a multi-frame message had started.
2. A request message is successfully received, the application has finished processing the request, and no response is required.
3. A request is received which requires a response, the application has processed the request and

- The application receives confirmation from the Network Layer (or Data Link Layer for UUDT responses) that the complete diagnostic response message (either USDT single frame, the last frame of a USDT multiple frame response, or the last UUDT response message) has been successfully transmitted on the bus, **or**

Note: Certain diagnostic services such as the $A9 service may send multiple UUDT responses to a single request. The service is not considered completed in this case until all of the UUDT responses have been confirmed or an error is indicated.

- The application receives an indication from the Network Layer (or Data Link Layer for UUDT responses) that an error occurred while transmitting the diagnostic response message on the bus.

Note: A functionally performed TesterPresent service is finished after the reception and the processing of the request message, because there is no response to a functionally addressed TesterPresent request.

The next several sections describe the required behavior in each of the states.

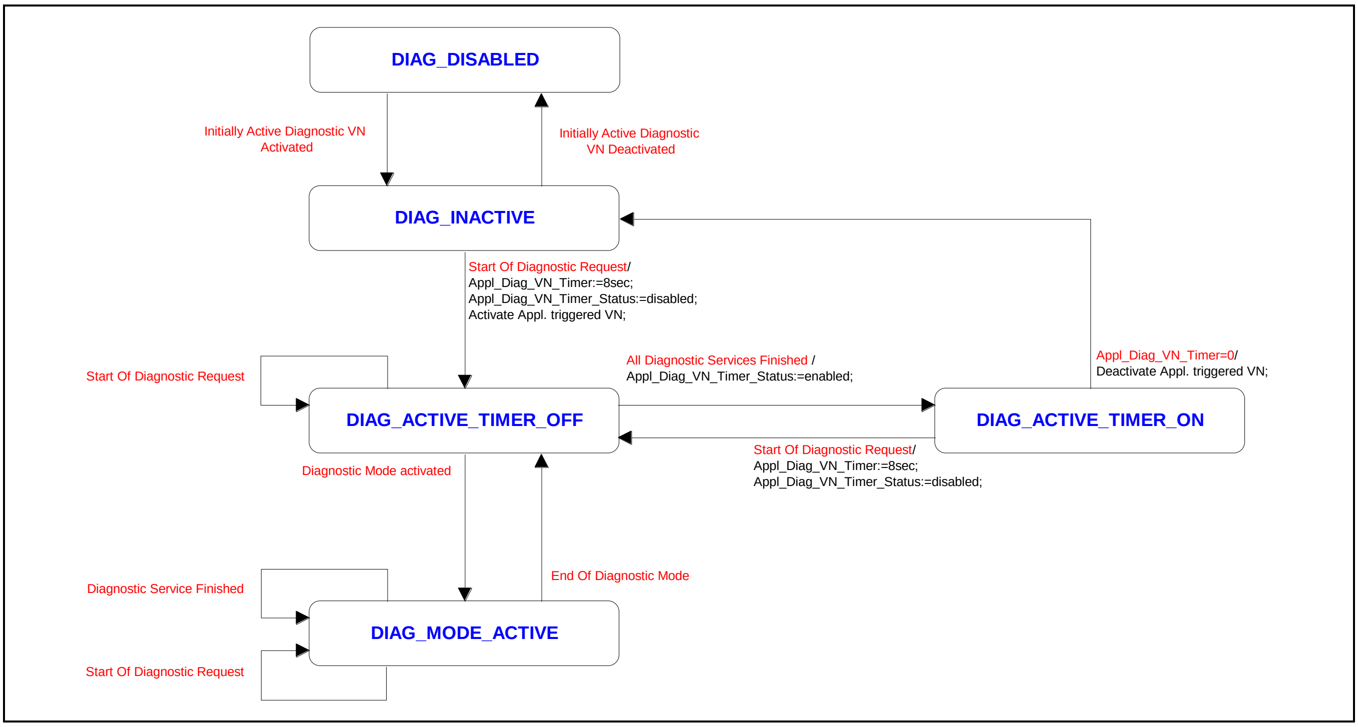

DIAG_DISABLED
Initially Active Diagnostic VN Activated
Initially Active Diagnostic VN Deactivated
DIAG_INACTIVE
Start Of Diagnostic Request/
Appl_Diag_VN_Timer:=8sec;
Appl_Diag_VN_Timer_Status:=disabled;
Activate Appl. triggered VN;
Start Of Diagnostic Request
All Diagnostic Services Finished /
Appl_Diag_VN_Timer_Status:=enabled;
Appl_Diag_VN_Timer=0/
Deactivate Appl. triggered VN;
DIAG_ACTIVE_TIMER_OFF
DIAG_ACTIVE_TIMER_ON
Start Of Diagnostic Request/
Appl_Diag_VN_Timer:=8sec;
Appl_Diag_VN_Timer_Status:=disabled;
Diagnostic Mode activated
End Of Diagnostic Mode
Diagnostic Service Finished
DIAG_MODE_ACTIVE
Start Of Diagnostic Request

Figure 11: Diagnostic Application State Diagram

5.2.4.1 DIAG_DISABLED State. This is the state of the diagnostic application when the diagnostic CAN Identifiers are not enabled and no diagnostics can take place. The diagnostic application exits this state and enters the DIAG_INACTIVE state once a tester or a node on the link issues a wake-up which enables the initially active diagnostic VN.

5.2.4.2 DIAG_INACTIVE State. In the DIAG_INACTIVE state, the ECU shall be capable of receiving diagnostic messages but no diagnostic requests are currently active (i.e., the Application triggered diagnostic VN is not active). The diagnostic application shall enter this state from the DIAG_DISABLED state after a bus wake-up is received. The diagnostic applications shall remain in this state while the initially active diagnostic VN is active and the application triggered diagnostic VN is inactive.

5.2.4.3 DIAG_ACTIVE_TIMER_OFF State. The diagnostic application shall enter this state from the DIAG_INACTIVE state or the DIAG_ACTIVE_TIMER_ON state upon notification from the Network Layer that a diagnostic request has started (Start of Diagnostic Request). A node shall enter this state from the DIAG_MODE_ACTIVE state upon termination of Diagnostic Mode. If the diagnostic application is in the DIAG_INACTIVE state when a diagnostic request starts, the diagnostic application shall activate the application triggered diagnostic VN and reset and disable a diagnostic application based VN timer (further referenced as Appl_Diag_VN_Timer). If the diagnostic application is in the DIAG_ACTIVE_TIMER_ON state when the diagnostic request starts, then the diagnostic application shall reset and disable its Appl_Diag_VN_Timer.

Note: "Reset" in this case means modification of the value of the timer such that 8 additional seconds shall elapse before a timeout occurs.

Note: "Disable" in this case means preventing the value of the timer from being modified during subsequent processing loops.

The diagnostic application exits this state and enters the DIAG_MODE_ACTIVE state if a diagnostic request is received which activates Diagnostic Mode. The diagnostic application enters the DIAG_ACTIVE_TIMER_ON state from this state once the last diagnostic service has finished. When transitioning to the DIAG_ACTIVE_TIMER_ON state, the Appl_Diag_VN_Timer is enabled.

Note: "Enable" in this case means allowing the value of the timer to be modified during subsequent processing loops.

5.2.4.4 DIAG_ACTIVE_TIMER_ON State. The diagnostic application shall enter this state from the DIAG_ACTIVE_TIMER_OFF state once the last diagnostic service has finished. The diagnostic application shall remain in this state until either a new diagnostic request is started (which transitions the application back to the DIAG_ACTIVE_TIMER_OFF state) or the Appl_Diag_VN_Timer times out. The diagnostic application shall deactivate the application triggered diagnostic VN and transition to the DIAG_INACTIVE state once the Appl_Diag_VN_Timer times out.

5.2.4.5 DIAG_MODE_ACTIVE State. The diagnostic application shall enter this state anytime Diagnostic Mode is active and shall remain in this state as long as Diagnostic Mode remains active. The Appl_Diag_VN_Timer shall remain disabled while in this state. Upon termination of Diagnostic Mode, the diagnostic application shall transition to the DIAG_ACTIVE_TIMER_OFF state.

5.2.5 Activation of the Application Triggered Diagnostic VN. The application triggered diagnostic VN is activated when the diagnostic application is in the DIAG_INACTIVE state and a diagnostic request is started. Once in the DIAG_INACTIVE state, the tester can send a diagnostic request which is made up of one or multiple frames. Upon receipt of each SingleFrame (SF) diagnostic request, or the FirstFrame (FF) of a multi-frame diagnostic request, the network layer shall provide an indication to the diagnostic application. When the diagnostic application is in the DIAG_INACTIVE state and a SF or FF indication is received from the network layer, the application shall notify the GMLAN Handler that the application triggered diagnostic VN is requested to be active.

Figure 12 illustrates how the application triggered diagnostic VN is activated based on receipt of a FirstFrame of a diagnostic request. In the figure, a wake-up (shown as WuP) is generated by the tester when it is first plugged in to ensure that all nodes transition to (or remain in) the COMM_ACTIVE state (refer to Figure 10) and that the diagnostic application transitions into (or remains in) the DIAG_INACTIVE state. Upon receipt of the FirstFrame indication (N_USData_FF.ind) from the Network Layer, the diagnostic application tells the handler to activate the application triggered diagnostic VN.

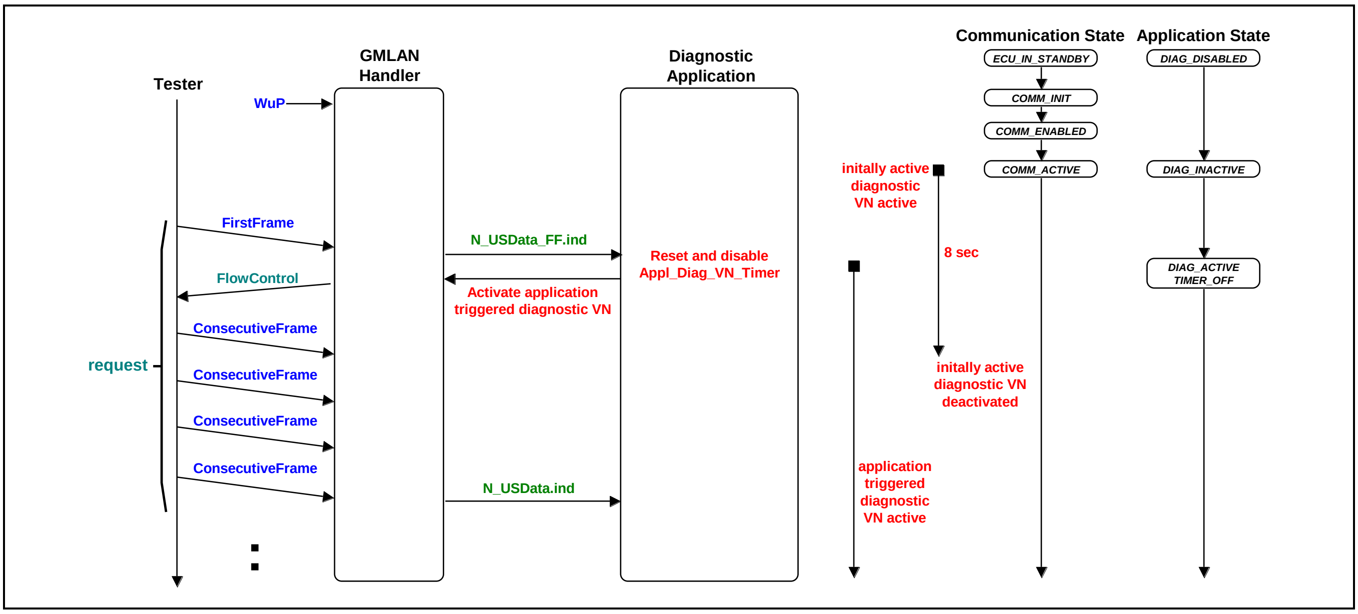


Figure 12: Application Triggered Diagnostic VN - Activation via FirstFrame Indication

Note: In Figure 12, it is assumed the node is in the ECU_IN_STANDBY state at the time the Tester initiates the WuP.

5.2.6 Keeping the Application Triggered Diagnostic VN Active. The diagnostic application shall keep the application triggered diagnostic VN active as long as the Appl_Diag_VN_Timer has not expired. (Refer to paragraph 5.2.4 for the detailed handling of the Appl_Diag_VN_Timer - reset, disable and enable conditions.)

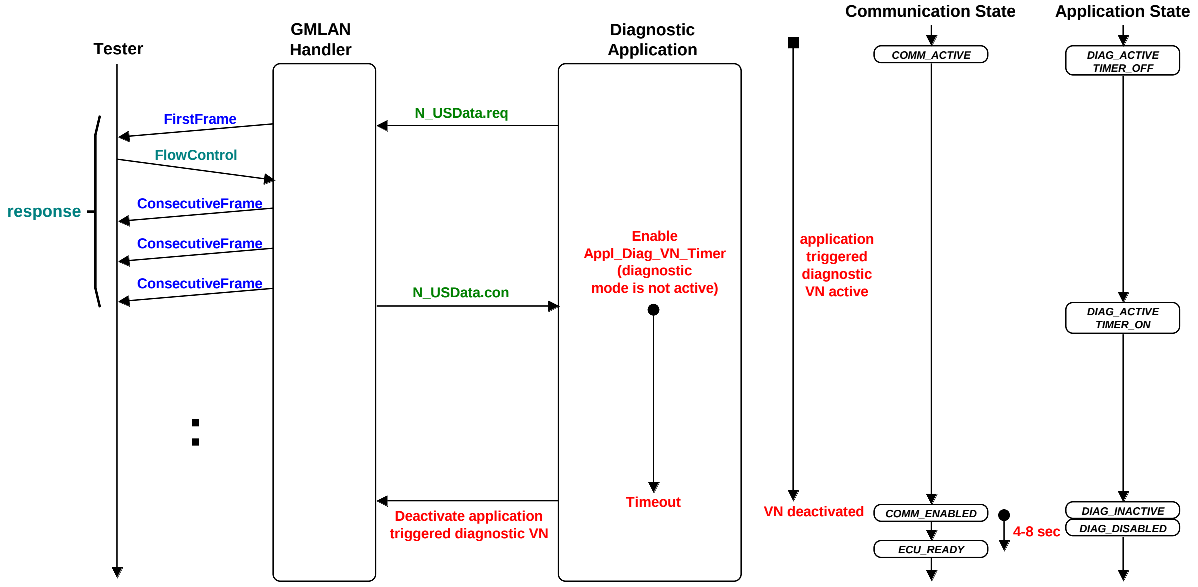
5.2.7 Deactivation of the Application Triggered Diagnostic VN. The diagnostic application shall notify the GMLAN Handler to deactivate application triggered diagnostic VN when the Appl_Diag_VN_Timer elapses. At this time the diagnostic application shall transition to the DIAG_INACTIVE state.

Figure 13 shows a case where diagnostic mode is not active and the ECU is sending a multiple frame response to a received request.

Note: It is assumed that the application triggered diagnostic VN is the only active VN.

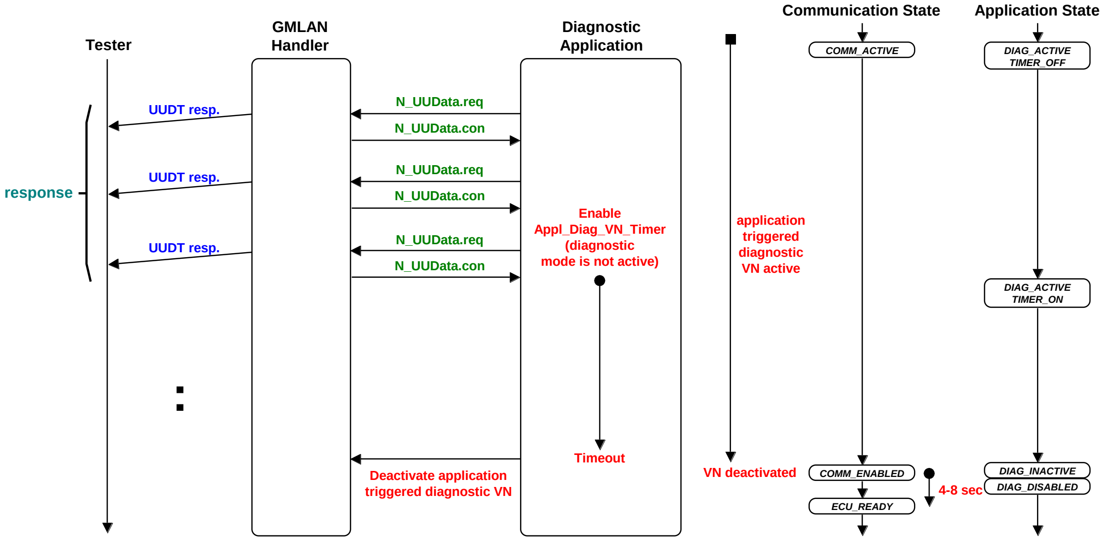
Figure 14 shows a case where diagnostic mode is not active and the requested diagnostic service results in 3 UUDT responses.

Note: It is assumed that the application triggered diagnostic VN is the only active VN.



Note: It is assumed that the application triggered diagnostic VN is the only active VN so when the diagnostic application deactivates it, the ECUs communication kernel transitions to COMM_ENABLED.

Figure 13: Application Triggered Diagnostic VN - Deactivation after Transmission of USDTResponse and Diagnostic Mode Is Not Active



Note: It is assumed that the application triggered diagnostic VN is the only active VN.

Figure 14: Application Triggered Diagnostic VN - Deactivation after Transmission Of Multiple UUDT Responses Without Diagnostic Mode Active

5.2.8 Verification of Diagnostic VN Deactivation. This section specifies the method on how to check whether a node correctly deactivates the initially active and the application triggered diagnostic VN. It takes into account the definitions in the previous sections.

5.2.8.1 Deactivation Verification of the Initially Active Diagnostic VN.

Test Procedure:

Perform a bus wake-up, which activates the initially active diagnostic VN and all other initially active Virtual Networks in the node.

Wait for all initially active Virtual Networks to timeout (8 s + 1 s safety margin after the bus wake-up). Make sure that no subsequent bus wake-up occurs.

Transmit a physically addressed ReadDataByIdentifier ($1A) service with DataIdentifier $B0 (diagnosticAddress) to the node using the node’s physical request CAN Identifier and make sure that the node does not respond to this diagnostic request.

Acceptance Criteria:

The node shall not respond to the last transmitted ReadDataByIdentifier ($1A) service. This indicates that the node has deactivated the initially active diagnostic VN properly.

5.2.8.2 Deactivation Verification of the Application Triggered Diagnostic VN.

Test Procedure:

a. Perform a bus wake-up, which activates the initially active diagnostic VN and all other initially active Virtual Networks in the node.

b. Wait for a duration of 7 s (less than the VN timeout value) and transmit a physically addressed ReadDataByIdentifier ($1A) service with DataIdentifier $B0 (diagnosticAddress) to the node using the node’s physical request CAN Identifier. Make sure that the node responds with the appropriate response message. This diagnostic request shall have caused the node to start the application triggered diagnostic VN and the node shall remain active for additional 8 s.

c. Wait another duration of 7 s (less than the VN timeout value) and again transmit a physically addressed ReadDataByIdentifier ($1A) service with DataIdentifier $B0 (diagnosticAddress) to the node using the node’s physical request CAN Identifier. Make sure that the node responds with the appropriate response message, which indicates that the node had activated the application triggered diagnostic VN with the previous diagnostic request. This second diagnostic request shall have caused the node to reset the application triggered diagnostic VN timer and the node shall remain active for additional 8 s.

d. Wait for the VN to timeout (8 s + 1 s safety margin after the reception of the response from the previous diagnostic request). Make sure that no subsequent bus wake-up occurs.

e. Transmit a physically addressed ReadDataByIdentifier ($1A) service with DataIdentifier $B0 (diagnosticAddress) to the node using the node’s physical request CAN Identifier and make sure that the node does not respond to this diagnostic request.

Note: ECUs implementing a communications handler based on versions of GMW3104 prior to version 1.5 may have an additional timer in the handler for deactivating local VNs. This timer value must be added to the 8 s specified in this verification procedure above.

Acceptance Criteria:

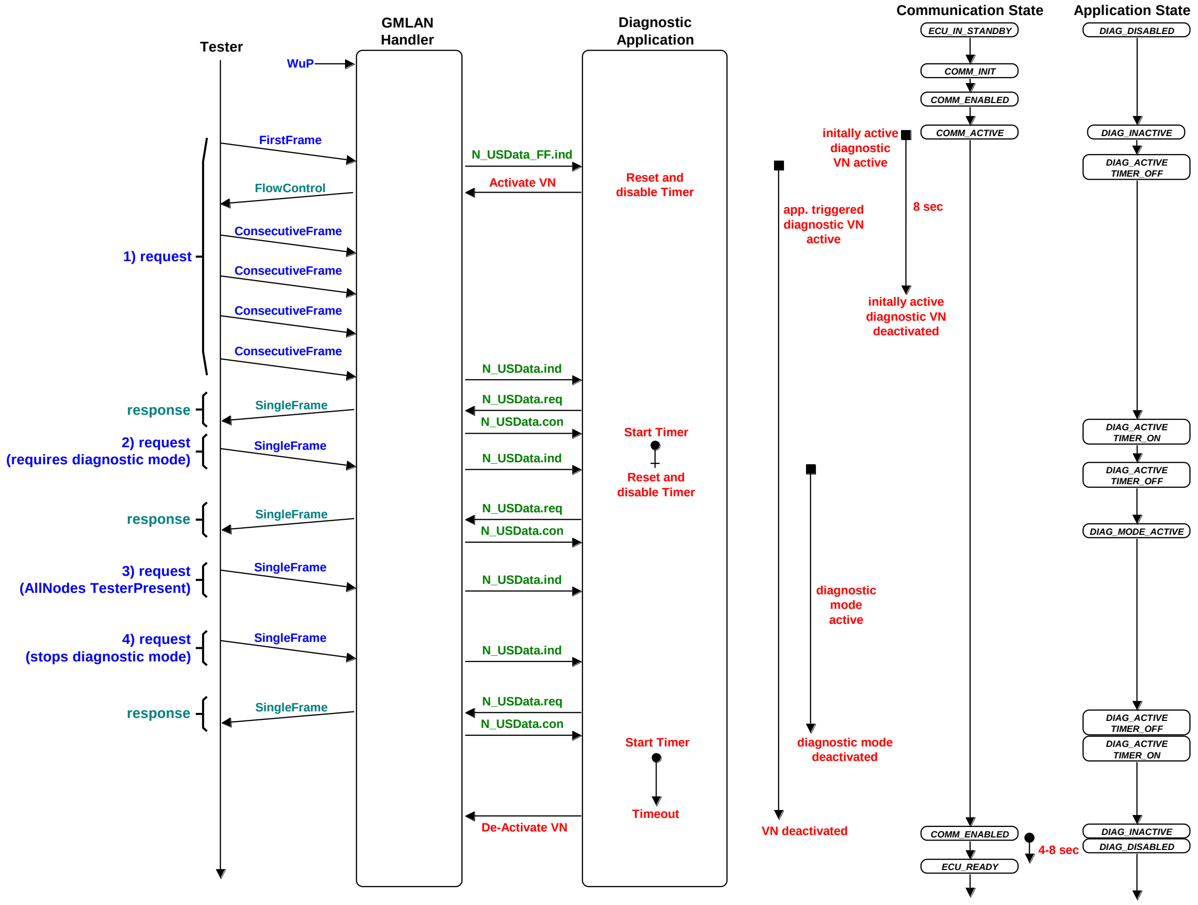
The node shall not respond to the last transmitted ReadDataByIdentifier ($1A) service. This indicates that the node has deactivated the initially active and application triggered diagnostic VN properly.

5.2.9 Example: Activation and Deactivation of the Application Triggered Diagnostic VN.

Note: It is assumed that the Application Triggered Diagnostic VN is the only Active VN.

Figure 15 provides an additional example for the activation and deactivation of the application triggered diagnostic VN. In the examples, it is assumed that the ECU is in the ECU_IN_STANDBY communication state at the time the tester sends a wake-up. Following the wake-up, the diagnostic application enters the DIAG_INACTIVE state.

1. The first request (1) is a multiple frame request message. Upon receipt of the FirstFrame, the Network Layer provides an indication to the diagnostic application that a request has started. The diagnostic application resets and disables the Appl_Diag_VN timer, tells the handler to activate the application triggered diagnostic VN, and transitions into the DIAG_ACTIVE_TIMER_OFF state. At the end of the SingleFrame response message the diagnostic application enables the Appl_Diag_VN_Timer and transitions to the DIAG_ACTIVE_TIMER_ON state (since the requested service does not require Diagnostic Mode and no response or request message is in progress).
2. The second request (2) is a SingleFrame request message that is received while the application triggered diagnostic VN is already active. The Network Layer indicates to the diagnostic application that the SingleFrame request has been received and the application resets and disables the Appl_Diag_VN_Timer (transitions back to DIAG_ACTIVE_TIMER_OFF state). The requested service requires the node to enable Diagnostic Mode and therefore the application layer transitions into the DIAG_MODE_ACTIVE state and does not enable Appl_Diag_VN_Timer after the transmission of the response message.
3. The third request (3) is a functionally addressed SingleFrame TesterPresent message which results in the node resetting the TesterPresent timer ($P3_C$) to keep Diagnostic Mode active. The diagnostic application stays in the DIAG_MODE_ACTIVE state.
4. The fourth request (4) is a SingleFrame message for the ReturnToNormalMode ($20) service. Upon receipt of the request, the Network Layer provides an indication to the diagnostic application that a diagnostic service has started. The application then processes the $20 request which cancels Diagnostic Mode and thus causes the ECU to transition into the DIAG_ACTIVE_TIMER_OFF state. Upon receiving confirmation from the network layer that the response was successfully transmitted, the diagnostic application transitions to the DIAG_ACTIVE_TIMER_ON state. After 8 s elapses, the application informs the handler to deactivate the application triggered diagnostic VN and transitions into the DIAG_INACTIVE state. Since the initially active diagnostic VN is already deactivated at this point, the diagnostic application then transitions into the DIAG_DISABLED state.



Note: It is assumed that the Application Triggered Diagnostic VN is the only active VN.

Figure 15: Diagnostic Local VN - Activation and Deactivation Example

6 Wake-up Requirements and Timing Parameters

6.1 Wake-up Requirements. The GMLAN communication strategy supports node communication shut down and also supports the possibility for a node to enter a low power state when its functionality is not needed. If a diagnostic tester wishes to communicate with a node which is in a low power state, or a state where communications capabilities have not been established (see note below), then the tester must first issue a wake-up. The wake-up mechanism for the Single Wire CAN (SWCAN) low speed link is the High Voltage Wake-up message. The allowable methods for performing a wake-up on the dual wire CAN links are defined in GMW3104 and include:

Note: See section on diagnostics and node management for more information about a nodes communication states.

Note: See the GMLAN Communication Strategy Specification (GMW 3104) for more information on the High Voltage Wake-up message.

1. Use of a wake-up wire.
2. Wake-up on bus activity.
3. Wake-up on discrete Accessory or Ignition power mode information.

The GMLAN communication strategy does not require gateways to cascade wake-up requests to all subnets that it supports. The strategy also allows for wake-up mechanisms which are not accessible to a tester (e.g., use of a wake-up wire). The InitiateDiagnosticOperation ($10) service provides a mechanism for a tester to request a gateway to generate a wake-up on all GMLAN subnets that it connects to where at least one Virtual Network is supported.

Note: The tester may have to first wake-up other subnets before being able to have a wake-up issued on the subnet that diagnostics are to be performed. For example, the tester may need to send the high voltage wake-up on the LS CAN link and then send the appropriate InitiateDiagnosticOperation request to the gateway between the LS and MS CAN links in order wake-up devices on the MS CAN bus.

GMLAN diagnostic tools shall allow **500 ms** for all nodes on a subnet to reach the Communication Active state after the tool issues a wake-up request. (See notes below.) This means that vehicle serial data designers must determine the worst case initialization time after a wake-up for all nodes on a given subnet, and factor that time in when determining the latency requirements for a gateway to generate wake-ups on the other subnets. In other words, if the worst case initialization time for all ECUs on a subnet was 350 ms, then the gateway would be required to transfer the wake-up across the subnet in less than 150 ms. (See note below.)

Note: Refer to the Diagnostics and Node Management chapter within this specification and the GMLAN Communication Strategy Specification (GMW 3104) for more information on the Communications Enabled state.

Note: The wake-up request may consist of sending the LS CAN high voltage wake-up message or sending an InitiateDiagnosticOperation request to a gateway.

Note: The 150 ms also takes into account worst case message traffic.

6.2 Application Timing Parameters Definitions. The following sections define the GMLAN Diagnostic Application timing parameters to be fulfilled by GMLAN ECUs and Testers. Each timing parameter definition is separated into two parts, except for the P3$_C$ application timing parameter. See paragraph 6.2.4.

- **ECU requirements:** This part describes the requirements for a single ECU communicating with a tester. Each timing parameters contains an **E** to indicate that it applies to a GMLAN **E**CU.
- **Tester requirements:** This part describes the requirements for the tester when communicating with one or multiple ECUs. Each timing parameters contains a **T** to indicate that it applies to a GMLAN **T**ester.

6.2.1 Application Timing Parameter P2$_{CE}$/P2$_{CT}$ Definition. The P2$_{CX}$ (see note below) application timing parameter applies to all diagnostic services defined in this specification. Each service uses a request/response scheme based on the following:

Note: “x” can either be “**E**” for **E**CU, or “**T**” for **T**ester.

- The **request message** is always a USDT single frame or multi-frame message **and**
- The **response message** following the request message is either:
- A USDT single frame or multi-frame message positive response (except for diagnostic services $AA and $A9), or a USDT single frame negative response.
- One or multiple diagnostic UUDT positive response message(s) for services $AA or $A9.

Note: If the initial response message is a negative response with response code $78 (RequestCorrectlyReceived-ResponsePending), then there shall also be a final response. Refer to paragraph 6.2.2 for additional information on the final response.

The following sections specify the timing requirements for the ECU and the tester based on the above described request/response schemes for GMLAN diagnostic services.

Further references to P2$_{\mathbf{C}}$ and P2$_{\mathbf{C}}$* within this specification are references to the ECU timing parameters P2$_{\mathbf{CE}}$ and P2$_{\mathbf{CE}}$* unless otherwise noted (e.g., P2$_{\mathbf{CT}}$ or P2$_{\mathbf{CT}}$*).

6.2.1.1 ECU Timing Parameter P2$_{CE}$ (Table 27). The timing parameter P2$_{CE}$ is defined to be the time between the end of the tester request message and the completion of either a single frame response, the first frame of a multi-frame response message, or the first UUDT message transmitted by the ECU in response to a service $AA or $A9.

Table 27: $P2_{CE}$ Timing Parameter Value

Parameter	Description	Minimum (CEmin) [ms]	Maximum (CEmax) [ms]
$P2_{CE}$	Time between tester request (N_USData.ind) and the ECU response as follows: The ECU has to make sure that a USDT single frame response message, the first frame of a multi-frame response message, or the first UUDT response message is completed (successfully transmitted onto the CAN bus) within $P2_{CE}$	0	100 Note 1

Note 1: OBD/EOBD devices are required to support the timing value $P2_{CE}$ as specified in ISO 15031-5 (referred to as $P2_{CAN}$ in ISO 15031-5) for emission related services. An OBD/EOBD device may choose to use a single value for $P2_{CE}$ for both emission related and enhanced diagnostic services provided that the value used is not greater than the value specified in this table.

The request message sent by the tester can either be a single frame (with address type either functional or physical), or a multi-frame message (with address type physical only). The end of a tester request message is indicated to the application layer in the ECU by the network layer (N_USData.ind).

Note: The network layer of a message sender only indicates the completion of an entire message (single frame/multi-frame USDT message, or UUDT message - see Figures 16 thru 18) to the application layer (N_USData.con, N_UUData.con). The network layer does not indicate the completion of the transmission of only the first frame of a multi-frame message.

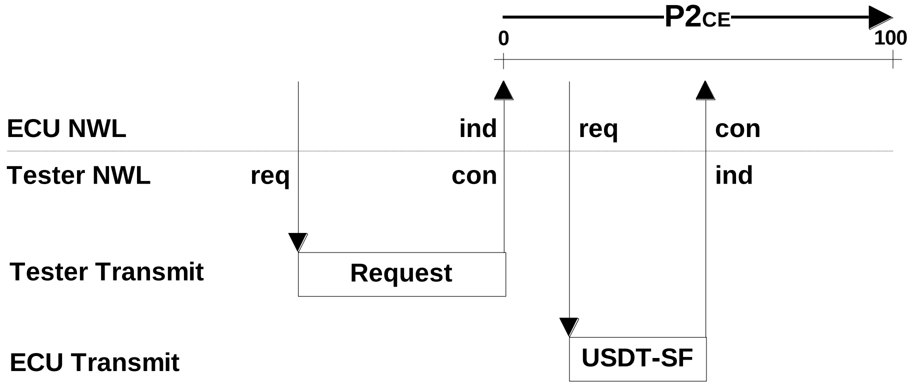


Figure 16: ECU $P2_{CE}$ Timing Parameter Definition - Single Frame Response Message

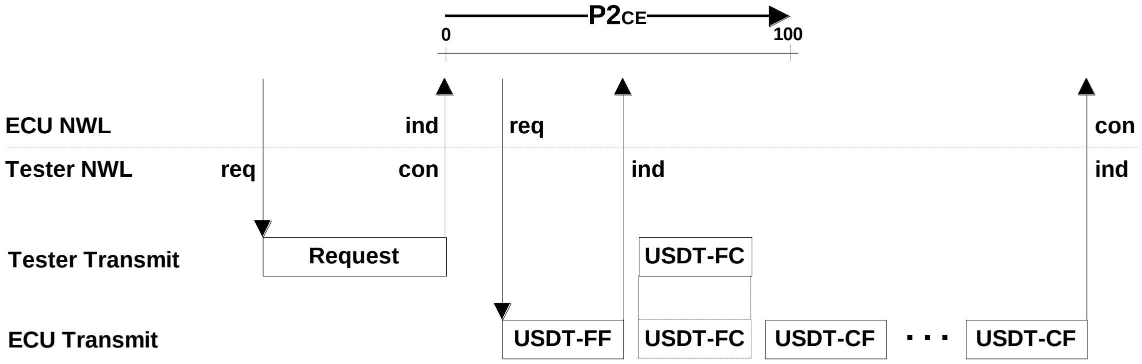


Figure 17: ECU $P2_C$ Timing Parameter Definition - Multiple Frame Response Message

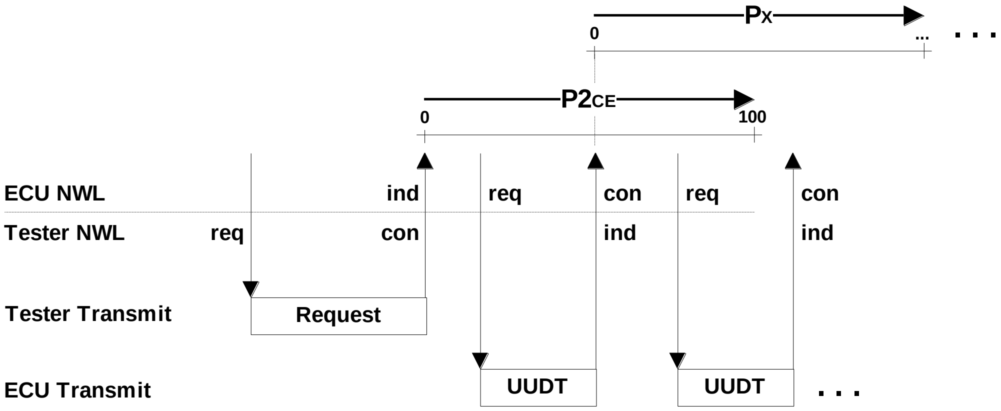


Figure 18: ECU P2$_C$ Timing Parameter Definition - UUDT Response Message(s)

Note: Figure 18 describes the timing parameter P2$_{CE}$ for services which send UUDT responses. Refer to diagnostic services $AA and $A9 for timing requirements (P$_X$) of UUDT responses following the initial UUDT response (if applicable).

The transmission and reception of USDT messages (single and multi-frame messages) is completely handled in the network layer.

When the application layer transmits a message, it sends a transmit request (N_USData.req) to the network layer and provides the data to be transmitted. The network layer performs the necessary timing handling to make sure that the data is transmitted (segmented data transfer, if data length requires multi-frames). The network layer indicates when the single frame or multi-frame message is completely transmitted (N_USData.con) or when an error occurred during the transmission (N_USData.ind with negative result code).

When receiving a multi-frame message, the application layer gets an indication from the network layer that a multi-frame message has started (N_USData_FF.ind) and gets another indication when the USDT multi-frame message is completely received (N_USData.ind) or when an error occurred (N_USData.ind with negative result code). Single frame messages are indicated to the application layer after its completion via a single indication (N_USData.ind with length ≤ 7 for normal addressing and length ≤ 6 for extended addressing).

The segmentation of data into multiple single frame UUDT response messages (diagnostic services $AA and $A9) is completely handled by the diagnostic application. The diagnostic application builds the individual UUDT messages and provides them to the network layer for transmission on the link (N_UUData.req). The network layer indicates when each individual UUDT message is completely transmitted (N_UUData.con).

6.2.1.2 Tester Timing Parameter P2$_{CT}$ (Table 28). For a GMLAN tester the timing parameter P2$_{CT}$ is defined to be:

- The timeout between the end of the tester request message (single frame or multi-frame request message) and the first received frame of either the USDT response message (single frame, or first frame of a multi-frame response), or the first UUDT message of the services $AA, $A9 (there may be subsequent UUDT messages transmitted by the ECU to transmit the requested data), **and**
- The timeout between each response message from multiple nodes responding during functional communication.

Table 28: Tester P2$_{CT}$ Timing Parameter Value

Parameter	Description	Minimum (CTmin) [ms]	Maximum (CTmax) [ms]
P2$_{CT}$	Timeout between tester request and ECU response or multiple ECU responses	150	N/A Note 1

Note 1: The maximum time that a tester waits for a response message is left for the discretion of the tester. The only requirement is that P2$_{CTmax}$ must be greater than P2$_{CTmin}$. Typically a tester would not wait beyond P3$_C$ or else certain active diagnostic services may time out.

The $P2_{CT}$ timer shall initially be started by the application layer of the tester after the successful transmission of the request message which is indicated by the network layer (N_USData.con). The $P2_{CT}$ timer shall be restarted by the application layer of the tester each time the network layer provides an indication, provided that the responses are associated with the corresponding request message for any of the following conditions:

- Receipt of a single frame USDT response message (N_USData.ind with length ≤ 7, response messages always use normal addressing).
- Receipt of the first frame of a multiple frame USDT response message (N_USData_FF.ind,).
- Receipt of the first (or only) UUDT response message from service $AA or $A9 response message (N_UUData.ind, only the first UUDT message of a service $AA or $A9 response message shall restart $P2_{CT}$).

If the tester does NOT know the number of responding ECUs to a functional request the following handling applies:

- The tester shall wait for a $P2_{CT}$ timeout, and
- For services other than $AA and $A9, all USDT multi-frame response messages must be completely received, and
- For services $AA and $A9, depending on the sub-level parameter value, the tester either has to wait for the completion of all expected UUDT response messages, or only the first UUDT response message (reference services $AA and $A9).

Note: If the tester sends a functionally addressed request message which results in either multiple ECUs sending multi-frame USDT responses or multiple single frame UUDT responses, then the tester must pick an appropriate value for $P2_{CTmax}$ which allows all ECUs to have sufficient time to gain access to the bus.

If the tester knows the number of responding ECUs the following handling applies:

The tester should not wait for a $P2_{CT}$ timeout, but

- For services other than $AA and $A9, all USDT multi-frame response messages must be completely received, and
- For services $AA and $A9, depending on the sub-level parameter value, the tester either has to wait for the completion of all expected UUDT response messages, or only the first UUDT response message (reference services $AA and $A9).

In both above listed cases the application layer has to keep track of the response messages following the request message.

The tester implementation of $P2_{CT}$ as described assumes that the tester has implemented the network layer and utilizes the service primitives that it provides. The network layer of the tester provides an indication to the application layer each time any of the following occurs:

- The first frame of a multi-frame response is received (N_USData_FF.ind).
- The last consecutive frame of a multi-frame response message is received (N_USData.ind).
- A SingleFrame response is received (N_USData.ind with length ≤ 7). See Figure 19.
- An error condition is detected by the network layer (N_USData.ind with a negative result code).
- A UUDT diagnostic response is received (N_UUData.ind). See Figure 21.

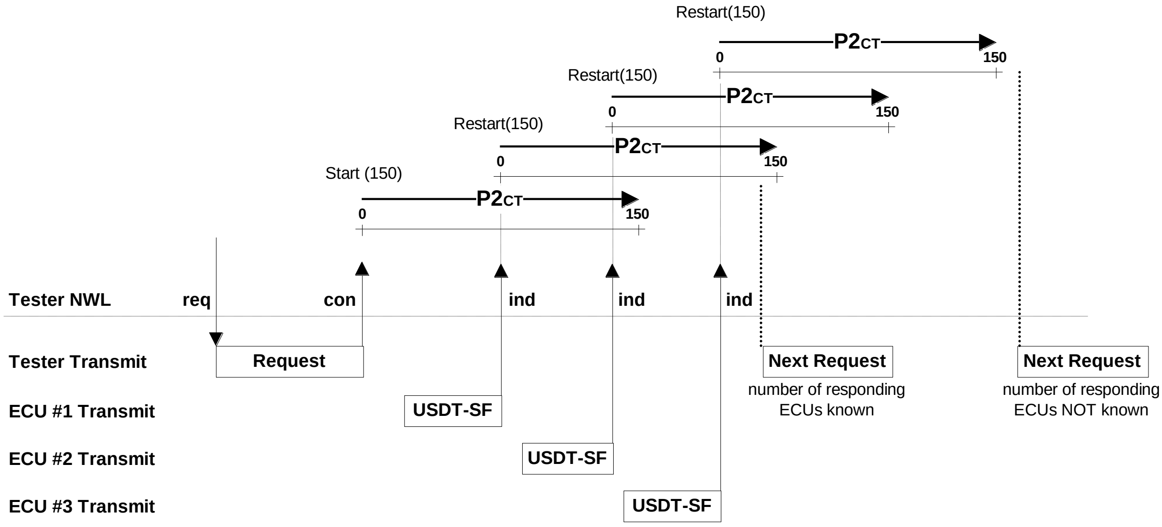


Figure 19: Tester $P2_C$ Timing Parameter Definition - Single Frame Response Messages

In Figure19, the tester transmits a functionally addressed request message (single frame message) and three ECUs respond. Each ECU responds with a single frame response message. If the tester knows the number of responding ECUs, then the next request message can be transmitted immediately after the reception of the last response message. If the number of responding ECUs is not known, then the tester shall wait until a $P2_{CT}$ timeout occurs before transmitting the next request message. See Figure 20.

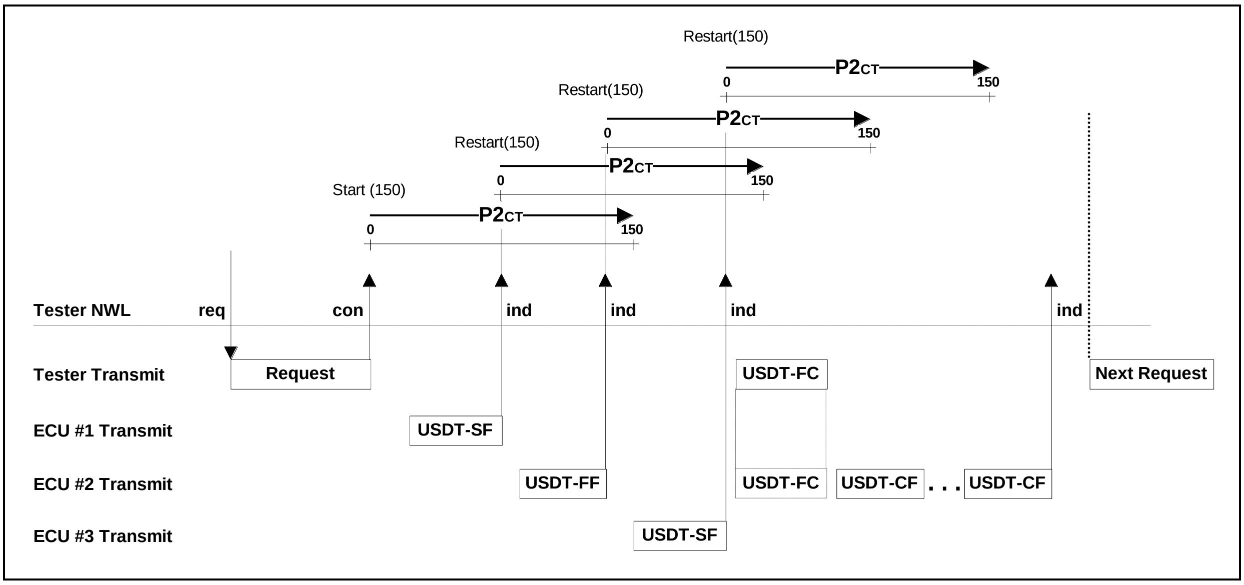


Figure 20: Tester $P2_{CT}$ Timing Parameter Definition - Single and Multi-Frame Response Messages

In Figure 20, the tester transmits a functionally addressed request message and three ECUs respond. Two ECUs respond with a single frame response message and one ECU responds with a multi-frame response message. The tester has to wait until each response message is completely received before transmitting the next request message.

If the $P2_{CT}$ application timer is not timed out at the point in time when the last pending multi-frame response message is completely received and the tester does not know the number of responding ECUs, then the tester has to wait for a timeout of $P2_{CT}$ before transmitting the next request message. If the tester knows the number of responding ECUs and all expected response messages are completely received then the tester can immediately transmit the next request message.

Note: The indication of a successful reception of a multi-frame response message does not restart the P2$_{\mathbf{CT}}$ application timer in the tester. Only the start of the multi-frame response message indicated via the first frame indication (N_USData_FF.ind) is relevant for the restart of the P2$_{\mathbf{CT}}$ timer.

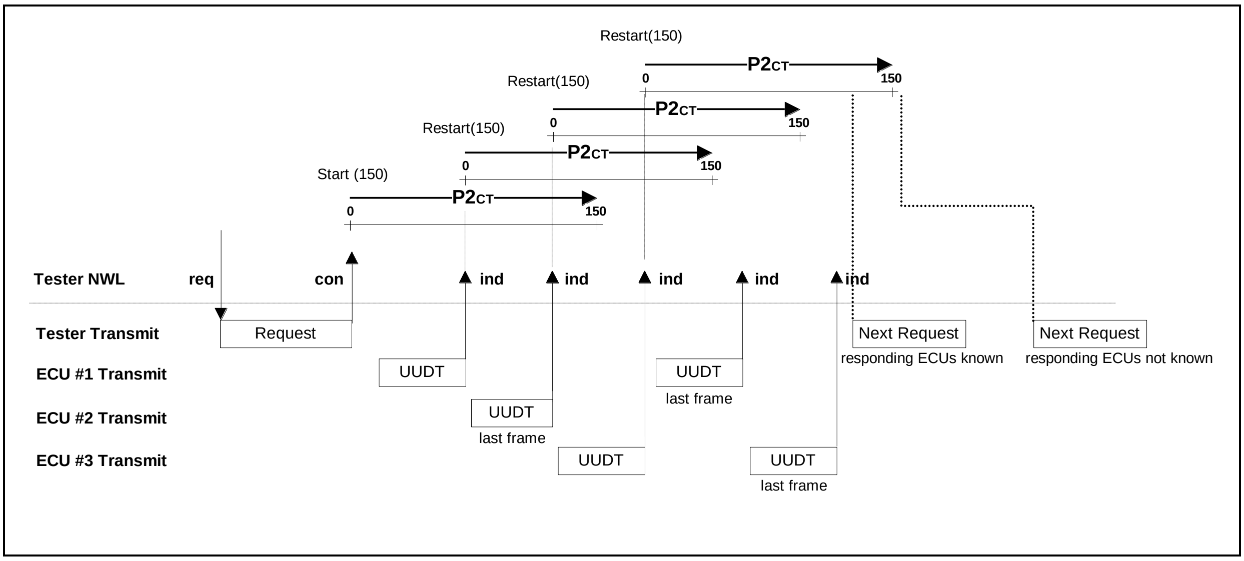


Figure 21: Tester P2$_{\mathbf{CT}}$ Timing Parameter Definition - Multiple UUDT Response Messages

Note: The timing handling of the tester that applies to subsequent UUDT messages, optionally following the first UUDT message is not shown in the figure above.

In Figure 21, the tester transmits a functionally addressed request message (e.g., $A9) and three ECUs respond. Two ECUs respond with two UUDT messages and one ECU responds with one UUDT message. In this example the tester either waits for a P2$_{CT}$ timeout and the completion of all UUDT response messages (number of responding ECUs not known), or only for the completion of all UUDT response messages (number of responding ECUs known).

Note: Only the successful reception of the first UUDT response message of each ECU following the request message restarts the P2$_{\mathbf{CT}}$ application timer in the tester.

6.2.2 Application Timing Parameter P2$_{\mathbf{CE}}$*/P2$_{\mathbf{CT}}$*Definition. If an ECU cannot respond with its final USDT response message or its first UUDT response message following a service $AA or $A9 request message within the required P2$_{CE}$ timing window as specified in paragraph 6.2.1.1, then the ECU can indicate to the tester that an enhancement of its response timing parameter to P2$_{CE}$* is required. This can be done by sending a negative response message with response code $78 (RequestCorrectlyReceived-ResponsePending) within the initial P2$_{CE}$ timing window as defined below. Upon receiving a negative response with response code $78, the tester shall modify its normal P2$_{CT}$ timing to the enhanced P2$_{CT}$* timing as defined below.

6.2.2.1 ECU Timing Parameter P2$_{\mathbf{CE}}$* (Table 29 and Figures 22 thru 25). If an ECU cannot respond with its final USDT response message or its first UUDT response message following a service $AA or $A9 request message within the required normal P2$_{CE}$ timing window as specified in paragraph 6.2.1.1, then it shall perform the following handling:

- The ECU shall respond with a negative response message (single frame message) with response code $78 - RequestCorrectlyReceived-ResponsePending within normal P2$_{\mathbf{CE}}$ timing as defined in paragraph 6.2.1.1. This forces the tester to set its P2$_{\mathbf{CT}}$ timing parameter value to P2$_{\mathbf{CT}}$*. See paragraph 6.2.2.2.
- If the ECU requires more than P2$_{\mathbf{CE}}$* to perform the requested action, then it shall repeat the negative response message with response code $78 prior to expiration of P2$_{\mathbf{CE}}$*. This shall continue until the ECU

is capable of sending the final USDT response (positive response or negative response with a response code other than $78) or the first UUDT response message.

- For its final USDT response message or UUDT response message the ECU has to make sure that the single frame response message (positive or negative response other than response code $78), the first frame of a multi-frame response message or the first UUDT response message is completed (transmitted on to the CAN bus) within the last requested $P2_{CE}$*.
- The ECU shall be able to receive and process a functionally addressed TesterPresent service request messages while it has activated the enhanced response timing and prepares the final response message.

Note: This functionality shall keep a node in a previously activated diagnostic mode of operation if the preparation of the final response message takes longer than $P3_C$.

Table 29: ECU $P2_{CE}$* Timing Parameter Value

Parameter	Description	Minimum (CEmin*) [ms]	Maximum (CEmax*) [ms]
$P2_{CE}$*	Enhanced response timing for a single ECU	0	5000 Note 1

Note 1: OBD/EOBD devices are required to support the timing value $P2_{CE*}$ as specified in ISO15031-5 (referred to as $P2*_{CAN}$ in ISO 15031-5) for emission related services. An OBD/EOBD device may choose to use a single value for $P2*_{CE}$ for both emission related and enhanced diagnostic services provided that the maximum value used is not greater than the value specified in this table.

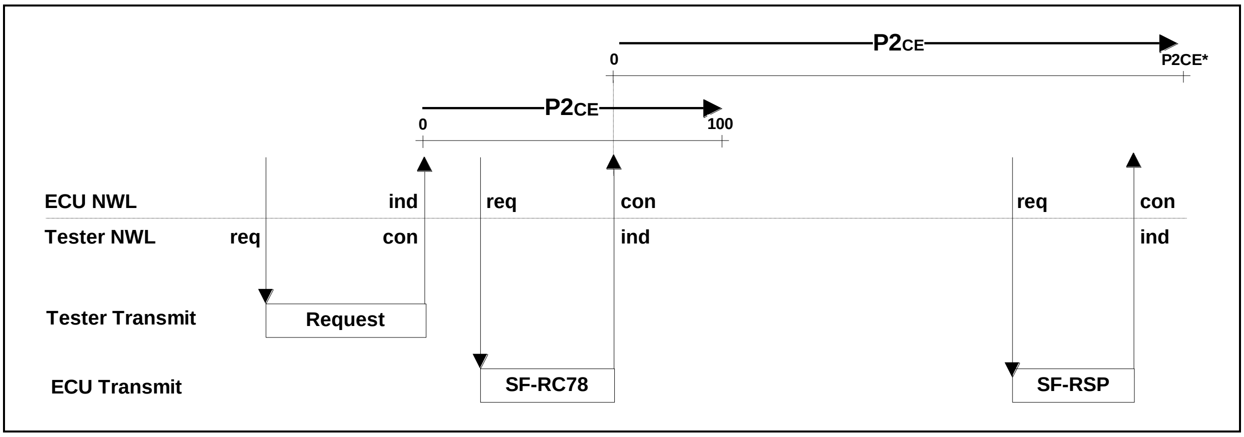


Figure 22: ECU $P2_{CE}$* Timing Parameter Definition - Single RC78 and Final SF Response Message

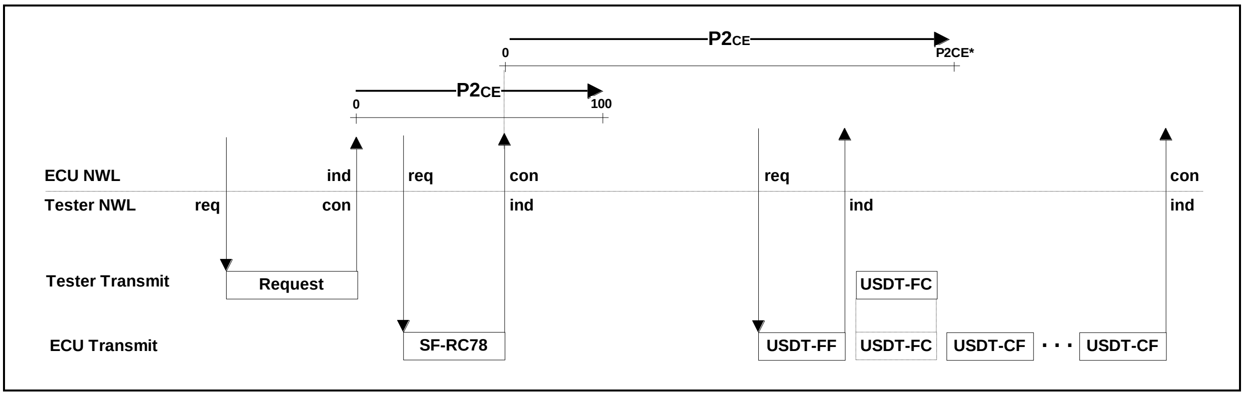


Figure 23: ECU $P2_{CE}$* Timing Parameter Definition - Single RC78 and Final Multi-Frame Response Message

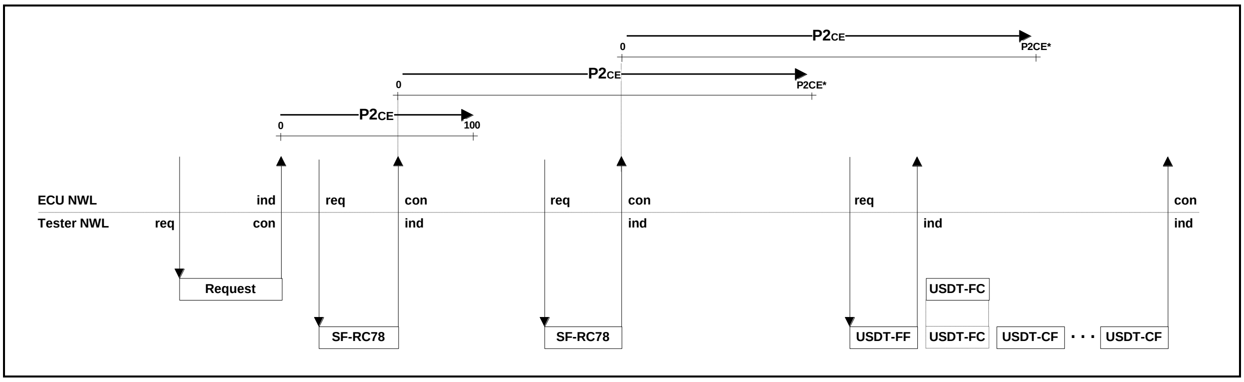


Figure 24: ECU $P2_{CE}$*Timing Parameter Definition - Multiple RC78 and Final Multi-Frame Response Message

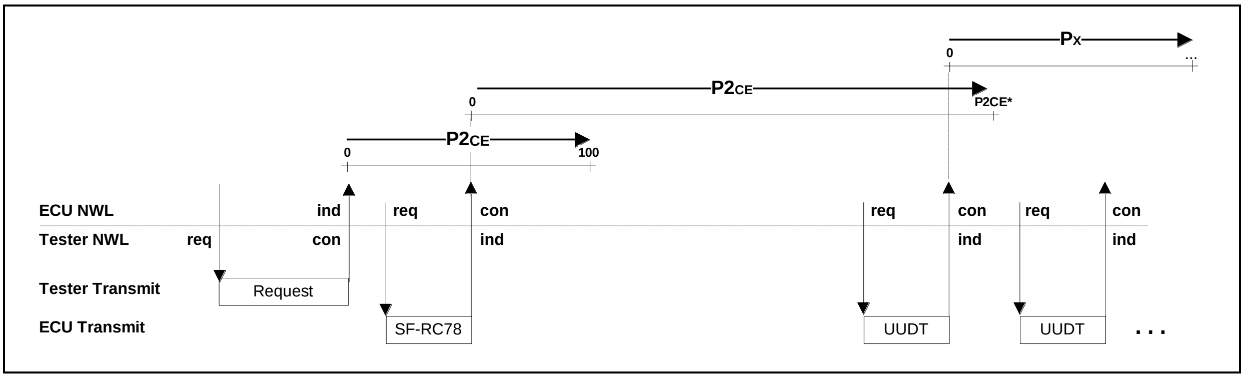


Figure 25: ECU $P2_{CE}$*Timing Parameter Definition - Single RC78 and Final UUDT Response Message (One or Multiple)

Note: The timing that applies to subsequent UUDT messages, optionally transmitted by the ECU following the first UUDT message is not shown in the figure above.

6.2.2.2 Tester Timing Parameter $P2_{CT}$* (Table 30 and Figures 26 thru 28). If an ECU requests an enhanced response timing by responding with a negative response message with response code $78 then the tester has to modify its $P2_{CT}$ timer reload value to the value of $P2_{CT}$*. The handling of the enhanced response timing using the $P2_{CT}$* timing parameter value is based on the general $P2_{CT}$ handling with the exception that the timer reload value is modified to be the $P2_{CT}$* timing parameter value.

Table 30: Tester $P2_{CT}$* Timing Parameter Value

Parameter	Description	Minimum (CTmin*) [ms]	Maximum (CTmax*) [ms]
$P2_{CT}$*	Enhanced response timing value for tester $P2_{CT}$ timer.	5100	N/A Note 1

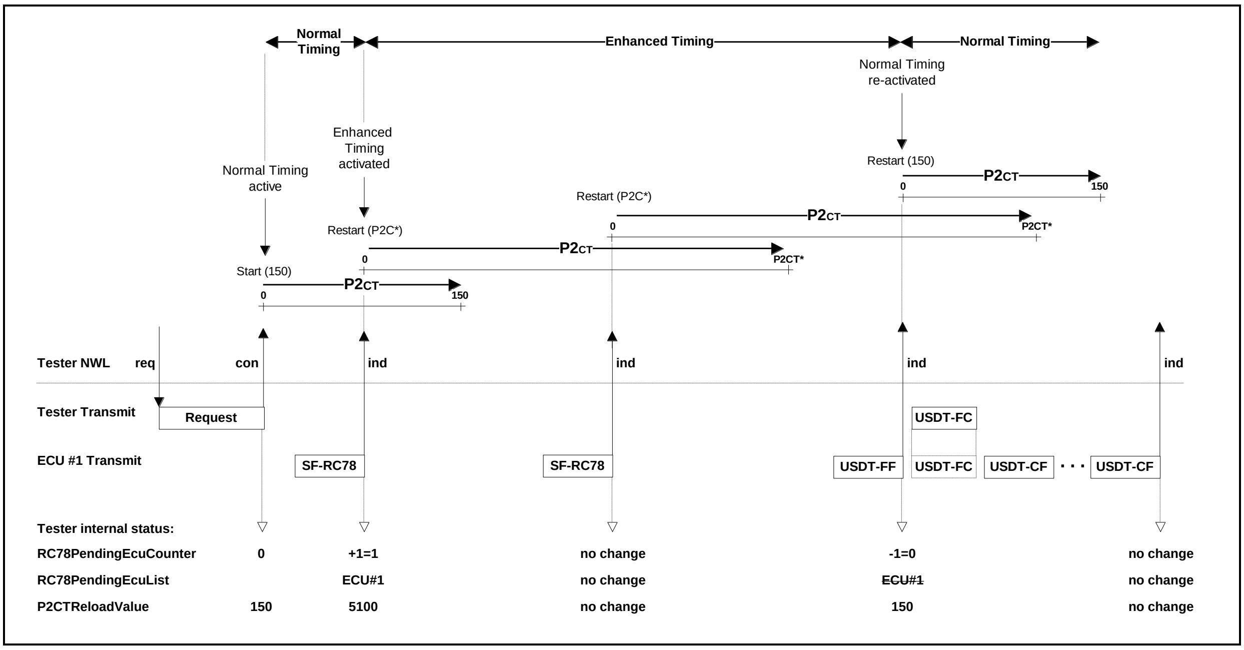
Note 1: The value that a tester uses for $P2_{CTmax}$* is left to the discretion of the tester as long as it is greater than $P2_{CTmin}$*.

The tester shall perform the following handling:

- After correct reception of a negative response message with response code $78 following a request message (single frame or multi-frame request message) the $P2_{CT}$ parameter timing value shall be set to $P2_{CT}$* by the tester and the restart of the $P2_{CT}$ timer shall be performed with this new value.
- Each USDT response message (positive and negative response) and the first UUDT response message, as defined for the general handling of $P2_{CT}$, shall restart the $P2_{CT}$ application timer in the tester with the

current active reload value, which can either be the normal $P2_{CT}$ timing parameter value (normal timing), or, if at least one ECU requested an enhanced response timing, the $P2_{CT}$* timing parameter value (enhanced timing).

- The tester shall keep track of the ECUs which requested an enhanced response timing. The tester has to build an ECU list which contains unique identifications of all ECUs which requested an enhanced response timing, because the tester has to know the ECUs which requested enhanced response timing in order to determine when the normal $P2_{CT}$ timing parameter reload value should be re-enabled. The normal $P2_{CT}$ parameter is re-enabled when all ECUs which have requested enhanced response timing have responded with a single frame response message, the first frame of a multi-frame response message or the first USDT response message. The list shall be built based on the negative response messages with response code $78 sent by the ECU(s).
- The tester shall be able to transmit functionally addressed TesterPresent request messages while performing the enhanced response timing (This functionality shall keep a node in a previously activated diagnostic mode of operation if the preparation of the final response message takes longer than $P3_C$.)



Tester internal status:					
RC78PendingEcuCounter	0	+1=1	no change	-1=0	no change
RC78PendingEcuList		ECU#1	no change	~~ECU#1~~	no change
P2CTReloadValue	150	5100	no change	150	no change

Figure 26: Tester $P2_{CT}$* Timing Parameter Definition - Single ECU Responding with Multi-Frame USDT Response Message

In Figure 26, the tester transmits a physically addressed request message and the addressed ECU requests an enhancement of the response timing twice before it finally transmits its multi-frame response message. The tester initially starts with the normal $P2_{CT}$ reload value (after the confirmation of the complete transmission of the request message). When it receives the first negative response with response code $78 it restarts its $P2_{CT}$ timer as defined in the general $P2_{CT}$ handling but with the $P2_{CT}$* reload value. In addition, the tester stores an ECU identification that the final response of this ECU is pending. Any subsequent negative response message with response code $78 restarts the $P2_{CT}$ timer with the currently active $P2_{CT}$ reload value (which is $P2_{CT}$* in the above case). When the ECU finally responds (positive or negative response message with response code other than $78 or first UUDT message), then the tester deletes the ECU identification in its internal list of pending ECU responses and restarts the $P2_{CT}$ timer with the normal $P2_{CT}$ timing parameter value, because no further ECU is stored in the tester internal response code $78 ECU list.

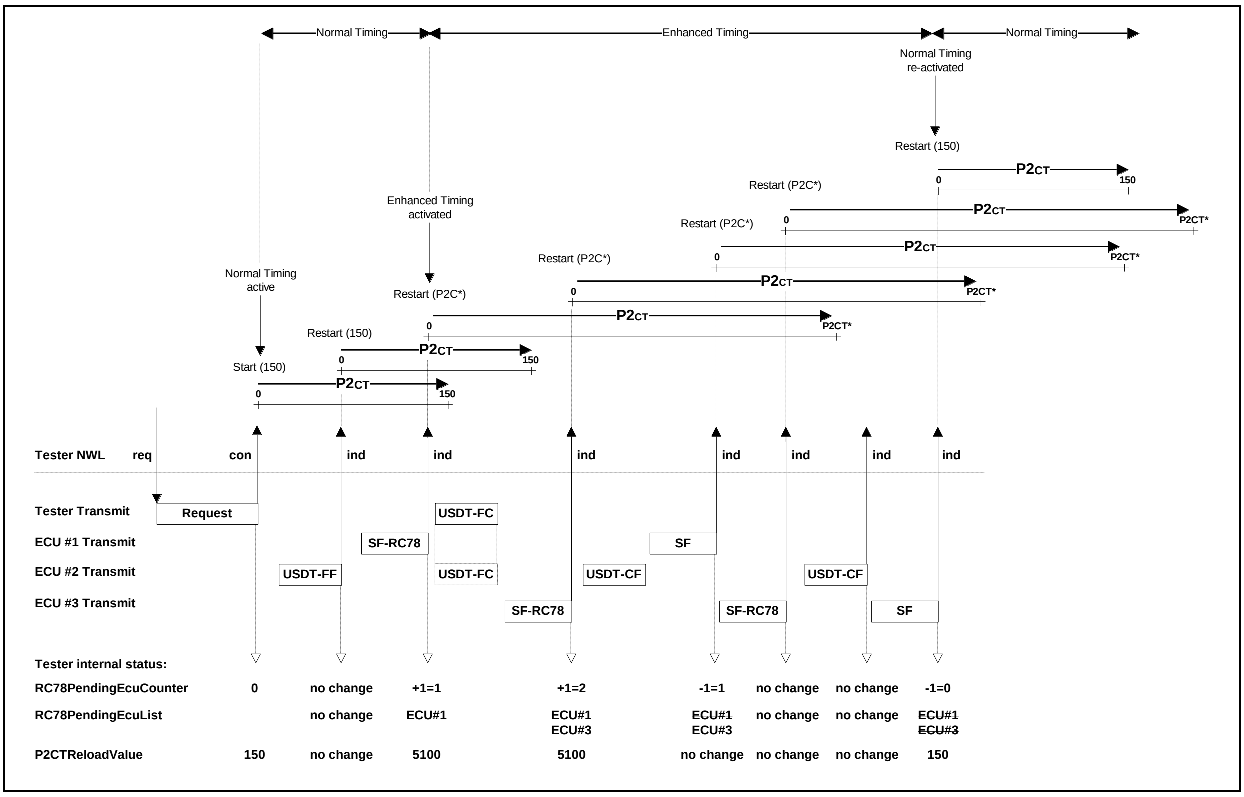


Figure 27: Tester $P2_{CT}$* Timing Parameter Definition - Multiple ECUs Responding

In Figure 27, the tester transmits a functionally addressed request message and three ECUs respond. Two ECUs request enhanced response timing and one ECU immediately responds with a multi-frame response message (two consecutive frames). Based on the general handling of $P2_{CT}$ the tester has to restart the $P2_{CT}$ timer with each starting response message provided the response is associated with the request message. At the point in time when the tester receives the first negative response with response code $78 the tester modifies the reload value of $P2_{CT}$ to the enhanced $P2_{CT}$* timing parameter value and stores an ECU identification that the final response of this ECU is pending. When the ECU finally responds, the tester deletes the ECU identification in its internal list of pending ECU responses. At the point in time when the last ECU which requested an enhanced response timing finally starts its response message, the tester re-enables the normal $P2_{CT}$ reload value for its $P2_{CT}$ timer. Any subsequent response message (single frame or first frame of multi-frame response message) during an active enhanced response timing $P2_{CT}$*forces the tester to restart its $P2_{CT}$ timer as defined for the general handling of $P2_{CT}$ using the currently active reload value.

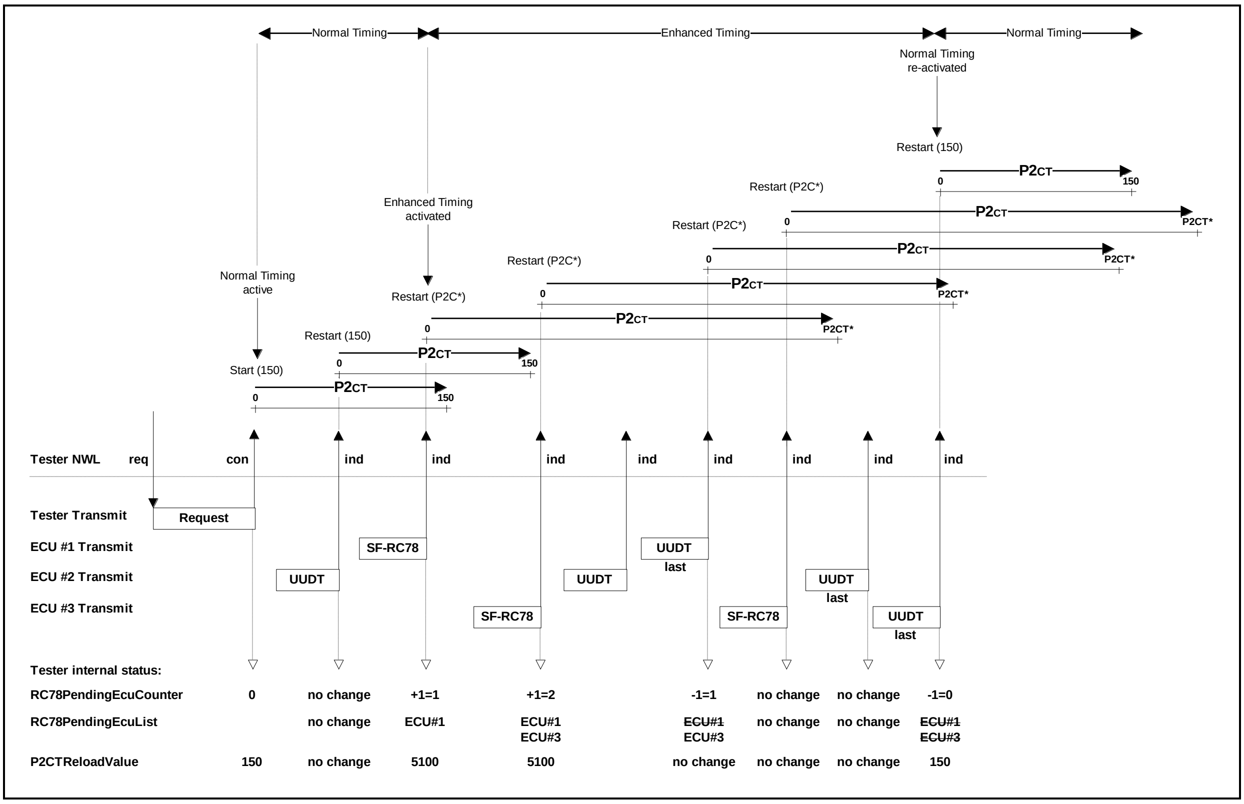


Figure 28: Tester $P2_{CT}$* Timing Parameter Definition - Multiple ECUs Responding with UUDT Response Messages

In Figure 28, the tester transmits a functionally addressed service $AA or $A9 request message and three ECUs respond. Two ECUs request enhanced response timing and one ECU immediately starts to respond with its UUDT message. Based on the general handling of $P2_{CT}$ the tester has to restart the $P2_{CT}$ timer with each starting response message (first UUDT response message) provided the response is associated with the request message (e.g., the Data Packet Identifier (DPID) number in the UUDT response message to a service $AA request message must match the list of requested DPID numbers embedded in the transmitted request message). At the point in time when the tester receives the first negative response with response code $78 the tester modifies the reload value of $P2_{CT}$ to the enhanced $P2_{CT}$* timing parameter value and stores an ECU identification that the final response of this ECU is pending. When the ECU finally starts to respond (first UUDT message), the tester deletes the ECU identification in its internal list of pending ECU responses. At the point in time when the last ECU which requested an enhanced response timing finally starts its response message, the tester re-enables the normal $P2_{CT}$ reload value for its $P2_{CT}$ timer. Any subsequent start of an ECU response message (first UUDT message of this ECU) during an active enhanced response timing $P2_{CT}$* forces the tester to restart its $P2_{CT}$ timer as defined for the general handling of $P2_{CT}$ using the currently active reload value.

6.2.2.3 Enhancement of P2$_{CE}$*/P2$_{CT}$* During A Programming Session (Tables 31 and 32). During a programming session enabled via the sequence defined in the Programming Procedure it is helpful to further enhance the P2$_{CE}$*/P2$_{CT}$* timing to allow ECUs to process, e.g., an **Erase Flash Memory** command which can take longer than 5000 ms without having to send any additional negative response messages with response code $78 (following the initial negative response message with response code $78).

The further enhancement of P2$_{CE}$*/P2$_{CT}$*applies to the ECU(s) and the tester only during an enabled programming session.

Table 31: ECU P2$_{CE}$* Timing Parameter Value During an Active Programming Session

Parameter	Description	Minimum (Cemin*) [ms]	Maximum (Cemax*) [ms]
P2$_{CE}$*	Enhanced response timing for a single ECU during an active programming session.	0	30000

Table 32: Tester P2$_{CT}$* Timing Parameter Value During an Active Programming Session

Parameter	Description	Minimum (Cmin) [ms]	Maximum (Cmax) [ms]
P2$_{CT}$*	Enhanced response timing value for tester P2$_{CT}$ timer during an active programming session.	30200	N/A Note 1

Note 1: The value that a tester uses for P2$_{CTmax}$* is left to the discretion of the tester as long as it is greater than P2$_{CTmin}$*.

The handling which applies to those timing parameter values is identical to the general P2$_{CE}$*/P2$_{CT}$* handling. The only differences are the values to be fulfilled in the ECU (timing requirement) and in the tester (timeout).

6.2.3 Application Timing Parameter Definition for UUDT Response Messages. The diagnostic services ReadDiagnosticInformation ($A9) and ReadDataByPacketIdentifier ($AA) use UUDT messages to transmit the requested data to the tester. UUDT messages use a different CAN Identifier than USDT messages, therefore they can interleave multiple frame USDT messages. UUDT messages are primarily used for the transmission of dynamic data.

An ECU has to comply with the general request/response scheme and therefore has to respond with the first UUDT response message after the successful reception of the request message. The timing of this UUDT response message shall be as described for the P2$_{CE}$ application timing parameter. Following this initial response the ECU can start to transmit the requested data using UUDT messages.

The definition of the timing that applies to the subsequent UUDT response messages can be found in the description of the diagnostic services ReadDiagnosticInformation ($A9) and ReadDataByPacketIdentifier ($AA).

If the ECU cannot respond to the diagnostic service $A9 or $AA within the P2$_{CE}$ timing, because of, e.g., internal checks based on the requested data than it can request an enhanced response timing as defined in paragraph 6.2.2.

6.2.4 Application Timing Parameter P3$_C$ Definition (Table 33). The timing parameter P3$_C$ is defined to be the maximum time between two consecutive TesterPresent ($3E) service request messages. TesterPresent request messages shall be sent by the tester within P3$_C$ when the tester desires certain previously activated diagnostic services to remain active within one or multiple ECU(s). The tester shall always use a P3$_C$ value less than the specified P3$_{Cnom}$ value.

Note: It is recommended that the tester transmits a TesterPresent request message every 2 s.

Note: See the sections of this specification describing the individual diagnostic services to determine which service requires a TesterPresent service to remain active.

Table 33: P3$_C$ Timing Parameter Value

Parameter	Description	Nominal (Cnom) [ms]	Maximum (Cmax) [ms]
P3$_C$	Maximum time between TesterPresent requests sent by the tester.	5000	5100

TesterPresent service request messages can be physically or functionally addressed. Each ECU shall be able to receive and process a functionally addressed TesterPresent request message (single frame message with AllNode CAN Id and valid extended address, e.g., AllNode extended address $FE) in between the reception/transmission of a physically addressed single or multi-frame request/response message and during an active enhanced response timing in order to reset its P3$_C$ timer.

Note: This functionality shall keep a node in a previously activated diagnostic mode of operation if the transmission/receipt of the multiple-frame message or the preparation of the final response message during an active enhanced response timing takes longer than P3$_C$.

The ECU shall recognize that a P3$_C$ timeout (also referred to as TesterPresent timeout) has occurred at a time greater than or equal to P3$_{Cnom}$ and less than or equal to P3$_{Cmax}$:

$$\textbf{Tester P3}_C\textbf{ timeout} < \textbf{P3}_{Cnom} \leq \textbf{ECU P3}_C\textbf{ timeout} < \textbf{P3}_{Cmax}$$

P3$_{Cmax}$ is equal to P3$_{Cnom}$ plus a 2% tolerance. Further references to P3$_C$ within this specification are references to the nominal value of P3$_C$ unless a subscript is included (e.g. P3$_{Cmax}$). Refer to the TesterPresent service description in this specification for more information about the TesterPresent timeout.

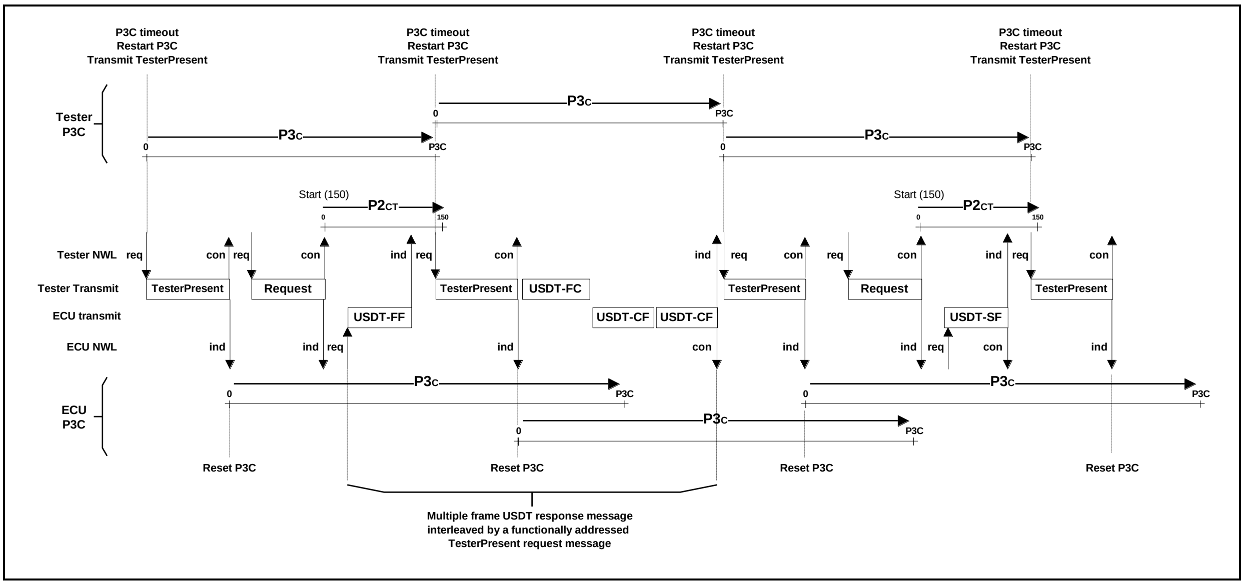


Figure 29: P3$_C$ Timing Parameter Definition and Handling of the TesterPresent Logic

In Figure 29, the tester has set up its P3$_C$ timer to a value less than the P3$_C$ value of the ECU. Therefore, the P3$_C$ timer of the tester times out earlier than the ECU P3$_C$ which forces the tester to transmit its functionally addressed TesterPresent service request message. See Table 34.

Table 34: P3$_C$ Start and Reset Definition for the Tester and ECU

	P3$_C$ Start	P3$_C$ Reset
Tester	Confirmation from the Network Layer that a service has been successfully transmitted that requires the tester to send a TesterPresent message in order to keep the service functionality active.	A P3$_C$ timeout triggers the reset of the P3$_C$ timer and the transmission of the TesterPresent request message.
ECU	When the ECU has received a diagnostic message and the application has determined that the service requested requires the activation of the P3$_C$ timer. See pseudo code of the diagnostic services in this specification.	When the ECU has received a diagnostic message and the application has determined that the service requested is the TesterPresent service. See pseudo code of the TesterPresent service.

The tester implementation of P3$_C$ assumes that the tester has implemented the network layer and utilizes the service primitives that it provides. The network layer of the tester provides a confirmation to the application layer each time any of the following occurs:

- A single frame request is transmitted (N_USData.con).
- The last consecutive frame of a multi-frame request message is transmitted (N_USData.con).
- An error indication is detected by the network layer (N_USData.con with a negative result code).

6.2.5 Application Timing Parameter Considerations. There is a delay involved in an ECU between the reception of a request message from the CAN bus and actually performing any action based on the request (e.g., queuing the transmission of a response message; see Figure 30). The amount of the delay is a function of:

- Whether the CAN controller receive processing is handled via a polling loop or is interrupt driven
- The time it takes for the network layer to evaluate the correctness of the last received CAN frame

- The execution rate of the application which adds an additional delay before network layer indications are actually acted upon **and**
- The time it takes the application to determine the validity of the received diagnostic message.

There is also a delay involved in an ECU between queuing the transmission of a response message and actually finishing the transmission of the first frame (or only frame) of the response message (SF or FF) within the required $P2_{CE}$ timing window. This delay is a function of:

- How long it takes to actually put the first frame (SF or FF) of the queued response message into a transmit object of the CAN controller,
- The amount of time that the message cannot be sent on the bus due to lost arbitration with higher priority CAN frame(s) **and**
- The time to actually transmit the frame on the bus (this is a function of the baud rate, the number of data bytes in the frame, and the bit pattern which effects the number of stuff bits).

Previous figures in the application timing section do not explicitly show these delays. However, the total delay time has to be taken into consideration during the ECU design in order to ensure that the timing requirements are met.

Figure 30 graphically depicts the possible delays involved in the reception of a request message and the earliest point in time that the request can be acted upon (e.g., queue transmission of a response message or start/reset the $P3_C$ timer, if applicable). It also shows the possible delays involved in the transmission of the response message and the earliest point in time the frame can be indicated as a successfully received in the tester.

Note: Reference the diagnostic services chapter of this specification for specific requirements on when it is appropriate to start/reset the $P3_C$ timer.

Note: The ECU has to make sure that the first frame (SF or FF) of the response message is transmitted onto the CAN bus within $P2_{CE}$ when no loss of arbitration occurs. The value of the $P2_{CT}$ timeout within the tester is purposely made greater than $P2_{CE}$ to compensate for bus arbitration under normal operating conditions.

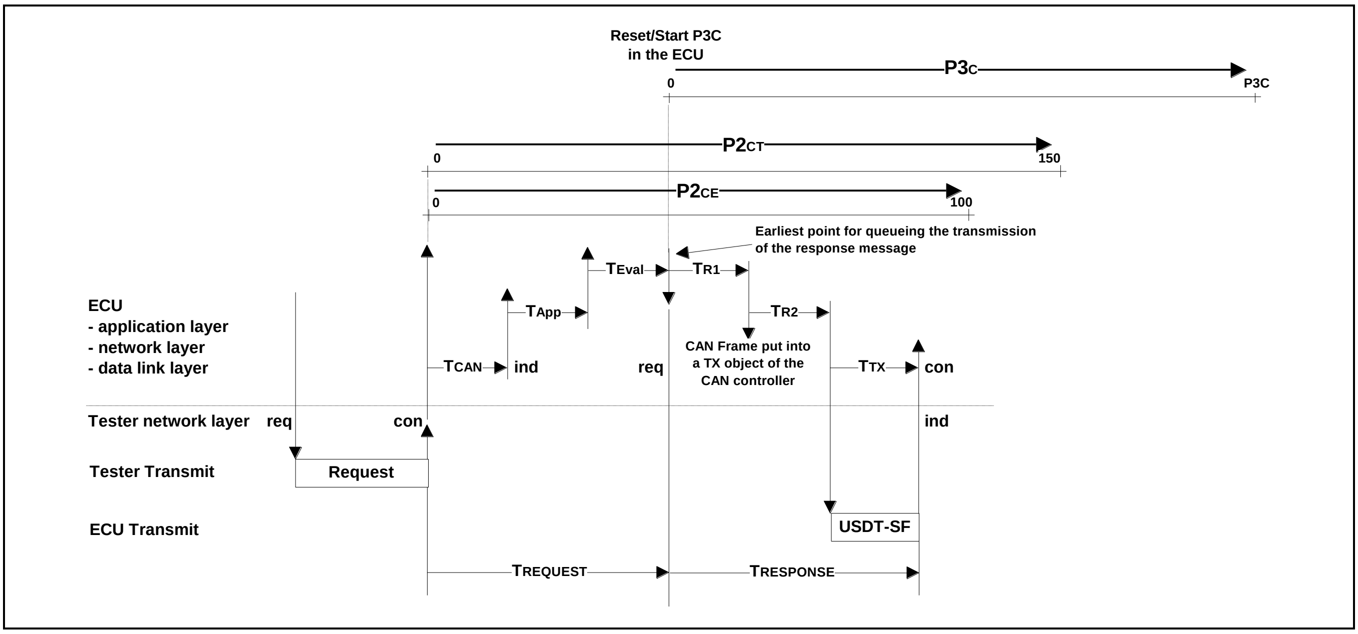


Figure 30: ECU Delays during Reception of A Tester Request Message - Example With SF Response

The total reception delay $T_{REQUEST}$ is made up from the following components:

- **T_{CAN}:** This delay is comprised of the CAN controller polling rate and the evaluation time that it takes the network layer to make sure that the frame is correctly received. In an interrupt driven scheme this delay is based solely on the evaluation time in the network layer.

- **T_{App}:** This delay is based on an application layer running in a certain time schedule. The application can only act on network layer indications when it is activated.
- **T_{Eval}:** This delay is based on the required evaluation of the request message (check of the service identifier and evaluation, that the request message is correct). The figure above shows the earliest point in time when the application can queue its response message.

The total transmit delay $T_{RESPONSE}$ is made up from the following components:

- **T_{R1}:** This is the delay within the ECU between the application queuing the response message and actually placing the first CAN frame of the response (SF or FF) in a transmit object of the CAN controller.
- **T_{R2}:** This is the delay based on arbitration with higher priority CAN frames which can occur before the frame is actually transmitted on the CAN bus. This delay varies based on the bus load and the priority of the response frame compared to the priority of CAN frames in the transmit objects of other ECUs.
- **T_{TX}:** This is the transmit time of a single CAN frame. This time depends on the baud rate of the GMLAN subnet where the CAN frame is transmitted, the number of data bytes in the frame, and the bit pattern which determines the number of stuff bits.

Note: The above described delays for the response message have to be considered in order to meet the $P2_{CE}$ timing requirements defined in this specification. T_{R2} depends on the bus load and frame priorities. If no loss of arbitration occurs, this delay time is equal to 0.

The reception delay ($T_{REQUEST}$) is also applicable when receiving a service which requires a TesterPresent (activation of the $P3_C$ timer) or when receiving a TesterPresent message which resets the $P3_C$ timer. As defined in paragraph 6.2.4, an ECU shall not timeout later than $P3_{Cmax}$. This allows the ECU a total tolerance of 100 ms (2% of 5000 ms) for its $P3_C$ timer ($P3_{Cmax}$ = 5100 ms). The $T_{REQUEST}$ delays shall be comprehended within this 100 ms tolerance.

6.3 Network Layer Parameter Definitions.

6.3.1 Network Layer Timing Parameter Definitions. Table 35 specifies the network layer timing parameters to be used by the tester and the GMLAN ECU(s) for enhanced diagnostic communication based on the definitions in OSEK/COM and ISO 15765-2 Network layer services. Transport layer error handling shall be per ISO 15765-2. The network layer timing parameters specified in Table 35 are also applicable during an active programming event.

Table 35: Network Layer Time-Out and Performance Requirement Values

Parameter	Description	Timeout Value N Note 1	Performance Requirement Value
N_As/ N_Ar	Time from transmit request until a CAN frame transmit confirmation is received.	250 ms	-
N_Bs	Time that the transmitter of a multi-frame message shall wait to receive a flow control (FC) frame before timing out with a network layer error.	250 ms	-
N_Br	Maximum time for the receiver of a multi-frame message to transmit a flow control frame after receiving the first frame (FF).	-	< 100 ms
N_Cs	Maximum time from either receiving a flow control continue to send (FC.ConTS) frame or transmitting a consecutive frame (CF) until transmitting the next CF.	-	< 100 ms
N_Cr	Time that the receiver of a multi-frame (MF) message shall wait to receive a consecutive frame (CF) after sending a flow control (FC) frame or receiving the previous consecutive frame (CF) before timing out with a network layer error.	250 ms	-

Note 1: OBD/EOBD devices whose network layer only supports a single set of timing parameters shall support the timing values specified ISO15765-4 (which references ISO15765-2) for both OBD and enhanced diagnostic communication. Non-OBD/EOBD devices, and OBD/EOBD devices which can support different timing values for OBD vs. enhanced diagnostics, shall support the values in this table for enhanced diagnostic communication.

Note: Each network layer timing parameter (N_Ax, N_Bx, and N_Cx) is specified from the perspective of both the sender of a multi-frame message (e.g., N_As) and the receiver of a multi-frame message (e.g., N_Ar). Timeout values define the amount of time a sender or a receiver must wait for the expected frame before notifying the application of an error condition, and assume that the sender is not transmitting other messages of higher priority than the diagnostic messages. Performance requirements are the timing requirements placed on the sender of a specified frame. A detailed description of the network layer timing parameters can be found in ISO 15765-2 and OSEK/COM specification.

6.3.2 Tester Network Layer Flow Control Frame Transmit Parameter Requirements. A GMLAN tester shall support the Network Layer parameters defined in Table 36 when transmitting flow control frames. The values described in the table are the minimum requirements for the ECU to support when receiving a flow control frame. See Figures 31 and 32. The ECU may support flow control parameter values over and above the minimum specified. However, any time an ECU receives a flow control frame with parameter values that the ECU does NOT support, the ECU shall treat the received frame as an invalid frame and discard it. The ECU shall continue to wait for a valid flow control frame until one is received, or the network layer N_Br timeout occurs. See Tables 37 and 39.

Table 36: Tester Network Layer Network Layer Parameter Values

Parameter	Name	Value	Description
WFT_{max}	WaitFrame Transmission	0 Note 1	No FlowControl wait frames are allowed. The FlowControl frame sent by the tester following the FirstFrame of an ECU response message shall contain the FlowStatus FS set to 0 (ContinueToSend), which forces the ECU to start immediately after the reception of the FlowControl frame with the transmission of the ConsecutiveFrame(s).
BS	BlockSize	0	The FlowControl **BS** parameter value of the first FlowControl frame following the FirstFrame of an ECU response message shall be **zero**, which means that all following ConsecutiveFrames of the response message are sent in one block by the ECU without any subsequent FlowControl frame sent by the tester.
ST_{min}	SeparationTime	0	The FlowControl **ST_{min}** parameter value of the FlowControl frame shall be **zero** (minimum time between consecutive frames). The tester shall be able to receive back-to-back ConsecutiveFrames sent by one or multiple ECU(s) in response to a physically or functionally addressed request message.

Note 1: The WFT_{max} value defines the maximum number of WaitFrames allowed to be transmitted in a row by the receiver of a FirstFrame.

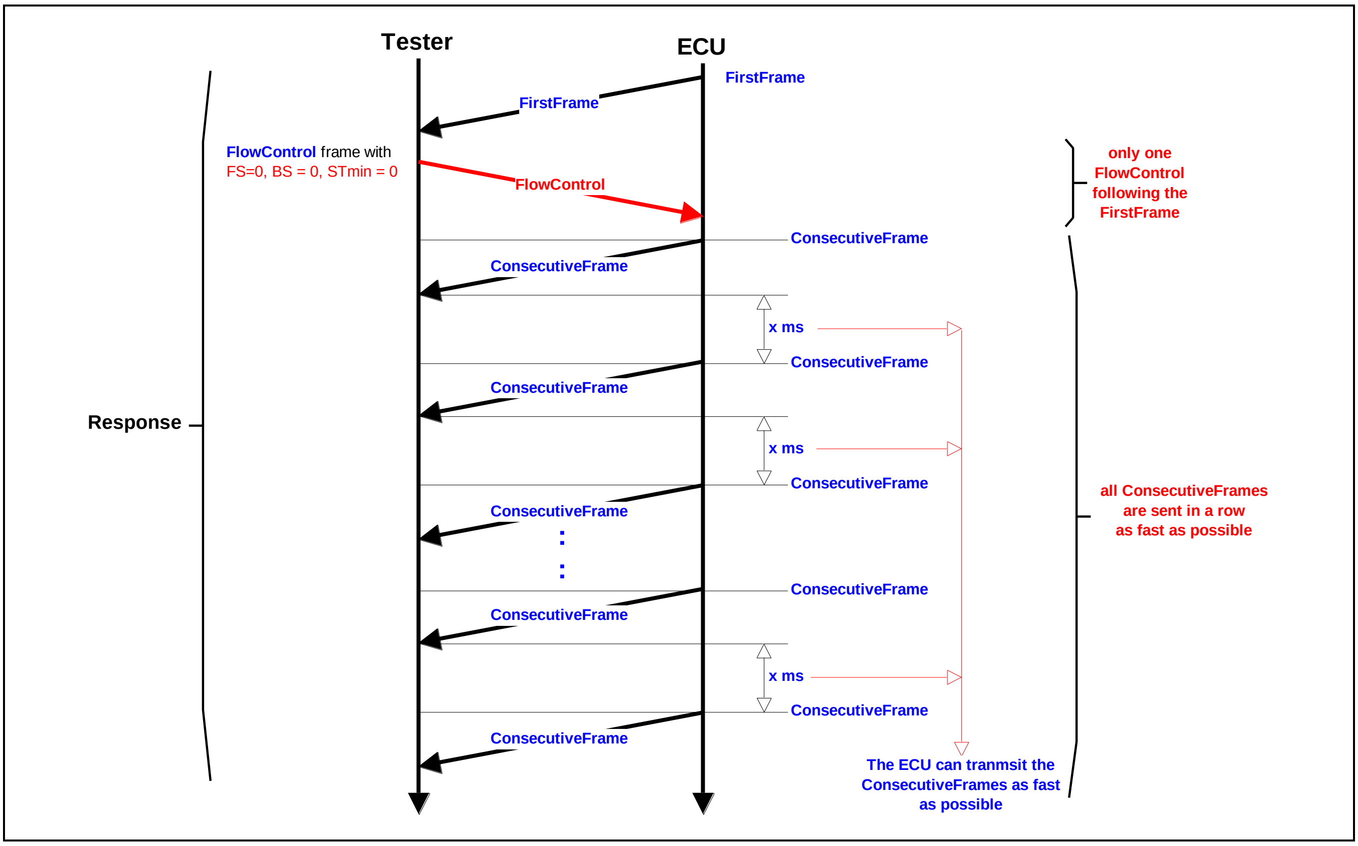


Figure 31: Tester Requirements for Multi-Frame Response Messages - Example Message Flow

Table 37: USDT Response Message Flow Example

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
N(USDT-**FF**)	641	10	24	5A	45	01	02	03	04
T(USDT-**FC**)	241	**30**	**00**	**00**	---	---	---	---	---
N(USDT-**CF**)	641	21	05	06	07	08	09	0A	0B
N(USDT-**CF**)	641	22	0C	0D	0E	0F	10	11	12
N(USDT-**CF**)	641	23	13	14	15	16	17	18	19
N(USDT-**CF**)	641	24	1A	1B	1C	1D	1E	1F	20
N(USDT-**CF**)	641	25	21	22	---	---	---	---	---

6.3.3 ECU Network Layer Flow Control Frame Transmit Parameter Requirements. A GMLAN ECU shall support the following network layer parameters when transmitting flow control frames (Table 38):

Table 38: ECU Network Layer Network Layer Parameter Values

Parameter	Name	Value	Description
WFT_{max}	WaitFrame Transmission	0 [Note 1]	No FlowControl wait frames are allowed. The FlowControl frame sent by the ECU following the FirstFrame of a tester request message shall contain the FlowStatus FS set to 0 (ContinueToSend), which forces the tester to start immediately after the reception of the FlowControl frame with the transmission of the ConsecutiveFrame(s).
BS	BlockSize	0	The FlowControl **BS** parameter value of the first FlowControl frame following the FirstFrame sent by the tester shall be **zero**, which means that all following ConsecutiveFrames are sent in a row by the tester without any subsequent FlowControl frame sent by the ECU.
ST_{min}	SeparationTime	0 ms to Max Polling Rate	The FlowControl **ST_{min}** parameter value (minimum time between consecutive frames, see ISO 15765-2 and OSEK/COM) shall be adjusted to allow the ECU to receive and process a ConsecutiveFrame before receiving the next ConsecutiveFrame. See paragraph 4.3.2 of this specification for requirements on Max Polling Rate.

Note 1: The WFT_{max} value defines the maximum number of WaitFrames allowed to be transmitted in a row by the receiver of a FirstFrame.

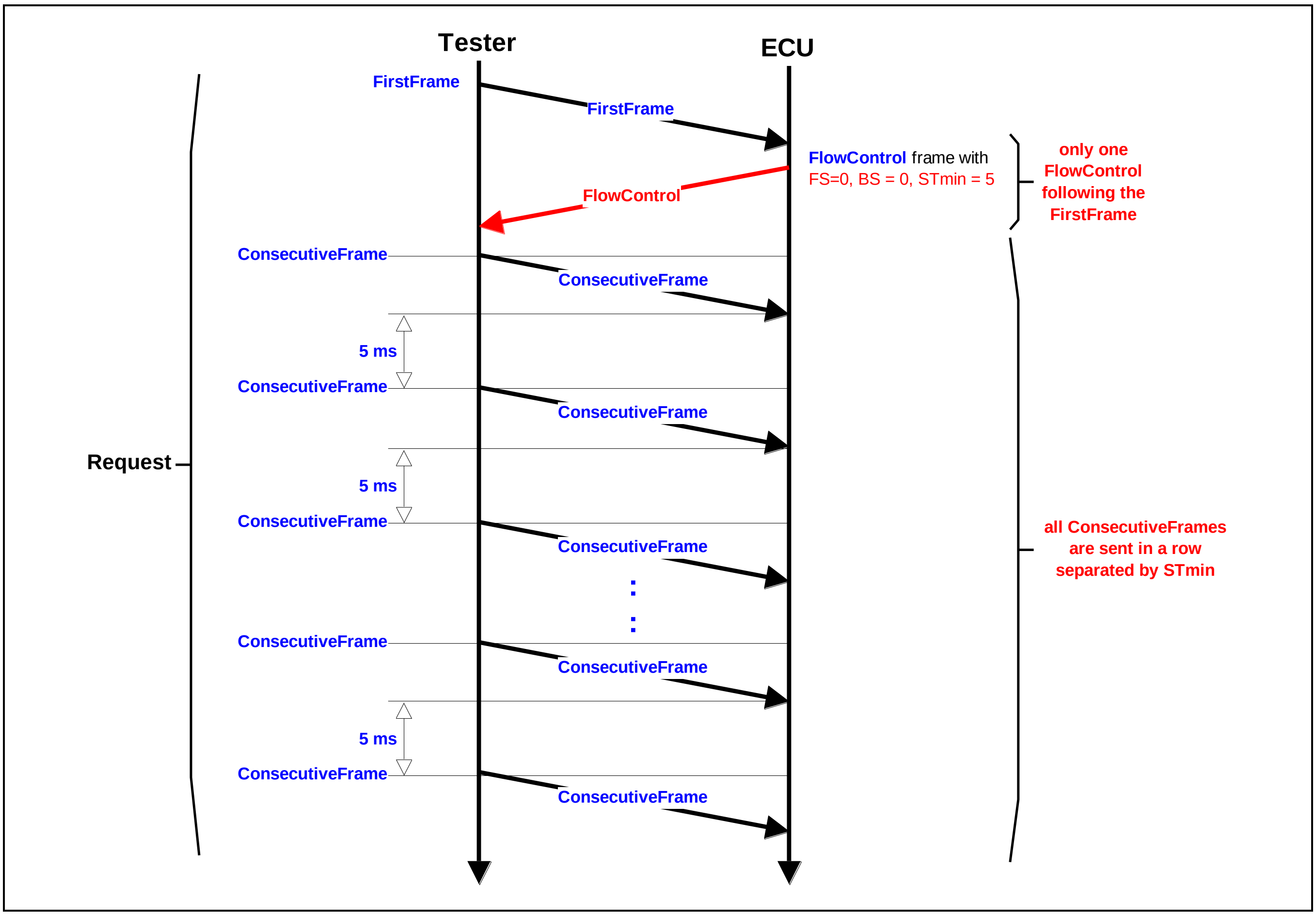


Figure 32: ECU Requirements for Multi-Frame Request Messages - Example Message Flow

Table 39: USDT Request Message Flow Example

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
T(USDT-**FF**)	241	10	24	3B	45	01	02	03	04
N(USDT-**FC**)	641	**30**	**00**	**05**	---	---	---	---	---
T(USDT-**CF**)	241	21	05	06	07	08	09	0A	0B
T(USDT-**CF**)	241	22	0C	0D	0E	0F	10	11	12
T(USDT-**CF**)	241	23	13	14	15	16	17	18	19
T(USDT-**CF**)	241	24	1A	1B	1C	1D	1E	1F	20
T(USDT-**CF**)	241	25	21	22	---	---	---	---	---

7 Negative Response ($7F) Service Definition

The negative response service shall be used by an ECU to indicate that a diagnostic service request message is either invalid, being terminated, or currently cannot be acted upon at the time of the request. Included in the negative response message is a return code to indicate to the tool the reason the negative response message was sent. This service shall use the USDT response CAN Identifier when transmitted on the link.

7.1 Negative Response Message Format (Table 40).

Table 40: Negative Response Message Format

Data Byte	Parameter Name	Cvt Note 1	Hex Value	Mnemonic
#1	Negative Response Service Identifier	M	7F	SIDNR
#2	requestServiceId	M	xx	SIDRQ
#3	returnCode	M	xx	RC_
#4	deviceControlLimitExceeded = [ExceededLimitMsb	C	xx	DCLEH
#5	ExceededLimitLsb]	C	xx	DCLEL

Note 1: C = Conditional: **If returnCode** = $E3 then the deviceControlLimitExceeded parameter is appended.

7.2 Return Code Definition. The following subsections specify the definition of each return code. Each GMLAN Enhanced Diagnostic Service includes a subsection which lists the return code(s) supported by the service. All values which are currently not used as return codes are reserved by this document for future expansion as necessary.

7.2.1 ServiceNotSupported ($11, RC_SNS). This response code indicates that the requested action will not be taken because the ECU does not support the requested service. This return code is only valid for physically addressed diagnostic requests.

Example: The ECU shall send this response code if the tester sends a physically addressed request message with a service Identifier which is either unknown or not supported by the ECU. If a tester sends a valid functionally addressed diagnostic request for a service that is not supported by the ECU, then the ECU shall ignore the request (no response shall be sent). Refer to the **Diag_API_Process_Recv_Msg()** function in the pseudo code of the ReportProgrammedState service ($A2) for implementation details on this response code.

7.2.2 SubFunctionNotSupported-InvalidFormat ($12, RC_SFNS_IF). This response code indicates that the requested action will not be taken because the ECU does not support the arguments of the request message or the format of the argument bytes do not match the prescribed format for the specified service.

Example: The ECU shall send this response code in case the tester has sent a request message with a known and supported service Identifier but with **sub-function parameters** which are either unknown or not supported or have an invalid format.

7.2.3 ConditionsNotCorrectOrRequestSequenceError ($22, RC_CNCRSE). This response code indicates that the requested action will not be taken because the ECU prerequisite conditions are not met. This request may occur when sequence sensitive requests are issued in the wrong order.

Example: The ECU shall send this response code in:

- **Case #1:** The tester has sent a known and supported request message at a time where the ECUs conditions to execute the requested service are unsafe or too critical to perform the requested action.
- **Case #2:** The tester has sent a known and supported request message at a time where the ECU has expected another request message because of a predefined sequence of services. A typical example of occurrence is the securityAccess service which requires a sequence of messages as specified in the message description of this service.

7.2.4 RequestOutOfRange ($31, RC_ROOR). This response code indicates that the requested action will not be taken because the ECU detected a data byte(s) in the request message which attempt(s) to substitute (a) value(s) beyond its range of authority (e.g., attempting to substitute a data byte of 111 when the data is only defined to 100).

Example: The ECU shall send this response code in case the tester has sent a request message including data bytes to adjust a variant which does not exist (invalid) in the ECU. This response code shall be implemented for all services which allow the tester to write data or adjust functions by data in the ECU.

7.2.5 InvalidKey ($35, RC_IK). This response code indicates that security access has not been given by the ECU because the key sent by the tester did not match with the key in the server's (ECU's) memory. This counts as an attempt to gain security. The ECU shall remain locked!

Example: The ECU shall send this response code in case the tester has sent a securityAccess request message with the sendKey and Key parameter where the key value does not match the key value stored in the server's (ECU's) memory. The ECU shall increment its internal securityAccessFailed counter.

7.2.6 ExceedNumberOfAttempts ($36, RC_ENOA). This response code indicates that the requested action will not be taken because the tester has unsuccessfully attempted to gain security access more times than the server's (ECU's) security strategy will allow. Refer to message description of the securityAccess service definition.

Example: The ECU shall send this response code in case the tester has sent a securityAccess request message with the sendKey and Key parameter where the key value does not match the key value stored in the ECU's memory and the number of attempts (securityAccessFailed counter value) have reached the ECU's securityAccessFailed calibration value.

7.2.7 RequiredTimeDelayNotExpired ($37, RC_RTDNE). This response code indicates that the requested action will not be taken because the tester's latest attempt to gain security access was initiated before the ECUs required timeout period had elapsed.

Example: An invalid Key requires the tester to start over from the beginning with a securityAccess service. If the security protection algorithm is not passed after **X** failed attempts (**X** = specified in the ECU's memory), all additional attempts are rejected for at least **Y** seconds (**Y** = specified in the ECU's memory). The **Y** second timer shall begin with the **X** failed attempt or upon a power/ignition cycle or reset of the ECU to ensure a security lockout after all power interruptions.

7.2.8 RequestCorrectlyReceived-ResponsePending ($78, RC_RCR-RP). This response code indicates that the request message was received correctly, and that any parameters in the request message were valid, but the action to be performed may not be completed yet. This response code can be used to indicate that the request message was properly received and does not need to be retransmitted, but the server (ECU) is not yet ready to receive another request. The negative response message with this response code may be repeated by the ECU(s) within $P2_C = P2_C*$ until the final response message (which may be positive or negative, where RC_ is not equal to $78).

Example: A typical example where this response code may be used is when periodic messages are scheduled (see $AA service) and a new request is sent to modify the contents of the scheduler. It is possible that the first response may not be sent within $P2_C$ due to the status of the other messages in the scheduler.

7.2.9 SchedulerFull ($81, RC_SCHDFULL). This response code indicates that the tester attempted to schedule a diagnostic message via the $AA service and the ECUs scheduler table was full.

Example: ECU **A** is capable of scheduling 4 messages and currently has two messages in its scheduler. A request is received by ECU **A** which would attempt to schedule three additional messages. This request would be rejected as it would make the total number of messages that ECU **A** would have scheduled (5), greater than the number which it is capable of scheduling.

7.2.10 VoltageOutOfRangeFault (High/Low) ($83 RC_VOLTRNG). This response code indicates that the voltage is out of the acceptable range at the primary power pin of the ECU.

7.2.11 GeneralProgrammingFailure ($85 RC_PROGFAIL). This response code indicates that the ECU detected an error when erasing or programming a memory location in the permanent memory device (e.g., Flash Memory).

7.2.12 DeviceTypeError ($89 RC_DEV_TYPE_ERR). This response code indicates that the ECU detected an incompatibility between the programming algorithm downloaded and the permanent memory device type.

Example: In this example, an SPS programmable ECU can be manufactured with a flash memory device which can come from two possible suppliers. As such, the programming algorithms for each flash device are contained in the routine section of the utility file. When the programming algorithm is downloaded into the ECU, it checks for the type of flash device present. If the routine is not compatible with the flash device, a negative response message is generated with this response code.

7.2.13 ReadyForDownload-DTCStored ($99 RC_RFD-DS). This response code indicates that the ECU is ready for the download of data but the checksum of flash or EEPROM has resulted in a DTC being set (reference service $34).

7.2.14 DeviceControlLimitsExceeded ($E3, RC_DCLE). This response code indicates that one or more of the ECUs internal device control limitations/restrictions have been exceeded and that all active device control commands have been terminated. This response code requires two additional data bytes to be appended. The additional data bytes identify the specific cause of why the ECU has aborted the device control requested by the tester. A GM Corporate list is included in Appendix **A** of this document. This type of reject message can be sent any time a limit has been exceeded, not just in response to a request message (see examples below).

Example #1: An Anti-lock Brake System (ABS) controller would not want to allow a tester to release the brakes if a vehicle was moving down the road. If the ABS controller sensed that the vehicle was moving above a calibrated threshold and a device control command was received to release the brakes on a given wheel, the node would reject the request and provide this return code in the reject message.

Example #2: The vehicle is not moving and the ABS controller receives a request to release the brakes on one or more wheels. The controller sends a positive response to the request and performs the requested command. Shortly after the request is processed, the vehicle begins moving. When the speed exceeds the maximum allowed, the ECU will send an unsolicited negative response message with this return code and terminates all active device control functions.

7.3 Negative Response Message Flow Example.

7.3.1 Tester Requests An Unsupported Service. In the following example (Table 41), a tester sends a physically addressed request to an ECU for a diagnostic service that is not supported. This results in the ECU sending a negative response with the returnCode set to $11 (ServiceNotSupported).

The following information is given for the **example:**

- The USDT request CANId for the ECU is $241 (and thus, the USDT response CANId is $641).
- The Diagnostic Service Requested is SecurityAccess ($27) and the ECU does not support this service.

Table 41: Negative Response Message Flow Example – ServiceNotSupported

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$27	$01	---	---	---	---	---
N(USDT-SF)	$641	$03	$7F	$27	$11	---	---	---	---

8 Diagnostic Services (Test Modes) Definition

Note: This section describes the detailed operation of each individual test mode.

8.1 ClearDiagnosticInformation ($04) Service. The ClearDiagnosticInformation service is used by the tester to clear diagnostic information in one or multiple nodes' memory. The ClearDiagnosticInformation service is based on the ClearDiagnosticInformation service specified in ISO 15031-5 (SAE J1979: test mode $04).

8.1.1 Service Description. The node shall send a positive response upon receipt of a ClearDiagnosticInformation request (even if no DTCs are stored). It is understood that it may take the node additional time after the positive response to actually complete the clearing of all DTC information. If the amount of time to complete the clear DTC information exceeds 1 s, the worst case time must be documented in the Component Technical Specification. If a node supports multiple copies of DTC status information in memory (e.g., one copy in Random Access Memory (RAM) and one copy inElectronically Eraseable Programmable Read Only Memory (EEPROM)), the node shall clear the copy used by the DTC status reporting service ($A9) followed by the remaining copy.

DTC information reset/cleared via this service includes but is not limited to the following:

- DTC status byte (see ReadDiagnosticInformation ($A9) service).
- Freeze frame data (emission related node(s) only).
- Failure record information.
- Other DTC related data such as flags, counters, timers, etc., specific to DTCs.

8.1.2 Request Message Definition. The ClearDiagnosticInformation request message is used to indicate that one or multiple nodes shall clear the stored diagnostic information. See Table 42.

Table 42: ClearDiagnosticInformation Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ClearDiagnosticInformation Request Service Id	M	04	SIDRQ

8.1.2.1 Request Message Sub-function Parameter $Level (LEV_) Definition. There are no sub-function parameters used by this service.

8.1.2.2 Request Message Data Parameter Definition. There are no data parameters used by this service.

8.1.3 Positive Response Message Definition (Table 43).

Table 43: ClearDiagnosticInformation Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ClearDiagnosticInformation Positive Response Service Id	M	44	SIDPR

8.1.3.1 Positive Response Message Data Parameter Definition. There are no data parameters used by this service in the positive response message.

8.1.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 44) shall be implemented for this service.

Table 44: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This response code shall occur if the length of the request message is incorrect.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** This response code shall be used if internal conditions within the node (ECU) prevent the clearing of DTC related information stored in the node (ECU)	C	CNCORSE
78	**RequestCorrectlyReceived-ResponsePending** This response code can be sent if the ECU is not capable of processing additional diagnostic requests while the DTC clear is taking place. In this case, the final positive response shall be sent upon completion of the DTC clear or once the ECU can accept further diagnostic requests (prior to the finishing of the DTC clear). The ECU shall also reset and temporarily disable the $P3_C$ timer (if the $P3_C$ timer is active) for the duration that the ECU is performing the DTC clear and is not capable of receiving diagnostic requests.	C	RCR-RP

8.1.5 Message Flow Example ClearDiagnosticInformation.

8.1.5.1 Point to Point ClearDiagnosticInformation Request. The tester sends a ClearDiagnosticInformation request message to a physical node. See Table 45.

Network parameter:

- Node physicalRequestCANId = $241.
- Node USDTResponseCANId = $641.

Table 45: ClearDiagnosticInformation Targeted to Physical Node

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$01	$04	---	---	---	---	---	---
N(USDT-SF)	$641	$01	$44	---	---	---	---	---	---

8.1.5.2 Functional ClearDiagnosticInformation Request (Table 46). The tester sends a ClearDiagnosticInformation request message to a functional system using the all node functional request CANId ($101):

- Functional system: $FD = Gateway Devices.
- Physical nodes:
  Node #1: USDTResponseCANId: $641.
  Node #2: USDTResponseCANId: $642.

Table 46: ClearDiagnosticInformation Targeted to Functional System

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FD	$01	$04	---	---	---	---	---
N(USDT-SF)	$641	$01	$44	---	---	---	---	---	---
N(USDT-SF)	$642	$01	$44	---	---	---	---	---	---

8.1.6 Node Interface Function.

8.1.6.1 Node *I*nterface Data Dictionary (Table 47).

Table 47: Node Interface Data Dictionary of ClearDiagnosticInformation Service Pseudo Code

Variable/Meaning	Values
message_data_length	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.1.6.2 Node Interface Pseudo Code. The following logic is executed upon receipt of a ClearDiagnosticInformation ($04) service request message:

Powerup States:

None

Each time a $04 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_04_Msg_Recvd()
IF (message_data_length != 1) THEN
   send Negative Response ($7F $04 $12) /* SubfunctionNotSupported-InvalidFormat */
ELSE
   IF (conditions are not correct to clear diagnostic information) THEN
       send Negative Response ($7F $04 $22) /* ConditionsNotCorrect */
   ELSE
        IF (the ECU cannot continue to process diagnostics while the DTC clear is in process) THEN
```

```
            send Negative Response ($7F $04 $78) /* RequestCorrectlyReceived-ResponsePending */
            Clear DTC status information to be used by $A9 service
            Clear Freeze Frame and Failure Record DTC Data
            Reset any DTC related timers, flags, counters, etc.
            send ($44) response
        ELSE
            send ($44) response
            Clear DTC status information to be used by $A9 service
            Clear Freeze Frame and Failure Record DTC Data
            Reset any DTC related timers, flags, counters, etc.
        ENDIF
    ENDIF
ENDIF
ENDFUNCTION
```

8.1.7 Node Verification Procedure.

Procedure 1:

1. Create various normal operating conditions as well as fault conditions which allow on-board diagnostic routines to execute (thus resulting in changes to the supported status bits of each supported DTC). Verify the status of each supported DTC via the ReadDiagnosticInformation ($A9) service. Continue to exercise the component (including creating fault conditions) to allow all supported status bits for all DTCs to change from their power on initialization value. If a device supports freeze frame or failure record data, verify that freeze frame and or failure record data is present in the node. (Use ReadFailureRecordData ($12) service to verify freeze frames and/or failure record data is present in the node.)
2. Send a ClearDiagnosticInformation ($04) service request and verify the positive response.
3. Wait 1 s (or the time value specified in a CTS, SSTS, or supplemental diagnostic specification referenced by one of the preceding documents) and then read the status of DTCs with the $A9 service. Verify that the appropriate status flags have reset.

Note: It is possible that some DTC algorithms may execute within the 1 s wait time described in this procedure. This must be taken into account when verifying that the status bits reset.

Procedure 2:

1. If certain operating conditions can exist which would inhibit a diagnostic information reset, then place the device in those conditions.
2. Use the ReadDiagnosticInformation ($A9) service to verify the status of all DTCs.
3. Send a ClearDiagnosticInformation ($04) service request, verify negative response ($7F $04 $22).
4. Use the ReadDiagnosticInformation ($A9) service to verify the status of all DTCs. Verify that the DTC status bits have NOT been reset.

Procedure 3:

1. Use the ReadDiagnosticInformation ($A9) service to verify the status of all DTCs.
2. Send a ClearDiagnosticInformation ($04) service request with extra data bytes and verify negative response ($7F $04 $12).
3. Use the ReadDiagnosticInformation ($A9) service to verify the status of all DTCs. Verify that the DTC status bits have NOT been reset.

Procedure 4: (to be checked only if negative response code $78 is supported by the node).

1. Perform step 1 of Procedure 1, then send a ClearDiagnosticInformation ($04) service request message and verify the $7F $04 $78 response. After receiving the final positive response, verify that the node is capable of performing additional diagnostics by requesting a different diagnostic service and verifying the proper response. If the ClearDiagnosticInformation positive response is sent after 1 s (or a maximum value specified in the CTS, SSTS, or supplemental diagnostic specification referenced by the CTS or SSTS), send a ReadDiagnosticInformation ($A9) service and verify that the status of each DTC has been reset.

8.1.8 Tester implications. The tester should take into account that it may take the node(s) additional time after the positive response to actually complete the clearing of all DTC information.

8.2 InitiateDiagnosticOperation ($10) Service. This service allows the tester to perform the following tasks:

- Disable the setting of all DTCs while the tool continues to perform other diagnostic services.
- Allow ECU DTC algorithms to continue to execute while the DeviceControl ($AE) service is active.
- Request a gateway ECU to issue a wake-up request.

8.2.1 Service Description. The disableAllDTCs ($02) and enableDTCsDuringDevCntrl ($03) levels of this service require that a TesterPresent ($3E) message be sent within the P3 timing window in order to keep the functionality active.

If a sub-function parameter has been requested by the tester, which is already running, the ECU shall send a positive response message. If the ECU sends a negative response message to a request for this service, any levels active due to previous requests shall continue to remain active.

8.2.2 Request Message Definition (Table 48).

Table 48: InitiateDiagnosticOperation Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	InitiateDiagnosticOperation Request Service Id	M	10	SIDRQ
#2	Sub-function = [disableAllDTCs enableDTCsDuringDevCntrl wakeUpLinks]	M	 02 03 04	LEV_ DADTC EDDDC WUPLNK

8.2.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. The sub-function parameter is used by the InitiateDiagnosticOperation service to select the specific behavior of the ECU. Explanations and usage of the possible levels are detailed below. The following sub-parameter values are specified in this document (Table 49).

Table 49: Definition of Sub-Function Values

Hex	Description	Cvt
02	**disableAllDTCs** This level shall disable setting of all DTCs.	C
03	**enableDTCsDuringDevCntrl** This level shall be used to allow DTC algorithms to continue to execute while the DeviceControl ($AE) service is active. This request shall have to be made prior to activating DeviceControl or the request shall be rejected. If this service and level are not requested before entering DeviceControl, then DTCs shall be inhibited while DeviceControl is active. (See the $AE service for further details). **Note:** If another diagnostic service is requested which disables DTCs (after the request is sent to allow DTCs to run during DeviceControl) then the DTCs shall become inhibited and remain inhibited until after a TesterPresent timeout occurs or a $20 service is requested.	C
04	**wakeUpLinks** This level shall cause a gateway ECU to initiate the appropriate wake-up sequence on all GMLAN subnets that it is connected to (provided that a given subnet has a wake-up mechanism defined). **Note:** The rules for sending a wake-up as defined in GMW 3104 - GMLAN Communications Strategy Specification still apply (e.g., the strategy specification restricts wake-up requests to have a minimum time interval between them. If a diagnostic request is received to initiate a wake-up and the minimum interval has not expired, then the ECU shall send the positive response message back to the tester without initiating another wake-up). If a GMLAN subnet uses a shared local input as a wake-up wire and the shared local input has to remain asserted to keep communications active, then the gateway device shall ensure that the wake-up wire is asserted while the gateways diagnostic VN is active. **Note:** An example of the shared local input wake-up mechanism described above would be a gateway that is connected to both the single wire CAN link and a dual wire CAN link. In this example, the gateway uses a relay to switch power to the other devices on the dual wire CAN subnet. For normal operations the gateway would receive the High Voltage wake-up on the single wire CAN bus and then enable the relay to provide power to the dual wire devices. If the ECU receives a request for this service with the wakeUpLinks ($04) sub-function parameter, then the ECU would ensure that the relay providing power to the dual wire link ECUs remains enabled as long as the diagnostic VN is active in the gateway (or longer if the ECU would otherwise keep the relay enabled for normal functionality after the diagnostic VN is no longer active).	$C_1^{\textbf{Note 1}}$
00, 01, and 05 thru FF	**ReservedByDocument** This value is reserved by this document for future definition.	M

Note 1: $\mathbf{C_1}$ = The need to support this sub-function is vehicle architecture dependant and as such shall jointly agreed upon by the DRE, the GM Service and Parts Operations responsible engineer and the GM Manufacturing responsible engineer. In general, this sub-function should be supported in vehicles where one or multiple GMLAN links employ a wake-up mechanism that cannot otherwise be accessed by a diagnostic tool.

8.2.2.2 Request Message Data Parameter Definition. This service does not support data parameters in the request message.

8.2.3 Positive Response Message Definition (Table 50).

Table 50: InitiateDiagnosticOperation positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	InitiateDiagnosticOperation Positive Response Service Id	M	50	SIDPR

8.2.3.1 Positive Response Message Data Parameter Definition. There are no data parameters used by this service in the positive response message.

8.2.4 Supported Negative Response Codes (RC_). The following negative response codes (see Table 51) shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 51: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** 1. The length of the request message is invalid. 2. The ECU does not support the sub-function parameter value	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** 1. This return code shall be sent if DeviceControl is active when a request for this service is received with the sub-function parameter set to $03. 2. This return code shall be sent if a request for this service is received with the sub-function parameter set to $03 and another diagnostic service has already disabled DTCs (e.g., DisableNormalCommunication $28).	C	RC_CNCRSE
78	**RequestCorrectlyReceived-ResponsePending** This return code shall be sent if it will take more than P2C ms to process this request (e.g., It might take longer than P2C ms for a gateway to initiate a wake-up on all subnets that it is connected to.)	C	RCR-RP

8.2.5 Message Flow Example InitiateDiagnosticOperation Service.

8.2.5.1 InitiateDiagnosticOperation(disableAllDTCs). The example below (Table 52) shows how the InitiateDiagnosticOperation(level = disableAllDTCs) service shall be implemented. The example assumes the following information to be true:

- The All-Node CAN Id is $101 and the All-Node functional system address is $FE.
- The USDT response CAN Id of the ECUs are: $641 thru $645.

Table 52: InitiateDiagnosticOperation(disableAllDTCs)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$10	$02	---	---	---	---
N(USDT-SF)	$641	$01	$50	---	---	---	---	---	---
N(USDT-SF)	$642	$01	$50	---	---	---	---	---	---
N(USDT-SF)	$643	$01	$50	---	---	---	---	---	---
N(USDT-SF)	$644	$01	$50	---	---	---	---	---	---
N(USDT-SF)	$645	$01	$50	---	---	---	---	---	---

8.2.5.2 InitiateDiagnosticOperation(enableDTCsDuringDevCntrl). The example below (Table 53) shows how the InitiateDiagnosticOperation(level = enableDTCsDuringDevCntrl) service shall be implemented. The example assumes the following information to be true:

- The physical request CAN Id of the ECU is $241.
- The USDT response CAN Id of the ECU is $641.

Table 53: InitiateDiagnosticOperation(enableDTCsDuringDevCntrl)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$10	$03		---	---	---	---
N(USDT-SF)	$641	$01	$50	---	---	---	---	---	---

8.2.5.3 InitiateDiagnosticOperation(wakeUpLinks). The example below (Table 54) shows how the InitiateDiagnosticOperation(level = wakeUpLinks) service shall be implemented. The example assumes the following information to be true:

- The request message is sent on the LS-CAN sub-net of GMLAN and there are two gateways connected to the sub-net (a gateway to HS-CAN and a gateway to MS-CAN).
- The USDT response CAN Id of the first gateway is $641.
- The functional system address for gateways is $FD (used as extended address in functional request).
- The USDT response CAN Id of the second gateway is $64A.

Table 54: InitiateDiagnosticOperation(wakeUpLinks)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FD	$02	$10	$04	---	---	---	---
N(USDT-SF)	$641	$01	$50	---	---	---	---	---	---
N(USDT-SF)	$64A	$01	$50	---	---	---	---	---	---

8.2.6 Node Interface Function.

8.2.6.1 Node Interface Data Dictionary (Table 55).

Table 55: Node Interface Data Dictionary of InitiateDiagnosticOperation Service Pseudo Code

Variable/Meaning	Values
message_data_length **TesterPresent_Timer_State** **Diag_Services_Disable_DTCs** **DTCs_Enabled_In_Device_Control**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
valid_request This flag is locally used within this service to determine if a positive response is needed and whether to activate the TesterPresent_Timer_State	YES/NO

8.2.6.2 Node Interface Pseudo Code.

Powerup States:

Each time a $10 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_10_Msg_Recvd()
IF (message_data_length != 2) THEN
    Send Negative Response ($7F $10 $12) /*SubfunctionNotSupported-InvalidFormat */
    valid_request ← NO
ELSE
    valid_request ← YES
    SELECT FIRST
    WHEN ($Level= LEV_DADTC ) /* $02 Disable All DTCs */
        TesterPresent_Timer_State ← ACTIVE
        Diag_Services_Disable_DTCs ← TRUE
    WHEN ($Level = LEV_EDDDC) /* $03 allow DTCs to run during device control */
        /* verify that device control is not already active and that no other service has disabled DTCs before
        setting the flag to allow them to continue to run during device control */
        IF (Diag_Services_Disable_DTCs = TRUE) THEN
            Send negative response ($7F $10 $22) /*ConditionsNotCorrectOrRequestSequenceError */
            valid_request ← NO
        ELSE
            TesterPresent_Timer_State ← ACTIVE
            DTCs_Enabled_In_Device_Control ← YES
        ENDIF
    WHEN ($Level= LEV_WUPLNK ) /* $04 wake-up all links */
        IF (more than $P2_C$ is needed to process this request) THEN
            send Negative Response ($7F $10 $78) /* request correctly received - response pending */
        ENDIF
        generate wake-up on all GMLAN links where wake-up mechanism is supported
        if the wake-up mechanism implemented requires the gateway to continuously provide a hardwired
        signal to keep the ECUs awake then the ECU shall set any necessary flags to ensure that the wake-up
        remains active as long as the diagnostic VN is active
    OTHERWISE
```

```
        Send negative response ($7F $10 $12) /*for SubfunctionNotSupported-InvalidFormat */
        valid_request ← NO
    ENDSELECT
ENDIF
IF (valid_request = YES)
    Send ($50) positive response message
ENDIF
ENDFUNCTION
```

8.2.7 Node Verification Procedure.

Procedure 1:

1. Send a $10 message with an invalid $level parameter and verify the negative response ($7F $10 $12).
2. Send a $10 message without a sub-parameter included and verify the negative response ($7F $10 $12).
3. Send a $10 message with data bytes after the sub-parameter and verify the negative response ($7F $10 $12).
4. If negative response code $78 is supported by the ECU, then create the conditions under which the ECU should return the $7F $10 $78 response and verify that proper response is sent.
5. Repeat the previous step of this procedure for each possible reason an ECU would send the negative response with response code $78.

Procedure 2: (if sub-function $02 is supported).

1. Send a $10 message with the sub-function parameter set to $02 (disableAllDTCs).
2. Create a failure which would set a DTC and verify that no DTC is set.
3. Send TesterPresent messages for 2 minutes and then verify that no DTCs are set.
4. Stop sending diagnostic messages for $P3_{Cmax}$ ms (ensures a TesterPresent timeout to enable DTCs) and verify that the appropriate DTC is set (once all the criteria to set the DTC have been fulfilled).
5. Repeat steps 1 through 3 of this procedure, and then send a $20 service request. Verify the appropriate response to the service $20 and then check that the appropriate DTC is set (once all the criteria to set the DTC have been fulfilled).

Procedure 3: (if sub-function $03 is supported).

1. Send a $10 message with $level = $03 and then activate a device control. Send TesterPresent messages for a time greater than $P3_{Cmax}$ (but less than any device control restriction timer) and then create a failure which would set a DTC. Verify that the DTC is set via the $A9 service (undo the failure condition created and clear DTCs prior to proceeding to step 2 of this procedure).
2. Send a $10 message with $Level = $03 and then activate a device control. Next send a $10 message with $level = $02. Create a failure which would set a DTC and then verify that no DTC is set.
3. Send a $10 message with $Level = $03 while device control is already active and verify the negative response ($7F $10 $22).
4. Send a service $28 message on the link and verify the positive response. Then send a $10 request with $Level = $03 and verify the negative response ($7F $10 $22).

Procedure 4: (if sub-function $04 is supported).

1. Send a $10 request with $Level = $04 and verify that the device performs the appropriate wake-up mechanism on all GMLAN subnets that it is connected to that support a wake-up.
2. If the wake-up mechanism on any GMLAN link requires continuous activation to keep the ECUs awake (e.g., gateway provides power through a relay to ECUs on a specific subnet), then send messages to the gateway to keep the gateways diagnostic VN active. Periodically check that the tester can communicate to other ECUs on the subnet via diagnostic services. This test should be performed under conditions where the ECU would otherwise let the link go down if the diagnostic VN were not active. Allow the diagnostic VN to end, and then verify that communications with other ECUs is no longer possible.

8.2.8 Tester Implications. The disableAllDTCs ($02) and enableDTCsDuringDevCntrl ($03) levels of this service require a tester to send the TesterPresent ($3E) service at least once every $P3_C$ ms or the requested functionality will be terminated.

8.3 ReadFailureRecordData ($12) Service. This service is used to obtain failure record information that was captured due to a fault detected within the node.

8.3.1 Service Description. Two levels are used within this service to be able to obtain the failure record information from a node. One level, readFailureRecordIdentifiers (sub-function parameter = $01), allows the tester to obtain information necessary to send a request to retrieve the data parameters associated with a specific failure record stored in a node. The information needed to request specific failure record data is called a failure record identifier and is comprised of a failure record number and the DTC identifier (2-byte DTC number + 1-byte DTC fault type). The second level, readFailureRecordParameters (sub-function parameter = $02), allows the test tool to retrieve the data parameters in the failure record associated with the failure record identifier.

Note: Refer to the $A9 service and Appendix E for more information.

Note: In emission-related devices, failure record number $00 is also known as the Freeze Frame and shall be reserved for Emission Related Freeze Frame data required for OBD and EOBD. The Freeze Frame is required for storing the failure information for the first emission related DTC that is stored. There are certain specified data parameters that are required to be captured in the freeze frame for OBD and EOBD. The required parameters for OBD are documented in SAE J1979 and for EOBD are documented in ISO/WD 15031-5. Additional data parameters may also be captured and stored in the freeze frame if they are needed for engineering or service.

Data parameters that are stored in a failure record must be identified by the node with either 2-byte Parameter Identifiers (PID) or 1-byte Data Packet Identifiers (DPID). The node will indicate which format is used with the failureRecordDataStructureIdentifier parameter in the readFailureRecordIdentifiers positive response. A failureRecordDataStructureIdentifier parameter value of $00 indicates that PIDs are used to identify the data parameters in the failure record. A failureRecordDataStructureIdentifier parameter value of $01 indicates that DPIDs are used to identify the data parameters in the failure record.

All failure records for a given node shall support the same format. PID and DPID numbers and their associated data shall be coordinated with service and manufacturing and shall be documented within the device's CTS or within a supplemental diagnostic specification referenced by the CTS.

Note: Nodes that support failure records shall clear all association of failure record numbers to DTC numbers upon a successful code clear.

8.3.2 Request Message Definition (Table 56).

Table 56: ReadFailureRecordData Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadFailureRecordData Request Service Id	M	$12	SIDRQ
#2	sub-function = [readFailureRecordIdentifiers readFailureRecordParameters]	M	 $01 $02	LEV_ RFRI RFRP
 #3 #4 #5 #6	failureRecordIdentifier = [failureRecordNumber DTCHighByte DTCLowByte DTCFailureTypeByte]	M_1 Note 1	 xx xx xx xx	FRDTCIREC FRN DTCHB DTCLB DTCFT

Note 1: M_1 = Bytes 3 thru 6 are mandatory if the sub-function parameter is readFailureRecordParameters ($02). They are not included in the request message if the sub-function parameter is readFailureRecordIdentifiers ($01).

8.3.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. The following sub-function $levels (LEV_) of operation are defined for service $12 ReadFailureRecordData (Table 57):

Table 57: Definition of Sub-function Values

Hex	Description	Cvt
$01	**readFailureRecordIdentifiers** Allows a tester to read the failureRecordDataStructureIdentifier and the failure record identifiers. A failure record identifier is made up of a failure record number and a DTC identifier. **Note:** This level is only used to obtain the failureRecordDataStructureIdentifier parameter and the failure record identifiers. The data bytes contained within a particular failure record are retrieved with the readFailureRecordParameters ($02) level of service.	M
$02	**readFailureRecordParameters** Allows a tester to read the failure record data parameters for a single failure record. Failure record data parameters consist of important ECU inputs, outputs, and calculated values which provide information about the operating conditions of the vehicle at the time the DTC is logged. Examples of failure record data parameters might include engine revolutions per minute (rpm), vehicle speed, etc. Valid failure record identifier data values are obtained by the tester from the readFailureRecordIdentifiers ($01) level of this service.	M
$00 $03 thru $FF	**ReservedByDocument** This value is reserved by this document for future definition.	M

8.3.2.2 Request Message Data Parameter Definition (Table 58).

Table 58: ReadFailureRecordData Request Message Data Parameter Definition

Definition
failureRecordIdentifier The failureRecordIdentifier bytes are only included in the request message if the sub-function parameter is readFailureRecordParameters ($02). The failureRecordIdentifier consists of 4 bytes which includes the failure record number and the DTC identifier (2 byte DTC number + 1-byte DTC fault type). This format allows for maximum flexibility as it allows ECUs to store a single failure record per DTC, or to store multiple failure records for each new occurrence of the same DTC.

8.3.3 Positive Response Message Definition. Below are the tables which describe the format and content of positive responses corresponding to the sub-function level of the request message and the structure of the data record (using PIDs or DPIDs).

8.3.3.1 Positive Response Message - $Level $01 (readFailureRecordIdentifiers). The response to a readFailureRecordIdentifiers ($01) request contains the failureRecordDataStructureIdentifier parameter and the failure record identifiers for each failure record linked to a stored DTC within the node. See Table 59. If the node has no failure record identifiers linked to a stored DTC, then the response to a readFailureRecordIdentifiers request shall only contain the level byte and the failureRecordDataStructureIdentifier parameter.

Table 59: ReadFailureRecordData(readFailureRecordIdentifiers) Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadFailureRecordDataPositiveResponse Service Id	M	$52	SIDPR
#2	sub-function = [readFailureRecordIdentifiers]	M	$01	LEV_ RFRI
#3	failureRecordDataStructureIdentifier = [	M		FRDSI
	PID		$00	PID
	DPID]		$01	DPID
	failureRecordIdentifier#1 = [	C Note 1		FRDTCIREC
#4	failureRecordNumber		xx	FRN
#5	DTCHighByte		xx	DTCHB
#6	DTCLowByte		xx	DTCLB
#7	DTCFailureTypeByte]		xx	DTCFT
. . .	. . .	. . .	. . .	. . .
	failureRecordIdentifier#m = [	C Note 1		FRDTCIREC
#n-3	failureRecordNumber		xx	FRN
#n-2	DTCHighByte		xx	DTCHB
#n-1	DTCLowByte		xx	DTCLB
#n	DTCFailureTypeByte]		xx	DTCFT

Note 1: C = Conditional. The number of failure record identifiers reported in the response message will vary based on the number of failure records which are linked to a stored DTC at the time of the request. If no failure records are linked to a DTC at the time of the request, then the failureRecordDataStructureIdentifier byte is the last byte of the response message.

8.3.3.2 Positive Response Message - $Level $02 (readFailureRecordParameters). The positive response to a readFailureRecordParameters request contains all the failure record data parameters associated with the requested failure record identifier. Tables 60 thru 63 show the response formats based on the value of the failureRecordDataStructureIdentifier parameter in the readFailureRecordIdentifiers ($01) response message.

8.3.3.2.1 readFailureRecordParameters Positive Response Format Using PIDs (Table 60).

Table 60: ReadFailureRecordData(readFailureRecordParameters) Positive Response Message with PIDs

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadFailureRecordDataPositiveResponse Service Id	M	$52	SIDPR
#2	sub-function = [	M		LEV_
	readFailureRecordParameters]		$02	RFRP
	failureRecordIdentifier = [	M		FRDTCIREC
#3	failureRecordNumber		xx	FRN
#4	DTCHighByte		xx	DTCHB
#5	DTCLowByte		xx	DTCLB
#6	DTCFailureTypeByte]		xx	DTCFT
	PIDRecord#1 = [			PIDREC
#7	PIDHB	M	xx	PIDHB
#8	PIDLB	M	xx	PIDLB
#9	PIDDataByte#1	M	xx	PIDDB
:	:	:	xx	:
:	PIDDataByte#m]	C_1 Note 1	xx	PIDDB
:	:	:	:	:
	PIDRecord#k = [	C_2 Note 2		PIDREC
:	PIDHB		xx	PIDHB
:	PIDLB		xx	PIDLB
:	PIDDataByte#1		xx	PIDDB
:	:		xx	:
#n	PIDDataByte#m]		xx	PIDDB

Note 1: C_1 = The number of data bytes associated with the PIDRecord is determined by the definition of the PID.
Note 2: C_2 = The number of PIDRecords in the response is based on the number of PIDs that the ECU records into the failure record each time a failure record is stored.

8.3.3.2.2 readFailureRecordParameters Positive Response Format Using DPIDs (Table 61).

Table 61: ReadFailureRecordData(readFailureRecordParameters) Positive Response Message with DPIDs

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadFailureRecordDataPositiveResponse Service Id	M	$52	SIDPR
#2	sub-function = [	M		LEV_
	readFailureRecordParameters]		$02	RFRP
	failureRecordIdentifier = [	M		FRDTCIREC
#3	failureRecordNumber		xx	FRN
#4	DTCHighByte		xx	DTCHB
#5	DTCLowByte		xx	DTCLB
#6	DTCFailureTypeByte]		xx	DTCFT
	DPIDRecord#1 = [			DPIDREC
#7	DPID	M	xx	DPID
#8	DPIDDataByte#1	M	xx	DPIDDB
:	:	:	xx	:
:	DPIDDataByte#m]	C_3 Note 1	xx	DPIDDB
:	:	:	:	:
	DPIDRecord#k = [	C_4 Note 2		DPIDREC
:	DPID		xx	DPID
:	DPIDDataByte#1		xx	DPIDDB
:	:		xx	:
#n	DPIDDataByte#m]		xx	DPIDDB

Note 1: C_3 = The number of bytes in a DPIDRecord is determined by the definition of the DPID. DPIDs can range in length from one to seven bytes.

Note 2: C_4 = The number of DPIDRecords in the response is based on the number of DPIDs that the ECU records into the failure record each time a failure record is stored.

8.3.3.3 Positive Response Message Data Parameter Definition (Table 62).

Table 62: Response Message Data Parameter Definition

Definition
sub-function This byte is an echo of the sub-function parameter from the request message.
FailureRecordDataStructureIdentifier This data parameter is used to indicate whether or not the ECU stores failure records in a PID or DPID format. A PID is a 2-byte value pre-assigned to a specific piece of ECU signal/parameter data (e.g., rpm or Coolant temperature are pieces of signal/parameter data, and each would have a unique two byte value used to identify it). The number of data bytes associated with a specific PID is fixed when the PID is defined. A DPID is a 1- to 7-byte set of ECU signals/parameters packed into a UUDT message reported via the $AA service. If an ECU stores failure records in a DPID format, then the order of the signals/parameters within the failure record match the way they are defined when reporting the DPID.
FailureRecordIdentifier See request message data parameter definition section.
PIDRecord# This parameter consists of a 2-byte PID value used to identify the PID data and the actual PID data itself (ECU signal/parameter data). The number of data bytes in the record is dependant upon the PID being reported.
DPIDRecord# This parameter consists of a 1-byte DPID number used to identify which signal/parameter data is being reported and the actual ECU signal/parameter data itself. The number of data bytes in the record is dependant upon the DPID being reported.

8.3.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 63 below:

Table 63: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** 1. A readFailureRecordIdentifiers request message does not contain one data byte following the service identifier. 2. A readFailureRecordParameters request message does not contain five data bytes following the service identifier. 3. The request message contains a sub-parameter which is not supported.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** This response code shall be used if internal conditions within the node (ECU) prevent the reading of failure record information stored in the node (ECU).	C	CNCORSE
31	**RequestOutOfRange** A request for the parameters in a failure record contains a failure record identifier (failure record number + DTC identifier) that does not correspond to a single unique failure record or is not a valid failure record identifier for the node.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** See 7.2 Return Code Definition.	C	RCR-RP

8.3.5 Message Flow Examples. The example below (Table 64) shows a request for the failure record identifiers of the PCM followed by a request for the data parameters from one of the failure records which contains a DTC. The example assumes the following information to be true:

- The point to point request CAN Id for the PCM is $241.
- The USDT response CAN Id for the PCM is $641.
- The PCM contains failure record numbers $00, $01, $02, and $03.
- A failure record is stored for DTC P1671 ($16 $71) with a DTC Fault Type byte = $00 in both failure record $00 (freeze frame) and $01 (first failure record).
- No DTCs are stored in failure records $02 and $03.
- The PCM supports providing failure record data with PIDs.
- Engine rpm is a 2-byte parameter stored in the failure record and is identified by PID# $000C.
- Vehicle Speed is a 1-byte parameter stored in the failure record and is identified by PID# $000D.
- Manifold Pressure is a 1-byte parameter stored in the failure record and is identified by PID# $000B.
- Engine Coolant Temperature is a 1-byte parameter stored in the failure record and is identified by PID# $0005.
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.

8.3.5.1 Example #1 - Obtain Failure Record Identifiers.

Table 64: Example 1 ReadFailureRecordData Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$12	$01	---	---	---	---	---
N(USDT-FF)	$641	$10	$0B	$52	$01	$00	$00	$16	$71
T(USDT-FC)	$241	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$641	$21	$00	$01	$16	$71	$00	---	---

8.3.5.2 Example #2 - Obtain Failure Record Data Parameters from a Failure Record. The example below (Table 65) shows a request to obtain failure record data parameters from the PCM described in Example 1 (Table 64).

Table 65: Example 2 ReadFailureRecordData Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$06	$12	$02	$01	$16	$71	$00	---
N(USDT-FF)	$641	$10	$13	$52	$02	$01	$16	$71	$00
T(USDT-FC)	$241	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$641	$21	$00	$0C	$xx	$xx	$00	$0D	$yy
N(USDT-CF)	$641	$22	$00	$0B	$zz	$00	$05	$aa	---

The example below (Table 66) shows a request for the failure record identifiers of the PCM that supports the failure records from Example 1 (Table 64), but does not have any failure records currently stored.

8.3.5.3 Example #3 - Obtain Failure Record Identifiers.

Table 66: Example 3 ReadFailureRecordData Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$12	$01	---	---	---	---	---
N(USDT-SF)	$641	$03	$52	$01	$00	---	---	---	---

8.3.6 Node Interface Function.

8.3.6.1 Node Interface Data Dictionary (Table 67).

Table 67: Node Interface Data Dictionary of ReadFailureRecordData Service Pseudo Code

Variable/Meaning	Values
message_data_length	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
record_index This is a local variable used in the pseudo code when searching through the failure records to find the record that matches the request message.	$00 thru $FF
num_records This is a variable used in the pseudo code to indicate the maximum number of failure records that a node will store.	$00 thru $FF
record_array For the purposes of the pseudo code, it is assumed that the failure record data is stored as an array of data structures. Each data structure contains the following elements: 1. record_number_stored 2. DTC_Number 3. DTC_FailureType 4. Data The data referenced in element 4 above is the first byte of DTC specific data stored in the failure record. To access an element of the data structure, the following nomenclature is used: record_array[record_index].element **Where:** record_index is described above in the data dictionary and element is one of the four types listed here.	The values are described below for each data type of the record: 1. $00 thru $255 (node specific) 2. A 2-byte value is used to indicate the DTC set. Reference mode $A9 for more details on DTC format. 3. A 1-byte value used to describe the FailureType of a stored DTC. 4. This field can be multiple bytes long and contain data which must be specified in a CTS or supplemental diagnostic specification.

Variable/Meaning	Values
Record_match_found This is a flag used in the pseudo code to keep track of whether a record exists which matches the one requested.	YES/NO
DTC_Number This is a local variable that is set equal to the DTC Number from the request message.	Matches the value from the request message

8.3.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $12 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_12_Msg_Recvd()
SELECT FIRST
WHEN ($level=$01) /* readFailureRecordIdentifiers request message */
    IF (message_data_length != 2)
        send Negative Response ($7F $12 $12) /* Invalid Format. */
    ELSE IF (Device cannot provide Failure Records at this time) THEN
        Reject the request with a ($7F $12 $22) for conditions not correct
    ELSE IF (Processing this message will take more than P2C ms) THEN
        Send ($7F $12 $78) /*for RequestCorrectlyReceived-ResponsePending */
        …..
        Send ($52 $01 ….) /* positive response message */
    ENDIF
WHEN ($level=$02) /* readFailureRecordParameters request message */
    IF (message_data_length != 6)
        send Negative Response ($7F $12 $12) /* Invalid Format. */
    ELSE IF (Device cannot provide Failure Records at this time) THEN
        Reject the request with a ($7F $12 $22) for conditions not correct
    ELSE
        record_match_found ← NO
        DTC_Number ← ((failureRecordIdentifier[DTCHighByte] << 8) |failureRecordIdentifier[DTCLowByte])
        FOR (record_index ← 0 TO (num_records - 1) BY 1)
            /* the DTC comparison below shall check for a match of both the MSB and the LSB */
            IF ( (record_array[record_index].record_number_stored =
             failureRecordIdentifier[failureRecordNumber]) AND
            (record_array[record_index].DTC_Number = DTC_Number) AND
            (record_array[record_index].DTC_FailureType =
             failureRecordIdentifier[DTCFailureTypeByte]) )
                record_match_found ← YES
            ENDIF
        ENDFOR
        IF (record_match_found = NO)
            send Negative Response ($7F $12 $31) /* Request Out Of Range. */
        ELSE IF (Processing this message will take more than P2C ms) THEN
```

```
            Send ($7F $12 $78) /*for RequestCorrectlyReceived-ResponsePending */
            …..
            Send ($52 $02 ….) /* positive response message */
        ENDIF
    ENDIF
OTHERWISE
    Send negative response ($7F $12 $12) /*for SubFunctionNotSupported-InvalidFormat */
ENDSELECT
ENDFUNCTION
```

8.3.7 Node Verification Procedure.

Procedure 1:

1. Send a request message with no data bytes following the service identifier and verify the $7F $12 $12 response.
2. Send a readFailureRecordIdentifiers request message with more than 1 data byte following the service identifier and verify the $7F $12 $12 response.
3. Send a readFailureRecordParameters request message with more than 5 data bytes following the service identifier and verify the $7F $12 $12 response.
4. Repeat step 3 using less than 5 data bytes following the service identifier and verify the $7F $12 $12 response.
5. Send a request message with an invalid sub-function parameter value and verify the $7F $12 $12 response.

Procedure 2:

1. Send a valid readFailureRecordIdentifiers request message when the device has no failure records stored and verify the proper response which contains no data following the failureRecordDataStructureIdentifier.
2. Send a valid readFailureRecordIdentifiers request message when the device has at least one failure record stored and verify proper response.
3. Send a valid readFailureRecordParameters request message and verify proper response which includes data parameters for the failure record identifier.

Procedure 3:

1. Send a readFailureRecordParameters request message for a failure record with a failure record identifier (failure record number + DTC identifier) that does not correspond to a single unique failure record in the node and verify the $7F $12 $31 response.

Procedure 4:

1. If negative response code $78 is supported by the ECU, then create the conditions under which the ECU should return the $7F $12 $78 response and verify that proper response is sent.
2. Repeat previous step of this procedure for each possible reason an ECU would send the negative response with response code $78. Verify this for each applicable supported sub-function parameter.

Procedure 5:

1. If negative response code $22 is supported by the ECU, then create the conditions under which the ECU should return the $7F $12 $22 response and verify that proper response is sent.
2. Repeat previous step of this procedure for each possible reason an ECU would send the negative response with response code $22. Verify this for each applicable supported sub-function parameter.

8.3.8 Tester Implications. This service should only be physically addressed (point to point).

8.4 ReadDataByIdentifier ($1A) Service. The purpose of this service is to provide the ability to read the content of pre-defined ECU data referenced by a dataIdentifier (DID) which contains static information such as ECU identification data or other information which does not require **real-time** updates. (**Real-time** data is intended to be retrieved via the ReadDataByPacketIdentifier ($AA) service.)

8.4.1 Service Description. The request message shall always include one dataIdentifier. The length of the positive response message shall be adapted to the size of data referenced by the dataIdentifier parameter value. This message shall include the dataIdentifier parameter value received in the request message.

The tester is only required to provide the dataIdentifier value desired. The target ECU is responsible for knowing the address location and block length corresponding to the requested DID. An ECU is not required to support all corporate defined DIDs, and may support additional application specific DIDs.

Note: Appendix C of this specification contains a list of corporate standard DIDs.

8.4.2 Request Message Definition (Table 68).

Table 68: ReadDataByIdentifier Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDataByIdentifier Request Service Id	M	1A	SIDRQ
#2	dataIdentifier	M	xx	DID_

8.4.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. This service does not use a sub-function parameter.

8.4.2.2 Request Message Data Parameter Definition. Table 69 specifies the data parameter definitions for this service.

Table 69: Data Parameter Definition

Definition	Msg Type
dataIdentifier The DID is used within this service to indicate to the ECU application, which static data the tool is requesting. If an ECU supports application specific DIDs (non corporate standard), then the DID number, the data contents, and other information contained within the tables in Appendix C, must be documented within the ECU CTS or within a supplemental diagnostic specification referenced by the CTS.	Request

8.4.3 Positive Response Message Definition (Table 70).

Table 70: ReadDataByIdentifier Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDataByIdentifier Positive Response Service Id	M	5A	SIDPR
#2	dataIdentifier	M	xx	DID_
 #3 : #n	dataRecord = [Data Byte #1 : Data Byte #m]	 M : U	 xx : xx	DREC_ DB_1 : DB_m

8.4.3.1 Positive Response Message Data Parameter Definition (Table 71).

Table 71: Request Data Parameter Definition

Definition
dataIdentifier The dataIdentifier parameter in the response message is an echo of the one in the request message to confirm to the tool that the positive response is for the DID requested. If an ECU supports application specific DIDs (non corporate standard), then the DID number, the data contents, and other information contained within the tables in Appendix C, must be documented within the ECU CTS or within a supplemental diagnostic specification referenced by the CTS.
dataRecord[] This is the portion of the response message which contains the data stored in the ECU memory location(s) of the requested DID. The length of this field is dependant upon the dataIdentifier requested.

8.4.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 72) shall be implemented for this service.

Table 72: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This response code shall be sent if the request message does not have a DID number after the Request Service Id, or if additional data bytes are included beyond the DID byte.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** This response code shall be sent if the operating conditions of the ECU are such that it cannot perform the required action (e.g., the data for a DID is stored in EEPROM and an EEPROM failure has occured). This response code is not intended to be used if the ECU conditions are such that the request temporarily cannot be performed (e.g., the data to be retrieved temporarily cannot be read from EEPROM because another application task is currently writing to EEPROM). Response code $78 is used when conditions are temporarily not met. If this response code is supported by a node, the reason(s) to generate this return code must be documented in the Component Technical Specification (CTS).	C	CNC-RSE
31	**RequestOutOfRange** • This code shall be sent if the dataIdentifier value requested is not supported by the device. • This code shall be sent if the dataIdentifier is secured and the ECU is not in an unlocked state.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** • This response code shall be sent if reading the data from the address(es), which is referenced by the dataIdentifier, takes longer than $P2_C$ ms (e.g., the data to be read has to be retrieved from another processor via an ECU internal Serial Peripheral Interface (SPI) bus). • This response code shall be sent if the ECU operating conditions are such that the request temporarily cannot be performed (e.g., the data to be retrieved cannot be read from EEPROM because another application task is currently writing to EEPROM).	C	RCR-RP

8.4.5 Message Flow Example ReadDataByIdentifier Service.

8.4.5.1 ReadDataByIdentifier(dataIdentifier=VIN). The example below (Table 73) shows how the ReadDataByIdentifier(dataIdentifier=VIN) service shall be implemented. The example assumes the following information to be true:

- The point to point request CAN Id for the ECU is $241.
- The USDT response CAN Id of the ECU is $641.
- The Data Identifier for the Vehicle Identification Number (VIN) is $90 (as defined in Appendix C).
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.

Table 73: ReadDataByIdentifier(dataIdentifier=VIN)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$1A	$90	---	---	---	---	---
N(USDT-FF)	$641	$10	$13	$5A	$90	“W”	“0”	“L”	“0”
T(USDT-FC)	$241	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$641	$21	“J”	“B”	“F”	“3”	“5”	“W”	“1”
N(USDT-CF)	$641	$22	“0”	“4”	“2”	“7”	“6”	“5”	---

8.4.6 Node Interface Function.

8.4.6.1 Node Interface Data Dictionary (Table 74).

Table 74: Node Interface Data Dictionary of InitiateDiagnosticOperation Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **Security_Access_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.4.6.2 Node Interface Pseudo Code.

Powerup States:

None

The following logic is executed when a request message with ServiceId $1A is received by the ECU:

```
BEGIN FUNCTION Serv_1A_Msg_Recvd()
IF (message_data_length != 2) THEN
    send Negative Response ($7F $1A $12) /*Invalid Format */
ELSE
    IF (($dataIdentifier is NOT supported) OR ($dataIdentifier requires security access AND
    Security_Access_Unlocked = FALSE) OR ($dataIdentifier requires security code AND
    Security_Access_Allowed = FALSE)) THEN
        send Negative Response ($7F $1A $31) /* Request Out Of Range */
    ELSE IF (ECU long term conditions prevent retrieval of the data) THEN
        send Negative Response ($7F $1A $22) /* ConditionsNotCorrect */
    ELSE
```

```
        IF (device cannot respond with block contents within P2C ms) THEN
            send Negative Response ($7F $1A $78) /* RequestCorrectlyReceived-ResponsePending */
        ENDIF
        /*Get data values from memory address associated with the $dataIdentifier*/
        send ($5A $dataIdentifier $Data) /* send response message with DID data */
    ENDIF
ENDIF

ENDFUNCTION
```

8.4.7 Node Verification Procedure.

Procedure 1:

1. Send a $1A message with each $dataIdentifier supported and verify proper response (test data formatting).

Procedure 2: (Only applicable if ECU has secure DIDs).

1. Send a $1A message with a secure $dataIdentifier value when the device is the manufacturers enable counter (MEC) = $00 and the vulnerability flag < $FF and verify the negative response ($31 - Request Out Of Range).
2. Send a $1A message with a secure $dataIdentifier value when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has not been accessed (SecurityAccess ($27) request has not been sent) and verify the negative response ($31 - Request Out Of Range).
3. Send a $1A message with a secure $dataIdentifier value when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has been accessed (SecurityAccess ($27) request has been sent) and verify the appropriate positive response.
4. Repeats steps 1 through 3 for each secure DID.

Procedure 3:

1. Send a $1A message with an invalid $dataIdentifier parameter and verify the negative response ($31 - Request Out Of Range).
2. Send a $1A message without a dataIdentifier included (e.g., request message ends after the service Id) and verify the negative response ($12 - Invalid Format).
3. Send a $1A message with data bytes after the sub-parameter and verify the negative response ($12 - Invalid Format).
4. Send a $1A message to the ECU with a dataIdentifier where the ECU needs more than $P2_C$ ms to read data bytes from memory and send the positive response (if applicable). Verify that the ECU sends the negative response ($78 - RequestCorrectlyReceived-ResponsePending) within $P2_C$ ms followed by a positive response (reference application timing section of this specification).

Procedure 4: (Applicable if the ECU implements this return code).

1. Send a request for a valid dataIdentifier at a time when ECU internal conditions will not allow the data to be retrieved (e.g., EEPROM failure) and verify that the ECU sends the correct negative response ($22 ConditionsNotCorrect).

Procedure 5: (Only applicable if ECU has DIDs that require the usage of a security code).

Note: Security code as defined in the Vehicle Theft Deterrent SSTS.

1. Send a $1A message with a security code required $dataIdentifier value when the security code has not been sent, manufacturers enable counter = $00, and the vulnerability flag < $FF. Verify the negative response ($31 - Request Out Of Range) is sent.
2. Send a $1A message with a security code required $dataIdentifier value when the manufacturers enable counter > $00 or vulnerability flag = $FF and the security code has not been sent. Verify the negative response ($31 - Request Out Of Range) is sent.
3. Send a $1A message with a security code required $dataIdentifier value when the manufacturers enable counter > $00 or vulnerability flag = $FF and the security code has been sent. Verify the appropriate positive response is sent.

4. Repeats steps 1 thru 3 for each security code required $dataIdentifier value.

8.4.8 Tester Implications. This service can result in responses which are single frame or multiple frame based on the DID requested. The tester should not functionally address requests for this service for a DID which results in multiple frame responses unless it can handle sending the flow control frames to each responding node within the allowable network timing parameters.

8.5 ReturnToNormalMode ($20) Service. The purpose of this service is to return a node or group of nodes to normal mode operation by canceling all active diagnostic services and resetting normal message communications (if they were interrupted by a diagnostic operation).

All nodes participating in a GMLAN network shall support this service even if the node itself is diagnosed over another vehicle bus (e.g., KWP2000 or Class 2). This requirement is necessary to facilitate programming of other devices on the GMLAN subnet.

8.5.1 Service Description. The following enhanced diagnostic services are terminated and/or reset by service $20:

- All levels of service $10 **InitiateDiagnosticOperation** are terminated.
- Service $27 **SecurityAccess** is terminated and a node shall become locked if the Manufacturers Enable Counter is $00 and the Vulnerability Flag (if implemented) is not $FF.
- Service $28 **DisableNormalCommunication** is terminated.
- All levels of service $A5 **ProgrammingMode** are terminated.
- Service $A9 **ReadDiagnosticInformation** send-on-change reporting of DTC count information is terminated.
- Service $AA **ReadDataByPacketIdentifier** periodic message scheduler logic is reset. However, the node shall retain all dynamically defined message (DPID) information.
- Service $AE **DeviceControl** is terminated, thereby returning full control of input(s)/output(s) to the node.
- Any ECU resources allocated for the DataTransfer ($36) service which were a result of receiving a RequestDownload ($34) request, shall be re-allocated back to their original purpose.

In addition, if a request for this service is received during a programming session (activated via the $A5 service), the programming session shall be considered concluded and all devices receiving the request shall perform a software reset.

Note: The software reset allows a device which had just been programmed to begin executing the new software and calibrations downloaded. The reset of all nodes also resynchronizes the start-up of normal communications.

If a high speed programming event was enabled on the low speed SWCAN link when a request for this service is received, then all ECUs (including the tester) shall initialize their protocol converter hardware within 30 ms from the time that the $20 request is successfully transmitted on the link. The low speed ECUs shall perform the software reset after re-initializing the protocol converter hardware. If the low speed ECUs can reset the CAN controller and perform a software reset in less than 30 ms, the low speed ECUs shall reset the CAN controller immediately and delay the software reset the necessary amount of time to ensure that communication does not begin in less than 30 ms from the time that the $20 request is transmitted on the link.

Note: The delay is necessary to prevent bus errors that would occur if all nodes are not at the same baud rate when one node begins normal communication.

Note: Any node which uses a polling loop to service the protocol device shall ensure that the polling loop is fast enough to process the request message and initialize the protocol converter hardware within 30 ms. The reset of the protocol device shall take place prior to invoking the software reset. This is necessary to ensure that no timing issues exist with some nodes completing the reset and attempting to initialize normal communications before another device can initialize its protocol converter during its reset.

When using this service to end a programming session, the tester must target the request at all nodes on the network via a functionally addressed request ($101 $FE $01 $20). A valid request for this service which concludes a programming event shall not be followed by a positive response. The positive response for this case has been eliminated due to timing issues involved with the possibility of transitioning back to the normal baud rate on the low speed subnet.

If a request for this service is received while normal communications are disabled with the ($28) DisableNormalCommunications service, and a programming session is not active, then the node shall reinitialize normal communications. Reinitializing normal communications consists of the nodes application performing necessary tasks (e.g., resetting or clearing flags, variables, etc.) as needed, and then invoking the handler function(s) executed while in the Comm Init state.

Note: Refer to GMW 3104 and the Diagnostics And Node Management section of this specification for more details about the Comm Init state.

If the setting of diagnostic trouble codes (DTC) has been disabled via the $10, $28, or $AE services, then the node shall re-enable the diagnostics and take the necessary steps to reinitialize diagnostic counters, flags, timers, etc in order to prevent false DTCs from being set due to the fact that the algorithm was disabled for some period of time.

Note: It may be necessary for a node to reset certain DTC algorithms or variables used within these algorithms prior to executing a software reset at the conclusion of a programming event.

An ECU shall send an unsolicited service $20 positive response message any time a TesterPresent ($3E) timeout ($P3_C$) occurs and a programming session is not active.

Note: If a $P3_C$ timeout occurs while the ECU is in the process of receiving or transmitting a multi-frame USDT diagnostic message, the unsolicited service $20 positive response message shall be delayed until after the multi-frame message (reception or transmission) has completed.

8.5.2 Request Message Definition (Table 75).

Table 75: Request Message Definition

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	returnToNormalMode Request Service Id	M	20	SIDRQ

8.5.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. Because this service is defined as a single operation, there are no sub-parameters required or valid for the request message.

8.5.2.2 Request Message Data Parameter Definition. There are no data parameters used by this service.

8.5.3 Positive Response Message Definition (Table 76).

Table 76: Positive Response Message Definition

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	returnToNormalMode Positive Response Service Id	M	60	SIDPR

8.5.3.1 Positive Response Message Data Parameter Definition. There are no data parameters used by this service in the positive response message.

8.5.4 Supported negative response codes (RC_). The following negative response codes (Table 77) shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 77: ReturnToNormalMode Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	SubFunctionNotSupported-InvalidFormat This code is returned if the request message format is incorrect (too many request message bytes).	M	SFNS-IF

8.5.5 ReturnToNormalMode Message Flow Example. The following message flow example (Table 78) shows what happens when a tester transmits a $20 ReturnToNormalMode request while diagnostic periodic data is active ($AA service) and one-shot DTC information is being returned by the node in response to earlier tester requests. Since the DTC response messages are one-shot data, the node is able to continue sending the single shot UUDT frames after receiving the $20 request. Refer to the ReadDiagnosticInformation section for more details regarding the operation of service $A9.

Since the information encapsulated in DPID $FE was scheduled, no further updates are received after the service $20 acknowledgement (although there is a residual message $FE response between the service $20 request and positive acknowledgement. This is because the $FE DPID message was already in the protocol device at the time the $20 request was received).

Refer to service $AA ReadDataByPacketIdentifier for more details regarding the operation of service $AA, and the Flash Programming section of this specification for more details about how mode $20 is used to end a programming session.

The following assumptions are made in the example below:

1. The tester had previously requested the node to report DTCs with a request for the $A9 service with a $81 value in the sub-parameter field. The response to this request is in progress but has not completed at the time the $20 request is received.
2. The tester had previously requested the node to transmit DPID $FE periodically.
3. The node has loaded into the protocol device (but not yet sent) a periodic update of DPID $FE when it received the mode $20 request.
4. The node has a USDT physical request CAN ID of $241.
5. The node has a UUDT physical response CAN ID of $541.
6. The node has a USDT physical response CAN ID of $641.

Table 78: ReturnToNormalMode Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
N(UUDT-SF)	541	FE	$xx	$xx	$yy	$zz	$aa	$bb	$bb
N(UUDT-SF)	541	81	16	35	02	63	---	---	---
N(UUDT-SF)	541	FE	$xx	$xx	$yy	$zz	$aa	$bb	$bb
N(UUDT-SF)	541	81	04	35	0E	23	---	---	---
T(USDT-SF)	241	01	20	---	---	---	---	---	---
N(UUDT-SF)	541	FE	$xx	$xx	$yy	$zz	$aa	$bb	$bb
N(USDT-SF)	641	01	60	---	---	---	---	---	---
N(UUDT-SF)	541	81	11	00	02	23	---	---	---
N(UUDT-SF)	541	81	15	01	0E	23	---	---	---

8.5.6 Node Interface Function.

8.5.6.1 Node Interface Data Dictionary (Table 79).

Table 79: Node Interface Data Dictionary of ReturnToNormalMode Service Pseudo Code

Variable Meaning	Values
manufacturers_enable_counter **message_data_length** **Diag_Services_Disable_DTCs** **DTCs_Enabled_In_Device_Control** **request_DeviceControl_exit** **Dev_Cntrl_Active** **Security_Access_Unlocked** **DTC_send_on_change_flag** **vulnerability_flag** **normal_message_transmission_status** **programming_mode_active** **high_speed_mode_active** **programming_mode_entry_OK** **high_speed_mode_entry_OK** **diagnostic_responses_enabled** **TransferData_Allowed** **DeviceContol_Security_Level** **TesterPresent_Timer_State** **TesterPresent_Timer**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.5.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $20 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_20_Msg_Recvd()
/**********************************************************************************
* Handle Negative Responses
**********************************************************************************/
IF (message_data_length != 1) THEN
send ($7F $20 $12) /* SubFunctionNotSupported - InvalidFormat */
/*********************************************************************************
* Handle Good Requests
*********************************************************************************/
ELSE
CALL Exit_Diagnostic_Services() /* procedure defined below to gracefully exit
                                    all active diagnostic services */
ENDIF
ENDFUNCTION
```

Each time a valid $20 message is received, or if a TesterPresent timeout occurs, the following function is called:

```
BEGINFUNCTION Exit_Diagnostic_Services()
TesterPresent_Timer_State ← INACTIVE
TesterPresent_Timer ← 0
IF (programming_mode_active = NO) THEN
    /***************************************************************************
    * Release All Device Controls
    ***************************************************************************/
    IF (Dev_Cntrl_Active = YES)
        request_DeviceControl_exit ← YES
        CALL Background_DeviceControl_Logic() /* procedure defined in $AE */
    ENDIF
    /***************************************************************************
    * Reset Security Access
    ***************************************************************************/
    DeviceContol_Security_Level ← SERVICE
    Security_Access_Unlocked ← FALSE
    /***************************************************************************
    * Reset Message Scheduler (but retain dynamic DPID definitions)
    ***************************************************************************/
    CALL ClearAndDisablePDS() /* procedure defined in service $AA */
    /***************************************************************************
    * Reset Send-On-Change DTC Reporting
    ***************************************************************************/
    DTC_send_on_change_flag ← 0
    /***************************************************************************
    * Reset levels of service $10 that have not already been covered
    ***************************************************************************/
    DTCs_Enabled_In_Device_Control ← NO
    /***************************************************************************
    * Enable DTCs to set again
    ***************************************************************************/
    IF (Diag_Services_Disable_DTCs = TRUE) THEN
        Diag_Services_Disable_DTCs ← FALSE
        CALL Diag_App_Reset_Vars() /* local diagnostic application to reset flags, timers, and variables as
        appropriate */
    ENDIF
    /***************************************************************************
    * check if normal message transmission was disabled and reinitialize
    * communications if it was. In addition, reset mode $A5 flags that may have
    * been set prior to the mode $20 request or the P3C timeout.
    ***************************************************************************/
        IF (normal_message_transmission_status = DISABLED) THEN
```

```
            normal_message_transmission_status ← ENABLED /* cancel out mode $28 */
            programming_mode_entry_ok ← NO
            high_speed_mode_entry_ok ← NO
            CALL appl_Comm_Init() /*application function to do any necessary cleanup */
            CALL hnd_Invoke_Comm_Init() /* GMLAN Handler function executed in Comm Init state */
            CALL APPL_ACTIVATE_LOCALDIAGVN() /* Application function call to reactivate diagnostic local VN */
        ENDIF
        IF (permanent diagnostic CAN Identifiers are programmed) THEN
            send ($60) /* send acknowledgement */
        ELSE
            diagnostic_responses_enabled ← NO
        ENDIF
    ELSE
        TransferData_Allowed ← NO
        programming_mode_active ← NO
        /****************************************************************************
        * Reset CAN Protocol Device if high speed mode was active before doing S/W Reset
        ****************************************************************************/
        IF (high_speed_mode_active = YES)
            high_speed_mode_active ← NO
            CALL hnd_Init_CAN_Device() /* handler call to reinit the baud rate in order to prevent bus errors which could be caused if reset times differ greatly*/
        ENDIF
        /****************************************************************************
        * check to see if any diagnostic information which is retained across multiple
        * drive cycles needs to be reinitialized before performing a software reset
        ****************************************************************************/
        DTCs_Enabled_In_Device_Control ← NO
        IF (Diag_Services_Disable_DTCs = TRUE) THEN
            Diag_Services_Disable_DTCs ← FALSE
            CALL Diag_App_Reset_Vars() /* local diagnostic application to reset flags, timers, and variables as appropriate */
        ENDIF
        CALL Invoke_Sw_Reset() /* causes a software reset to occur */
    ENDIF
ENDFUNCTION
```

8.5.7 Node Verification Procedure.

Procedure 1:

1. While not in programming mode, transmit a $20 service request with extra information appended to the message resulting in an incorrect length and verify negative response ($7F $20 $12).
2. Repeat step 1 while programming mode is active and verify that the negative response ($7F $20 $12).is sent.

Procedure 2:

1. While in programming mode, transmit a service $20 request message using the functional all-nodes CAN Identifier ($101) + all-nodes extended address ($FE). Verify that there is no response sent.

Procedure 3:

1. While not in programming mode, transmit a service $20 request message while periodic DPIDs are scheduled. Verify the positive response and that the scheduled messages are halted.
2. If the node supports dynamic DPIDs, define a dynamic message number via service $2C, and configure the node to schedule the message periodically via service $AA. Verify that periodic responses are received. Next transmit a service $20 request to the node and verify that the just-defined periodic message(s) are halted.
3. Repeat the $AA request to schedule the dynamically defined message, and verify that the format of the dynamically defined message was not lost after the service $20 request.

Procedure 4: (Valid if the send on change DTC logic is supported by the node).

1. While not in programming mode, transmit a service $20 request message while the DTC send-on-change algorithm is activated. Verify the positive response and that the updated DTC count information is no longer reported by the node. (This can be done by causing a code to transition to a state that satisfies the send-on-change DTC status mask, and verifying that no information is reported by the node.)

Procedure 5:

1. While not in programming mode, transmit a service $20 request message while tester-requested device controls are active. Verify the positive response and that full control of inputs/outputs is returned to the ECU.

Procedure 6: (Valid if the SPS security access levels of the $27 service are supported).

1. While not in programming mode, transmit a service $20 request message to a node whose security is currently unlocked via service $27 SecurityAccess (with Manufacturers Enable Counter set to $00 and vulnerability flag not $FF). After the $60 positive response, verify via a service $27 $01 request that the node becomes locked (seed != $0000).
2. While not in programming mode, transmit a service $20 request to a node with a non-zero Manufacturers Enable Counter (MEC) and vulnerability flag not equal to $FF. After the $60 positive response, verify via a service $27 $01 request that the node unlocks (seed = $0000).
3. While not in programming mode, transmit a service $20 request to a node with a Vulnerability Flag set to $FF (if applicable) and MEC set to $00. After the $60 positive response, verify via a service $27 $01 request that the node unlocks (seed = $0000).

Procedure 7: (Valid if the Device Control security access levels of the $27 service are supported. All steps assume programming mode is not active)

1. Transmit a service $20 request to a node with a non-zero Manufacturers Enable Counter (MEC). After the $60 positive response, verify via a service $27 $03 request that the node becomes unlocked (seed = $0000).
2. Transmit a service $20 request message to a node with the MEC = $00 and at a time when device control security is currently unlocked via service $27 SecurityAccess. After the $60 positive response, verify via a service $27 $03 request that the node is locked (seed != $0000).

Procedure 8:

1. While not in programming mode, transmit a service $28 request to disable normal mode communications. After normal communications have been halted, transmit a $20 service request message to return the node back to normal communications. Verify that the positive response is sent. Transmit a diagnostic request that should result in a diagnostic response and verify that the response is transmitted by the ECU. Then have the operator initiate a vehicle function, which should result in VN activation and normal communication. Verify that the ECU starts transmitting the appropriate normal mode message.

Procedure 9:

1. Invoke a diagnostic service that results in DTCs being disabled ($10, $28, or $AE). Disconnect any input which would normally result in a DTC being set. Verify that the DTC does not set via the $A9 service. Send a $20 request message and verify the positive response and that the DTC will set after the appropriate conditions are met to set the DTC.
2. Repeat step 1 of this procedure for each supported service that can disable the setting of DTCs.

8.5.8 Tester Implications. When sending a request for this service to terminate a high speed programming event on the SWCAN link, the tester shall reinitialize its CAN protocol converter hardware to low speed operation within 30 ms after the request. In addition, the test device must wait at least 50 ms before attempting communication at low speed.

Note: The tester should wait for approximately 1 s before attempting normal communications after exiting a programming event in order to allow the nodes adequate time to perform a software reset and reinitialize communications.

8.6 ReadDataByParameterIdentifier ($22) Service. The purpose of the ReadDataByParameterIdentifier service is to allow a tester access to ECU data by requesting one or multiple Parameter Identifier(s) (PID). This service is intended to be used during a device's development cycle and for special test conditions. It is not intended to be used in lieu of service $AA for manufacturing and/or field service diagnostics.

8.6.1 Service Description. The ReadDataByParameterIdentifier service provides a means for a tester to request ECU data by Parameter Identifier (PID). A PID number is a unique 2-byte value that the ECU translates into a specified piece of data (e.g., ABS left front wheel speed, or engine rpm). The length (number of data bytes in the response message for a given PID) and scaling of the response data (associated with a PID) must be documented in a CTS, SSTS, supplemental diagnostic specification, or another document or database referenced by any of the proceeding documents. PID response data can range from one to seven bytes in length per PID (not including the PID number).

Note: The 7-byte maximum results from the fact that PIDs can be packed into DPIDs and requested via the $AA service. The $AA service uses UUDT responses. UUDT responses are limited to seven bytes. For the purpose of service $22, the 7-byte maximum refers to the maximum number of bytes in the dataRecord for each PID. Reference the positive response message definition table.

This service only provides a single USDT positive response to a request for one or multiple PIDs provided that the request message is properly formatted and at least one of the requested PIDs is supported in the ECU. If a tester needs to retrieve PID data periodically then the tester must first pack the PID (or PIDs) into a DPID via the $2C service and then request the data periodically via the $AA service.

If a tester requests multiple PIDs with a single request of this service, the ECU shall include data in a positive response for all of the PIDs that it supports. No data shall be included in a positive response for unsupported PIDs, or for secure PIDs unless security access has been granted (ECU is unlocked) via service $27. See paragraph 8.8 for more details on service $27.

A functional request for this service containing only unsupported PID(s) shall result in no response being sent. A physically addressed request shall result in a negative response if none of the requested PIDs are supported in the ECU.

The maximum number of PIDs that the ECU supports with a single request shall be documented in the ECU CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents.

This service can also be used to retrieve emission related PIDs as specified in SAE J1979/ISO 15031-5. Because SAE J1979/ISO 15031-5 uses a single byte PID number, a tool must request these emission PIDs with the most significant byte (MSB) of the PID number set to $00 when using this service, to ensure request message compatibility. See paragraph 8.6.5 for example.

8.6.2 Request Message Definition. (Table 80)

Table 80: ReadDataByParameterIdentifier Request Message

Data Byte	Parameter Name [Note 1]	Cvt	Hex Value	Mnemonic
#1	ReadDataByParameterIdentifier Request Service Id	M	22	SIDRQ
 #2 #3	parameterIdentifier #1 = [byte 1 (MSB) byte 2 (LSB)]	M	 00 thru FF 00 thru FF	PID_ B1 B2
 #4 #5	parameterIdentifier #2 = [byte 1 (MSB) byte 2 (LSB)]	U	 00 thru FF 00 thru FF	PID_ B1 B2
:	:	:	:	:
 #n-1 #n	parameterIdentifier #k = [byte 1 (MSB) byte 2 (LSB)]	U	 00 thru FF 00 thru FF	PID_ B1 B2

Note 1: MSB = Most Significant Byte, LSB = Least Significant Byte.

8.6.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. This service does not use a sub-function parameter.

8.6.2.2 Request Message Data Parameter Definition. Table 81 specifies the data parameter definitions for this service.

Table 81: Data Parameter Definition

Definition
ParameterIdentifier # (1 - k) This parameter identifies the ECU data that is being requested by the tester.

8.6.3 Positive Response Message Definition (Table 82).

Table 82: ReadDataByParameterIdentifier Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDataByParameterIdentifier Response Service Id	M	62	SIDPR
#2 #3	parameterIdentifier # (1 – k) = [byte 1 (MSB) byte 2 (LSB)]	M	00-FF 00-FF	PID_ B1 B2
#4 : #(m+3)	dataRecord # (1 – k) = [data#1 : data#m]	M : U	00-FF : 00-FF	PIDdata_ DATA_1 : DATA_m
#(m+4)	parameterIdentifier # (2 – k) = [byte 1 (MSB) byte 2 (LSB)]	C	00-FF : 00-FF	PID_ B1 B2
#(m+6) : #(z+m+5)	dataRecord # (2 – k) = [data#1 : data#z]	C : U	00-FF : 00-FF	PIDdata_ DATA_1 : DATA_z
:	:	:	:	:
#p	parameterIdentifier # (k) = [byte 1 (MSB) byte 2 (LSB)]	C : U	00-FF : 00-FF	PID_ B1 B2
#p+2 : #(x+p+1)	dataRecord # (k) = [data#1 : data#x]	C : U	00-FF : 00-FF	PIDdata_ DATA_1 : DATA_x

8.6.3.1 Positive Response Message Data Parameter Definition (Table 83).

Table 83: Response Data Parameter Definition

Definition
ParameterIdentifier # (1 thru k) This parameter is an echo of one of the supported parameterIdentifier from the request message.
dataRecord (#1 to #m,z,x) This parameter is used by the ReadDataByParameterIdentifier positive response message to provide the requested data record values to the tester. The length of this field is dependent upon the PID requested.

8.6.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 84) shall be implemented for this service.

Table 84: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This response code shall be sent if the length of the request message is invalid. 1. The message_data_length is less than 3 (need at least one PID requested + SID). 2. The number of data bytes in the request exceeds 2 times the maximum number of PIDs allowed with a single request + 1. 3. The number of bytes in the request message after the SID is not an even number.	M	SFNS_IF
22	**ConditionsNotCorrectOrRequestSequenceError** This response code shall be sent if the operating conditions of the ECU are such that it cannot perform the required action (e.g., the data for a PID is stored in a satellite device and the link from the host to the satellite device has failed). This response code is not intended to be used if the ECU conditions are such that the request temporarily cannot be performed (e.g., the data must be retrieved from another processor internal to the ECU via a System Packet Interface (SPI) bus and will not be available within the required P2$_c$ time). Response code $78 is used when conditions are temporarily not met. If a node supports response code $22, the reason(s) to generate this response code must be documented in the CTS, SSTS, or supplemental diagnostic specification referenced by either of the two preceding documents, and must not conflict with any of the other negative response codes defined for this service.	C	CNCRSE
31	**RequestOutOfRange** A negative response with this response code shall be sent only if the request for this service was sent using the ECU Physical Request CAN Identifier (see paragraph 4.4.1 for description of Physical Request CAN Identifier) and: 1. None of the PID(s) being requested is supported in the ECU. 2. All supported PID(s) being requested require security access and ECU security access has not been granted. If a request for this service is sent using the All Node Functional Request CANId (including a valid extended address) and either of the above conditions exist then there shall be no response sent. Reference the pseudo code in paragraph 8.6.6.2 for further clarification.	M	ROOR

Hex	Description	Cvt	Mnemonic
78	**RequestCorrectlyReceived-ResponsePending** This response code shall be sent if retrieving the data associated with the requested PID and transmitting the final response on the bus takes longer than $P2_c$ ms (e.g., the data to be read has to be retrieved from another processor via an ECU internal SPI bus).	C	RCR-RP

8.6.5 Message Flow Example ReadDataByParameterIdentifier Service. This section specifies the conditions to be fulfilled for the example to perform a ReadDataByParameterIdentifier service. The tester may request PID data at any time independent of the status of the server (as long as another USDT response is not pending currently).

8.6.5.1 Request OBD PID $0C - Engine RPM. In the example below (Table 85), the following is assumed true:

- The point to point request CAN Id for the ECU is $7E0.
- The USDT response CAN Id of the ECU is $7E8.
- The PID for engine rpm (as defined in SAE J1979/ISO 15031-5) is $0C (so this service requests with $000C). The data returned is 2 bytes with a transfer function of rpm = N/4.
- Engine Speed is assumed to be 750 rpm at the time of the request.

Table 85: ReadDataByParameterIdentifier (PID=000C hex)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$7E0	$03	$22	$00	$0C	---	---	---	---
N(USDT-SF)	$7E8	$05	$62	$00	$0C	$0B	$B8	---	---

8.6.5.2 Physically Request Multiple PIDs Resulting In Multi-Frame Response. In the example below (Table 86), the following is assumed true:

- The ECU supports retrieving at least three PIDs in a single request.
- The point to point request CAN Id for the ECU is $7E0.
- The USDT response CAN Id of the ECU is $7E8.
- The PID for Engine Coolant Temperature (as defined in SAE J1979/ISO 15031-5) is $05 (so this service requests with $0005). The data returned is a single byte with a transfer function of °C = (N - 40). This provides a temperature range from -40 to 215°C.
- Engine Coolant Temperature is assumed to be 92°C at the time of the request.
- The PID for engine rpm (as defined in SAE J1979/ISO 15031-5) is $0C (so this service requests with $000C). The data returned is two bytes with a transfer function of rpm = N/4.
- Engine Speed is assumed to be 750 rpm at the time of the request.
- The time since engine start is defined in PID $001F contains two bytes of data with a scaling of 1 s per count.
- It is assumed that the engine has been running for 200 s.

Table 86: ReadDataByParameterIdentifier(PID=0005 hex, PID=000C hex, PID=001F hex)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$7E0	$07	$22	$00	$05	$00	$0C	$00	$1F
N(USDT-FF)	$7E8	$10	$0C	$62	$00	$05	$84	$00	$0C
T(USDT-FC)	$7E0	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$7E8	$21	$0B	$B8	$00	$1F	$00	$C8	---

8.6.5.3 Functionally Request OBD PID $0C - Engine RPM. In the example below (Table 87), the following is assumed true:

- The bus has two ECUs on it.
- Both ECUs support this service.
- ECU 1 supports PID $0C and ECU 2 does not.
- The point to point request CAN Id for the ECU 1 is $7E0.
- The USDT response CAN Id of the ECU 1 is $7E8.
- The point to point request and response CAN Identifiers for the ECU 2 are not detailed here as ECU 2 will send no response because the request is functional and the PID is not supported.
- The PID for engine rpm (as defined in SAE J1979/ISO 15031-5) is $0C (so this service requests with $000C). The data returned is 2 bytes with a transfer function of rpm = N/4.
- Engine Speed is assumed to be 750 rpm at the time of the request.

Table 87: Functional ReadDataByParameterIdentifier(PID=000C hex)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	03	$22	$00	$0C	---	---	---
N1(USDT-SF)	$7E8	$05	$62	$00	$0C	$0B	$B8	---	---
N2(No Resp)									

8.6.5.4 Functionally Request PID Resulting In Multi-Frame Response. In the example below (Table 88), the following is assumed true:

- The vehicle bus has multiple ECUs but only one ECU supports the requested PID.
- The PID requested is $FF0D (this is only an example).
- PID $FF0D is defined to be seven bytes in length.
- The point to point request CAN Id for the responding ECU is $7E0.
- The USDT response CAN Id of the responding ECU is $7E8.

Table 88: Functional ReadDataByParameterIdentifier(Multi-Frame Response)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	03	$22	$FF	$0D	---	---	---
N(USDT-FF)	$7E8	$10	$0A	$62	$FF	$0D	$aa	$bb	$cc
T(USDT-FC)	$7E0	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$7E8	$21	$dd	$ee	$ff	$gg	---	---	---

8.6.5.5 Functionally Request Two PIDs Resulting In Response from Multiple ECUs. In the example below (Table 89), the following is assumed true:

- The vehicle bus has two ECUs that support this service and both ECUs support requesting at least two PIDs in a single request.
- The point to point request CAN Id for ECU 1 is $7E0 and the USDT response CAN Id is $7E8.
- The point to point request CAN Id for ECU 2 is $7E2 and the USDT response CAN Id is $7EA.
- The PID for engine rpm (as defined in SAE J1979/ISO 15031-5) is $0C (so this service requests with $000C). The data returned is two bytes with a transfer function of rpm = N/4.
- Engine Speed is assumed to be 750 rpm at the time of the request.
- The PID for Transmission range select (PRNDL) is PID $194F.
- This PID has one byte of response data where $00 = illegal range, $01 = Park, $02 = Reverse, $04 = Neutral, $06 = Drive 6, $08 = Drive 5, $10 = Drive 4, $20 = Drive 3, $40 = Drive 2, and $80 = Drive 1.
- The gear select position is assumed to be park for this example.
- ECU 1 supports PID $000C but not PID $194F.
- ECU 2 supports PID $194F but not $000C.

Table 89: Functional ReadDataByParameterIdentifier (Multi-PID Request)

T = Frame Sent By Tester; N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	03	$22	$00	$0C	$19	$4F	---
N(USDT-SF)	$7E8	$05	$62	$00	$0C	$0B	$B8	---	---
N(USDT-FF)	$7EA	$04	$62	$19	$4F	$01	---	---	---

8.6.6 Node Interface Function.

8.6.6.1 Node Interface Data Dictionary (Table 90).

Table 90: Node Interface Data Dictionary of InitiateDiagnosticOperation Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **message_address_type** **Security_Access_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

Variable/Meaning	Values
max_22_Msglength This is a local flag that is set to 2 times the max number of PIDs supported in a single request and then add 1 (to accommodate the SID byte).	1 + 2(number of PIDs allowed in Single request)
Srv_22_NRC This is a local flag that is used to keep track of the negative response code if a negative response needs to be sent	$00, $12, $22, $31, $78
Num_PIDs_Requested This is variable that contains the number of PIDs in the request message. This is calculated by taking the message length, subtracting 1, and then dividing by 2.	1 to number of PIDs allowed in a single request
ReqPIDArray[] This is an array that contains the PID number from the request message.	N/A
Process_Request This is a local flag used to keep track of whether or not a positive response is to be generated based on the request message.	YES/NO
PidStructArray[] This is an array of structures that contain the data needed to support PIDs. for the purposes of the pseudo code of this service it is assumed that each structure contains the following information: 1. PidNumber - The PID number. 2. Rc78Needed - Set to YES if it is necessary to send a $7F $SID 78 response and NO if no $7F$SID $78 is needed. 3. PidSecure - Set to YES if security access is required to retrieve PID data. 4. PidSecurityCode - Set to YES if security code access is required to retrieve PID data. Other elements may exist in the structure such as a pointer to a function to retrieve the data, memory address and length information (for a dynamic PID, see $2D service), etc. Only the PidNumber, Rc78Needed, PidSecure, and PidSecurityCode are shown here for the purposes of the pseudo code. The pseudo code uses the following convention to access an element of the structure: PidStructArray[0].PidNumber would be the PID number of the first PID structure in the array.	N/A
ReqIndex This is a local flag used for indexing through the array of PIDs in the request array.	0 to (Num_PIDs_Requested-1)
NumArrayElements This is a local flag set to the number of elements in the PidStructArray.	ECU dependent
Index This is a local flag used to index through the array of PID structures when determining if the PID requested is valid	0 to (NumArrayElements -1)

8.6.6.2 Node Interface Pseudo Code.

Powerup States:

None

The following logic is executed when a request message with ServiceId $22 is received by the ECU:

```
BEGIN FUNCTION Serv_22_Msg_Recvd()
IF ((message_data_length < 3) OR ((message_data_length > max_22_Msglength) OR
((message_data_length MOD 2) = 0)) THEN
    Srv_22_NRC ← $12
ELSE
    Srv_22_NRC ← $00
    Num_PIDs_Requested ← ((message_data_length - 1)/2)

    /* Populate ReqPIDArray[] with PID numbers from request message. There is no eloquent way to show
    this in pseudo code so it is assumed in the pseudo code that it gets done through typecasting or some
    type of pointer manipulation (language dependant) */

    Process_Request ← FALSE

    FOR (ReqIndex ← $00 TO (Num_PIDs_Requested - 1) BY $01)
        /* loop through array of PID structures to see if requested PID is valid */
        FOR (Index ← $00 TO (NumArrayElements - 1) BY $01)
            IF (ReqPIDArray[ReqIndex] = PidStructArray[Index].PidNumber) THEN
                IF ((PidStructArray[Index].PidSecure = NO OR Security_Access_Unlocked = TRUE) AND
                (PidStructArray[Index].PidSecurityCode = NO OR Security_Access_Allowed = TRUE)) THEN
                    Process_Request ← TRUE
                    IF (PidStructArray[Index].Rc78Needed = YES) THEN
                        Srv_22_NRC ← $78
                    ENDIF
                    Index ← NumArrayElements /* to exit for loop */
                ENDIF
            ENDIF
        ENDFOR
    ENDFOR

    IF (Process_Request = TRUE) THEN
        IF (Srv_22_NRC = $78) THEN
            Srv_22_NRC ← $00
            send ($7F $22 $78)
        ENDIF

        /* obtain PID data - The pseudo code does not dictate how this is done. For example, the data may be
        retrieved via a call to a function whose pointer is contained in the PID structure. Another example
        would be to obtain the data via a global variable or by accessing memory and length information for a
        dynamic PID. This is not intended to be the comprehensive list on how to do this */

        FOR (ReqIndex ← $00 TO (Num_PIDs_Requested - 1) BY $01)
            FOR (Index ← $00 TO (NumArrayElements - 1) BY $01)
```

```
                IF ((ReqPIDArray[ReqIndex] = PidStructArray[Index].PidNumber) AND
                ((PidStructArray[Index].PidSecure = NO) OR
                 (Security_Access_Unlocked = TRUE)) AND ((PidStructArray[Index].PidSecurityCode = NO)
                OR (Security_Access_Allowed = TRUE))) THEN
                    attempt to obtain pid data
                    IF (operating conditions did not allow PID data to be successfully retrieved) THEN
                        Srv_22_NRC ← $22
                        Index ← NumArrayElements /* to exit inner for loop */
                        ReqIndex ← Num_PIDs_Requested /*exit outer for loop */
                    ELSE
                        copy PID number and data into response buffer
                        Index ← NumArrayElements /* to exit for loop */
                    ENDIF
                ENDIF
            ENDFOR
        ENDFOR
    ELSE
        Srv_22_NRC ← $31
    ENDIF
ENDIF

IF (Srv_22_NRC != $00) THEN
    IF ((message_address_type = PHYSICAL) OR (Srv_22_NRC != $31)) THEN
        send ($7F $22 $Srv_22_NRC)
    ENDIF
ELSE IF (Process_Request = TRUE) THEN
    send ($62, PID#_MSB, PID#_LSB, data…….)
ENDIF

ENDFUNCTION
```

8.6.7 Node Verification Procedure.

Procedure 1:

1. Send a physically addressed $22 message for one (unsecure) $PID that is supported by the ECU. Verify proper response, containing the data of the requested $PID (test data formatting).
2. Send a physically addressed $22 message for a single $PID where that $PID is not supported by the ECU. Verify the negative response ($7F $22 $31 - Request Out Of Range).
3. If applicable, send a physically addressed $22 message for one secure $PID (that is supported by the ECU) at a time when the ECU is secure. Verify the negative response ($7F $22 $31 - Request Out Of Range).
4. If applicable, unlock the ECU via the security access service then repeat the request in the previous procedure and verify the appropriate positive response and data content.
5. Send a physically addressed $22 request message for multiple $PIDs (if the ECU supports multiple PIDs in a single request), where at least one, but not all, of the requested $PIDs is supported by the ECU and does not require security access. Ensure that the number of PIDs in the request is less than or equal to the maximum number supported in a single request. Verify proper response, containing only the data of the requested $PID(s) which are supported by the ECU and do not require security access.
6. Send a physically addressed $22 request message for multiple $PIDs (if the ECU supports multiple PIDs in a single request), where all of the requested $PIDs are supported by the ECU (and do not require security access). Ensure that the number of PIDs in the request is less than or equal to the maximum

number supported in a single request. Verify proper response, containing data for all of the requested $PID(s).

7. If applicable, repeat the last procedure but include a secure PID in the request. Ensure that this is done at a time when security access has not been granted. Verify the proper response for the unsecure PID(s) and that no data is reported for the secure PID.
8. If applicable, use service $27 to access security and then repeat the previous procedure. Verify that the proper response is sent including data for all PIDs in the request.
9. Send a physically addressed $22 message with less than two bytes in the $PID field (0 or 1 data byte after the service indentifier $22). Verify the negative response ($7F $22 $12 - SubFunctionNotSupported-InvalidFormat).
10. Send a physically addressed $22 message with more data bytes than allowed per the maximum request message length (see pseudo code above). Verify the negative response ($7F $22 $12 - SubFunctionNotSupported-InvalidFormat).
11. Send a physically addressed $22 message for multiple $PIDs (if the ECU supports multiple PIDs in a single request) where all requested $PIDs are not supported by the ECU. Verify the negative response ($7F $22 $31 - Request Out Of Range).
12. If the ECU supports the use of negative response code $78, send a $22 message to the ECU with a valid $PID where the ECU needs more than $P2_C$ ms to read data bytes from memory and send the positive response. Verify that the ECU sends the negative response ($78 - RequestCorrectlyReceived-ResponsePending) within $P2_C$ ms followed by a positive response (reference application timing section of this specification).
13. (If applicable) Send a request for a valid $PID at a time when ECU internal conditions would not allow the data to be retrieved (e.g., EEPROM failure) and verify that the ECU sends the correct negative response ($7F $22 $22 ConditionsNotCorrect)).
14. If applicable, send a physically addressed $22 message for one security code (as defined in the Vehicle Theft Deterrent SSTS) required $PID (that is supported by the ECU) at a time when the security code has not been entered. Verify the negative response ($7F $22 $31 - Request Out Of Range).
15. If applicable, enter the security code then repeat the request in the previous procedure and verify the appropriate positive response and data content.
16. If applicable, send a physically addressed $22 request message for multiple $PIDs (if the ECU supports multiple PIDs in a single request), where at least one, but not all, of the requested $PIDs is supported by the ECU and include a security code required PID in the request. Ensure that this is done at a time when the security code has not been entered. Verify the proper response for the unsecure PID(s) and that no data is reported for the secure PID.
17. If applicable, enter the security code and then repeat the previous procedure. Verify that the proper response is sent including data for all PIDs in the request.

Procedure 2:

1. Send functionally addressed $22 request message for one $PID that is supported by the ECU and does not require security access (use $101 AllNode CAN Id and $FE AllNode functional system). Verify proper response, containing the data of the requested $PID (test data formatting).
2. If applicable, send a functionally addressed $22 message for a $PID that is supported by the ECU, where the PID is a secure $PID and the ECU is locked. Verify there is no response.
3. If applicable, use service $27 to access security then repeat the request from the previous step. Verify the proper positive response and data.
4. Send functionally addressed $22 request message for two $PIDs (if the ECU supports multiple PIDs in a single request) where only one of the requested PIDs is supported by the ECU and does not require security access (use $101 AllNode CAN Id and $FE AllNode functional system). Verify proper response, containing the only the data of the supported unsecure $PID.
5. If applicable, repeat the last procedure but include a secure PID in the request. Ensure that this is done at a time when security access has not been granted. Verify the proper response for the unsecure PID and that no data is reported for the secure PID.

6. If applicable, use service $27 to access security and then repeat the previous procedure. Verify that the proper response is sent including data for all PIDs in the request.
7. Send a functionally addressed $22 message with less than 2 bytes in the $PID field (0 or 1 data byte after the service indentifier $22). Verify the negative response ($7F $22 $12 - SubFunctionNotSupported-InvalidFormat).
8. Send a functionally addressed $22 message with more data bytes than allowed per the maximum request message length (see pseudo code above). Verify the negative response ($7F $22 $12 - SubFunctionNotSupported-InvalidFormat).
9. Send a correctly formatted functionally addressed $22 message for a $PID that is not supported by the ECU. Verify there is no response.
10. If the ECU supports the use of negative response code $78, send a functionally requested $22 message to the ECU with a valid $PID where the ECU needs more than $P2_C$ ms to read data bytes from memory and send the positive response. Verify that the ECU sends the negative response ($78 - RequestCorrectlyReceived-ResponsePending) within $P2_C$ ms followed by a positive response (reference application timing section of this specification).
11. (If applicable) Send a functional request for a valid $PID at a time when ECU internal conditions would not allow the data to be retrieved (e.g., EEPROM failure) and verify that the ECU sends the correct negative response ($7F $22 $22 ConditionsNotCorrect).
12. If applicable, send a functionally addressed $22 message for a $PID that is supported by the ECU, where the PID requires a security code (as defined in the Vehicle Theft Deterrent SSTS) and the security code has not been entered. Verify there is no response.
13. If applicable, enter the security code then repeat the request from the previous step. Verify the proper positive response and data.
14. If applicable, send functionally addressed $22 request message for two $PIDs (if the ECU supports multiple PIDs in a single request) where only one of the requested PIDs is supported by the ECU (use $101 AllNode CAN Id and $FE AllNode functional system), but include a secure code required PID in the request. Ensure that this is done at a time when the security code has not been entered. Verify the proper response for the unsecure PID and that no data is reported for the secure PID.
15. If applicable, enter the security code and then repeat the previous procedure. Verify that the proper response is sent including data for all PIDs in the request.

8.6.8 Tester Implications. This service can result in responses that are single frame or multiple frame based on the PID(s) requested. The tester should not functionally address requests for this service which result in a multiple frame response(s) unless it can handle sending the flow control frame(s) to each responding node within the allowable network timing parameters. The tester must be careful when requesting multiple PIDs functionally. If any ECU only allows requesting a single PID per request message then that ECU would generate a negative response even if it supported one of the PIDs in the request message. In addition, functionally addressed requests for this service should not contain more than two PIDs as this would generate a multi-frame request which would result in the transmission of flow frames from every ECU on the subnet.

8.7 ReadMemoryByAddress ($23) Service. The purpose of this service is to retrieve data from a contiguous range of ECU memory addresses. The range of ECU addresses is defined by a starting memory address parameter and a length (memory size) parameter included in the request message. This service is intended to be used during a device's development cycle to allow access to data that may not be available via another diagnostic service. The ReadMemoryByAddress service is only available as a one shot request-response service.

8.7.1 Service Description. The ReadMemoryByAddress service allows a tester to request the ECU to provide the data content of a range of ECU memory. The memory data to be read is identified by the parameters memoryAddress (MA) and memorySize (MS). The size of the memoryAddress parameter in the request message can be 2, 3, or 4 bytes in length depending on the ECU internal addressing scheme. The number of bytes that a tester shall use in the memoryAddress portion of the request message for a given ECU shall be documented in the appropriate CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents. The memorySize parameter in the request message shall always be two bytes.

The ReadMemoryByAddress positive response message includes the start address (MA), specified in request message, and requested memory data. Memory data shall be sent in ascending order beginning with the memory location determined by the memoryAddress parameter.

This diagnostic service shall be used to read RAM, Calibrations and unprotected areas of EEPROM/Flash memory. A device may choose to deny access to an address or a range of addresses. This allows production ECUs to protect their operating system and other secure data.

An ECU may restrict the testers ability to read specific addresses or ranges of addresses based on the security status of the ECU. If any of the addresses (that fall in the range of the request message) have security restrictions then the request shall be rejected unless the tester had previously successfully accessed security.

Note: Security access is only required if the ECU is locked based on the status of the MEC and/or the vulnerability flag see the SecurityAccess ($27) service for more information.

8.7.2 Request Message Definition (Table 91).

Table 91: ReadMemoryByAddress Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadMemoryByAddress Request Service Id	M	23	SIDRQ
	memoryAddress = [		xx	MA_
#2	memoryAddress (Byte1 - MSB),	M		B1
#3	memoryAddress (Byte2),	M		B2
#4	memoryAddress (Byte3),	C1		B3
#5	memoryAddress (Byte4)] **Note 1**	C2		B4
	memorySize = [	M	xx	MS_
#4/5/6	memorySize(High Byte),			B1
#5/6/7	memorySize(Low Byte)]			B2
C1: Byte 3 is present when the memoryAddress parameter contains 3 or 4 bytes. **C2:** Byte 4 is present when the memoryAddress parameter contains 4 bytes.				

Note 1: Byte1 in the memoryAddress parameter is always the most significant byte of the address and the last byte of the memoryAddress parameter (Byte2 for 2-byte addressing, Byte3 for 3 byte addressing, or Byte4 for 4 byte addressing) is always the least significant byte of the memoryAddress parameter.

8.7.2.1 Request Message Sub-Function $Level Parameter Definition. Because this service is defined as a single operation, there are no sub-function levels required or valid for the request message.

8.7.2.2 Request Message Data Parameter Definitions (Table 92).

Table 92: Request Message Data Parameter Definition

Definition
memoryAddress The parameter memoryAddress is used to select the starting address of ECU memory from which data is to be retrieved. A portion of the memoryAddress parameter (most significant bits or bytes) can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor ECU with 16-bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device). In this case, a 3-byte memoryAddress parameter can be specified and the most significant byte of the memoryAddress parameter could then be used to select which physical memory device the tester is attempting to retrieve data from. Any usage of a memoryIdentifier shall be documented in the appropriate CTS, SSTS or supplemental diagnostic specification referenced by the CTS or SSTS.
memorySize The parameter memorySize is used to select the number of consecutive memory addresses (in ascending order) to be read starting at memoryAddress (memorySize includes the starting location). The contents of the range of addresses defined by memoryAddress and memorySize are provided in the positive response message. memorySize shall have a value between 1 and 4092 bytes if a 2-byte memoryAddress parameter is used (4091 if 3-byte addressing is used and 4090 if 4-byte addressing is used). The maximum value comes from the maximum number of bytes allowed by the USDT protocol, less the bytes for the response service Identifier and the address information from the request message which is included in the response. An ECU may choose to limit the maximum value of memorySize based on ECU resources (e.g., the size of the Network Layer Buffer). If an ECU limits the maximum value of the memorySize parameter, then the maximum allowed value shall be documented in the appropriate CTS or SSTS.

8.7.3 Positive Response Message Definition (Table 93).

Table 93: ReadMemoryByAddress Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadMemoryByAddress Positive Response Service Id	M	63	SIDPR
	memoryAddress = [		xx	MA_
#2	memoryAddress (Byte1 - MSB),	M		
#3	memoryAddress (Byte2),	M		
#4	memoryAddress (Byte3)	C3		
#5	memoryAddress (Byte4)] Note 1	C4		
	memoryDataRecord = [	M	xx	MDREC
#4/5/6	memoryData #1,			
#5/6/7	memoryData #2,			
:	:			
#n	memoryData #m]			
C3: Byte 3 is present when the memoryAddress parameter contains 3 or 4 bytes. C4: Byte 4 is present when the memoryAddress parameter contains 4 bytes.				

Note 1: Byte1 in the memoryAddress parameter is always the most significant byte of the address and the last byte of the memoryAddress parameter (Byte2 for 2-byte addressing, Byte3 for 3-byte addressing, or Byte4 for 4-byte addressing) is always the least significant byte of the memoryAddress parameter.

8.7.3.1 Positive Response Message Data Parameter Definitions (Table 94).

Table 94: Request Message Data Parameter Definition

Definition
memoryAddress The parameter memoryAddress is an echo of the memoryAddress value contained in the request message identifying the starting memory location address of the memoryDataRecord[] bytes which follow.
memoryDataRecord[] The memoryDataRecord[] shall contain the actual data values contained in the sequence of memory locations defined by the memoryAddress and memorySize parameters.

8.7.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 95) shall be implemented for this service.

Table 95: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** Request message is not the correct number of bytes (3 + number of bytes used for memoryAddress parameter).	M	SFNS-IF
22	**conditionsNotCorrectOrRequestSequenceError** This response code shall be sent if the operating conditions of the ECU are such that it cannot perform the required action. (e.g., one or more of the address(es) requested is stored in a satellite device and the link from the host to the satellite device has failed). This response code is not intended to be used if the ECU conditions are such that the request temporarily cannot be performed (e.g., the data must be retrieved from another processor internal to the ECU via an SPI bus and would not be available within the required P2$_C$ time). Response code $78 is used when conditions are temporarily not met. If a node supports this response code, the reason(s) to generate this response code must be documented in the CTS, SSTS, or supplemental diagnostic specification referenced by either of the two preceding documents.	C	CNCORSE
31	**RequestOutOfRange** Note 1 • Any memory address within the interval [$MA, ($MA + $MS -$1)] is invalid. • Any memory address within the interval [$MA, ($MA + $MS -$1)] is restricted. • Any memory address within the interval [$MA, ($MA + $MS -$1)] is secure and the ECU is locked. • The memorySize parameter in the request message is greater than the maximum value supported by the ECU.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** See 7.2 Return Code Definition.	C	RCR-RP

Note 1: A restricted memory address is one that the ECU will not allow a tester to read under any circumstances (e.g., the addresses that contain the security seed and key values. Secure memory addresses are those which the ECU shall not allow the tester to read unless the ECU is unlocked (see service $27 SecurityAccess).

8.7.5 Message Flow Example ReadMemoryByAddress.

8.7.5.1 ReadMemoryByAddress Example with 4 Byte memoryAddress. In this example (Table 96):

The ECU uses 4 byte addressing and the tester requests memory data in addresses $00011102 to $00011103 (two bytes memory data so memorySize is $00 $02).

The tester responds with the memory data $34 and $56.

The node has a physical request CANId of $241.

The node has a physical USDT response CANId of $641.

Table 96: ReadMemoryByAddress(memoryAddress, memorySize < 4 bytes)

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$07	$23	$00	$01	$11	$02	$00	$02
N(USDT-SF)	$641	$07	$63	$00	$01	$11	$02	$34	$56

8.7.5.2 ReadMemoryByAddress Example with 3 Byte memoryAddress. In this example (Table 97), it is assumed:

The ECU uses 3-byte addressing and the tester requests memory data in addresses $011102 to $011110 (15 bytes of memory data so memorySize is $00 $0F).

The ECU responds with the memory data $34 (data in start address), $56, $78, $9A, $BC, $DE, $F0, $12, $34, $56, $78, $9A, $BC, $DE and $F0.

The node has a physical request CANId of $241.

The node has a physical USDT response CANId of $641.

Flow control is supported and flow control parameters BS and ST are assumed to be $00.

Table 97: ReadMemoryByAddress (memoryAddress, memorySize > 3 bytes)

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$06	$23	$01	$11	$02	$00	$0F	---
N(USDT-FF)	$641	$10	$13	$63	$01	$11	$02	$34	$56
T(USDT-FC)	$241	$30	$00	$00	---	---	---	---	---
N(USDT-CF)	$641	$21	$78	$9A	$BC	$DE	$F0	$12	$34
N(USDT-CF)	$641	$22	$56	$78	$9A	$BC	$DE	$F0	---

8.7.6 Node Interface Function.

8.7.6.1 Node Interface Data Dictionary (Table 98).

Table 98: Node Interface Data Dictionary of ReadMemoryByAddress Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
Bad_Address Status flag that indicates whether any of the range of addresses in the request is not valid.	TRUE/FALSE
memoryAddress This is the starting address in ECU memory from the request message.	Varies based on ECU
MemorySize This is the number or consecutive memory addresses that the tester has requested (parameter of request message).	$01 to maximum size, where maximum size is the lesser of the NWL buffer size or 4094 minus the number of bytes in the memoryAddress parameter
Memory_Address This is a local variable used to point to ECU memory when indexing through the requested addresses to determine if any portion of the requested memory is invalid, protected, or requires security access. The pseudo code uses Memory_Address.data to reflect the contents of a given memory address.	Varies based on ECU
memoryDataRecord[] This is the buffer that stores the contents of the memory addresses from the request message for transmission in the response.	N/A
Index This is an index into the buffer for the response data	0 to (messageSize -1)

8.7.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $23 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_23_Msg_Recvd()
Bad_Address ← FALSE
IF (message_data_length is invalid) THEN /* 3 + number of address bytes */
    send Negative Response ($7F $23 $12) /*Subfunction Not Supported or Invalid Format */
ELSE IF ((memorySize = 0d) OR (memorySize > max value supported by the ECU) OR
(memoryAddress is not a valid address in ECU memory) OR
((memoryAddress + memorySize - 1) is not a valid address in ECU memory)) THEN
    send Negative Response ($7F $23 $31) /*Request Out Of Range */
ELSE
    Index ← $00
    IF (the response cannot be sent within P2C ms) THEN
        send Negative Response ($7F $23 $78) /*RequestCorrectlyReceived-ResponsePending */
    ENDIF
    \* Check whether any memory address in the requested interval is protected, restricted (by Security
    Status) or invalid*/
    FOR (Memory_Address ← messageAddress TO (messageAddress + memorySize- $01) BY $01)
        IF ((Memory_Address is restricted) OR (Memory_Address is Invalid) OR
        ((Memory_Address is secure) AND (Security_Access_Unlocked = FALSE))) THEN
            Bad_Address ← TRUE
            Memory_Address ← (messageAddress + memorySize) /* exit for loop */
            send Negative Response ($7F $23 $31) /*Request Out Of Range */
        ELSE
            IF (operating conditions do not allow access to memory address)
                send Negative Response ($7F $23 $22) /* Condition Not Correct */
                Memory_Address ← (messageAddress + memorySize) /*exit for loop*/
                Bad_Address ← TRUE
            ELSE
                memoryDataRecord[Index] ← Memory_Address.data
                Index ← (Index + 1)
            ENDIF
        ENDIF
    ENDFOR
    IF (Bad_Address = FALSE) THEN
         /* send positive response message */
        send ($63 $MA_B1 $MA_B2 ... $memoryDataRecord[0]... $memoryDataRecord[m])
    ENDIF
ENDIF
ENDFUNCTION
```

8.7.7 Node Verification Procedure.

Procedure 1:

1. Request one valid memory address which is not restricted, and non-secured (MS = $0001). Verify the proper positive response.
2. Request the maximum number of consecutive addresses allowed by the ECU (max value is the lesser of the NWL buffer size or 4094, minus the number of bytes in the memoryAddress parameter) starting at a memory address which will result in all addresses being valid, not restricted, and not secure. Verify the proper positive response.
3. If applicable, send a request to an unlocked ECU for an address range that is completely valid, not restricted, and includes secure locations. Verify the proper positive response.

Procedure 2:

1. Send a request with less than the required number of data bytes (3 + the number of bytes in the memoryAddress parameter), verify negative response ($7F $23 $12).
2. Send a request with more than the required number of data bytes (3 + the number of bytes in the memoryAddress parameter), verify negative response ($7F $23 $12).
3. Send a request with a valid SA and a MS that results in an invalid ECU address. Verify negative response ($7F $23 $31).
4. Send a request with valid SA and MS = 0. Verify the negative response ($7F $23 $31).
5. Send a request with a MS greater than the maximum MS supported by the ECU. Verify the negative response ($7F $23 $31).
6. If applicable, send a request with a valid range of non secure addresses that includes at least one restricted address. Verify negative response ($7F $23 $31).

7. If applicable, send a request to an ECU with a manufacturers enable counter = $00 and a vulnerability flag < $FF, with a range of valid and non restricted addresses that includes at least one secure address. Verify negative response ($7F $23 $31).
8. If applicable, send a request to an ECU with a manufacturers enable counter > $00 or vulnerability flag = $FF when security has not been accessed (SecurityAccess ($27) request has not been sent), with a range of valid and non restricted addresses that includes at least one secure address. Verify negative response ($7F $23 $31).
9. If applicable, send a request to an ECU with a manufacturers enable counter > $00 or vulnerability flag = $FF when security has been accessed (SecurityAccess ($27) request has been sent), with a range of valid and non restricted addresses that includes at least one secure address. Verify positive response.
10. If negative response code $78 is supported by the ECU, then create the conditions under which the ECU should return the $7F $23 $78 response and verify that proper response is sent. Repeat for each possible reason an ECU would send the negative response with response code $78.
11. If negative response code $22 is supported by the ECU, then create the conditions under which the ECU should return the $7F $23 $22 response and verify that proper response is sent. Repeat for each possible reason an ECU would send the negative response with response code $22.

8.7.8 Tester Implications. The tester needs to ensure that it transmits the correct number of bytes in the memoryAddress parameter or the message shall be rejected.

8.8 SecurityAccess ($27) Service. The purpose of this service is to provide a means to access data and/or diagnostic services which have restricted access for security, emissions, or safety reasons. Diagnostic modes for downloading routines or data into a node and reading specific memory locations from a node are situations where security access may be required. Improper routines or data downloaded into a node could potentially damage the electronics or other vehicle components or risk the vehicle's compliance to emission, safety, or security standards. This mode is intended to be used to implement the data link security measures defined in SAE J2186 (E/E Data Link Security).

8.8.1 Service Description. The security concept uses a seed and key relationship. The seed and key are each 16-bit numbers (2-byte). The security seed and key shall be stored in non-volatile memory during the manufacturing process of the Node.

Note: The security seed shall be a random non-zero and non-$FFFF number. The security seed shall be generated for each individual part (all nodes with the same part number shall NOT have the same security key).

The security key shall be derived from the security seed using an external encryption algorithm**.** The SPS programming security algorithm for a node is assigned by GM Service and Parts Operations. The manufacturing device control security algorithm for a node is assigned by the Manufacturing Serial Data Design Engineer. Under no circumstances shall the encryption algorithm ever reside in the node.

A typical example of the use of this service is as follows:

Tester requests the Seed.

Node sends the Seed.

Tester sends the Key (appropriate for the Seed received).

Node responds that the Key was valid and that it will unlock itself

A 10 s time delay shall be required before the node can positively respond to a service SecurityAccess requestSeed message from the tester after node power-on. This delay is only required if the MEC > 0 or the vulnerability flag is = $FF when powered on. Reference the pseudo code in paragraph 8.8.6.2 for further clarification.

The tester shall request the node to unlock itself by sending the service SecurityAccess requestSeed message. The node shall respond by sending a seed using the service SecurityAccess "requestSeed" positive response message. The tester shall then respond by returning a key number back to the node using the service SecurityAccess sendKey request message. The node shall compare this key to one internally stored. If the two numbers match, then the node shall enable (unlock) the tester's access to specific services and indicate that with the service SecurityAccess sendKey positive response message. If upon two attempts of a service SecurityAccess "sendKey" request message by the tester, where the two keys do not match, then the node shall insert a 10 s time delay before allowing further attempts. The 10 s delay shall be invoked for each subsequent failed attempt. An invalid key requires the external tester to start over from the beginning with a Security Access request message.

If a device supports security, but is already unlocked (MEC > 0 or the vulnerability flag is = $FF) when a SecurityAccess requestSeed message is received, that node shall respond with a SecurityAccess requestSeed positive response message service with a seed value of $0000. A tester shall use this method to determine if a node is locked by checking for a non-zero seed.

Attempts to access security shall not prevent normal vehicle communications or other diagnostic communication.

Nodes which provide security shall support reject messages if a secure service is requested while the node is locked (e.g., TransferData $36 service). Reference the description and pseudo code sections of each diagnostic service to determine where security access applies.

This service requires a TesterPresent ($3E) to be sent prior to a $P3_C$ timeout or the node shall lock itself (unless it was already unlocked when this service was first requested).

Nodes are considered locked or unlocked based on the table below. See paragraph 9.3.2.6 for definitions of the Manufacturers Enable Counter and Vulnerability Flag. The Security Status does not change state when the MEC value is written to $00. The Security Status is evaluated when a SecurityAccess "requestSeed" message is received. See Table 99.

Table 99: Node Security Status

Manfacturers Enable Counter (MEC)	Vulnerability Flag	Security Status
$00	$FF	Unlocked (Unsecured) [Note 1]
$00	! = $FF	Locked (Secured)
> $00	$FF	Unlocked (Unsecured) [Note 1]
> $00	! = $FF	Unlocked (Unsecured) [Note 1]

Note 1: Requests for secured services, paramaters, or functions are rejected ($7F $XX $31) if no request for seed.

8.8.2 Request Message Definition. Table 100 indicates the structures of the valid request messages for this service.

Table 100: SecurityAccess Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Security Access Request Service Id	M	27	SIDRQ
#2	sub-function = [requestSeed sendKey]	M_1	 xx xx + 1	LEV_ RSD SK
 #3 #4	SecurityKey = [HighByte LowByte]	M_2	 yy zz	SECKEY_ KEYHB KEYLB

Where:

M_1 = The value represented by “xx” must be an odd number.

M_2 = Mandatory if request $Level is sendKey. Not present if request level is requestSeed

8.8.2.1 Request Message Sub-function Parameter $Level (LEV_) Definition. The sub-parameter used in the request message of the SecurityAccess service indicates to the node the step in progress for this service. The sub-parameters defined as valid for the request message of this service are indicated in Table 101.

The values of the parameter seed are not defined in this document except for the value $0000 which indicates to the tester that the node is not locked, and $FFFF which shall not be used because this value may occur if the Nodes memory has been erased.

The values of the parameter key are not defined in this document.

Table 101: Definition of Sub-function Values

Hex	Description	Cvt	Mnemonic
00	**ReservedByDocument** This value shall not be allowed.	M	RBD
01	**SPSrequestSeed** This is the requestSeed level used for ECU programming via the SPS system. The security encryption data shall be provided to the supplier by the GM Service Organization.	M_3	SPSRSD
02	**SPSsendKey** This is the sendKey level used for ECU programming via the SPS system. The security encryption data shall be provided to the supplier by the GM Service Organization.	M_3	SPSSK
03	**DevCtrlrequestSeed** This is the requestSeed level for enabling vehicle manufacturing specific device control restrictions (as opposed to having service device control restrictions enabled). The encryption data shall be provided to the supplier from either the GM Manufacturing Serial Data Design Engineer or the DRE. A request for this level shall be rejected if the service device control restrictions are active and the device control service is active at the time of the request.	C	DVCRSD
04	**DevCtrlsendKey** This is the sendKey level for enabling vehicle manufacturing specific device control restrictions (as opposed to having service device control restrictions enabled). The encryption data shall be provided to the supplier from either the GM Manufacturing Serial Data Design Engineer or the DRE.	C	DVCSK
05 thru 0A	**ReservedByDocument** This range shall be reserved for future standardized use as defined within this specification.	M	RBD
0B thru FA	**Reserved for vehicle manufacturer specific needs.** This range of values is reserved for future needs of the vehicle manufacturer. Any use of this range for specific ECUs shall be agreed upon by the Design Release Engineer (DRE), representative(s) from GM Service, and representative(s) from manufacturing. Levels used in this range and their purpose shall be documented in a CTS.	U	RFVM
FB thru FE	**Reserved for ECU or system supplier manufacturing needs** This range of values is reserved for ECU supplier needs. They can be used to support factory testing by the ECU supplier. Use of this feature shall be agreed upon by the supplier and the GM DRE. If this range is used by the supplier, the amount of ECU resources (ROM, RAM, Flash, etc) consumed to support factory testing shall be provided to the GM DRE.	U	RFSS
FF	**ReservedByDocument** This value shall not be allowed.	M	RBD

Where:

M_3 = This requestSeed and sendKey pair is required for all SPS programmable ECUs which provide emission, safety, or theft related functionality, or are otherwise required by law.

C = Conditional. This need for the use of this level for a given ECU is to be agreed upon by the DRE and Manufacturing Serial Data Design Engineer.

8.8.2.2 Request Message Data Parameter Definition (Table 102).

Table 102: Request Message Data Parameter Definition

Definition
SecurityKey (high and low bytes) The Key parameter in the request message is the value generated by the security algorithm corresponding to a specific seed value.

8.8.3 Positive Response Message Definition. Table 103 indicates the separate message structures for the positive responses relating to the requestSeed and sendKey request levels.

Table 103: SecurityAccess Positive Response

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Security Access Positive Response Service Id	M	67	SIDPR
#2	RespLevel = [requestSeed sendKey]	M	 xx xx + 1	RPLEV_ RSD SK
 #3 #4	SecuritySeed = [HighByte LowByte]	M_4	0000 thru FFFE	SS_ HB LB
Where: $\mathbf{M_4}$ = Mandatory if request level is requestSeed. Not present if request level is sendKey.				

8.8.3.1 Positive Response Message Data Parameter Definition (Table 104).

Table 104: Response Message Data Parameter Definition

Definition
RespLevel This byte is an echo of the sub-parameter from the request message.
SecuritySeed (high and low bytes) The seed parameter is a data value sent by the ECU and is used by the tester when calculating the key needed to access security. The SecuritySeed data bytes are only present in the response message if the request message was sent with the sub-parameter set to requestSeed.

8.8.4 Supported Negative Response Codes. The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 105.

Table 105: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** • Send if the sub-function parameter in the request message is not supported • Send if the length of the request message is incorrect for the requested sub-function.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** • Send if the “sendKey” sub-parameter is received without first receiving a “requestSeed” request message. • If the DeviceControl Restrictions of this service are supported, this code shall be returned if DeviceControl is already active and using the service device control restrictions at the time of the security request.	M	CNCRSE
35	**InvalidKey** Send if the value of key is not valid for the node.	M	IK
36	**ExceededNumberOfAttempts** Send if too many attempts with invalid values are requested.	M	ENOA
37	**RequiredTimeDelayNotExpired** Send if the delay timer is active and a request is transmitted.	M	RTDNE
78	**RequestCorrectlyReceived-ResponsePending** Send if it will take more than P2 ms to process the request.	C	RCR-RP

8.8.5 Message Flow Example SecurityAccess.

8.8.5.1 SecurityAccess (requestSeed). In the following example (Table 106), it is assumed:

- The node is programmable (supports mode $27).
- $aabb is a valid seed value for the target node.
- The node has a USDT physical request CANId of $241.
- The node has a USDT physical response CANId of $641.

Table 106: SecurityAccess(requestSeed) Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$27	$01	---	---	---	---	---
N(USDT-SF)	$641	$04	$67	$01	$aa	$bb	---	---	---

8.8.5.2 SecurityAccess (sendKey) Positive Response. In the following example (Table 107), it is assumed:

- The node is programmable (supports mode $27).
- The tester has already requested a seed and calculated the corresponding key.
- $ccdd is a valid key value for the target node.
- The node has a USDT physical request CANId of $241.
- The node has a USDT physical response CANId of $641.

Table 107: SecurityAccess(sendKey) Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$04	$27	$02	$cc	$dd	---	---	---
N(USDT-SF)	$641	$02	$67	$02	---	---	---	---	---

8.8.6 Node Interface Function.

8.8.6.1 Node Interface Data Dictionary (Table 108).

Table 108: Node Interface Data Dictionary of SecurityAccess Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **TesterPresent_Timer_State** **vulnerability_flag** **manufacturers_enable_counter** **Dev_Cntrl_Active** **DeviceContol_Security_Level**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
SPS_Bad_Key_Counter Counter of invalid Mode $27 requests. Following two invalid requests, the device will reject any further Mode $27 requests for the next 10 s.	Range: 0 to 2 counts Resolution: 1 count
DevCtrl_Bad_Key_Counter Counter of invalid Mode $27 requests. Following 2 invalid requests, the device will reject any further Mode $27 requests for the next 10 s.	Range: 0 to 2 counts Resolution: 1 count
Security_Delay_Timer Count down timer used to reject Mode $27 requests for 10 s following two level $02 requests with an incorrect key.	Range: 0 to 10 s Resolution: 0.1 s
SPS_Security_Key_Allowed Indication that a level $01 security request for the seed has been received by the device and a level $02 security key request is allowed.	TRUE/FALSE
DevCtrl_Security_Key_Allowed Indication that a level $03 security request for the seed has been received by the device and a level $04 security key request is allowed.	TRUE/FALSE
Security_Access_Allowed Indication that a valid security code (as defined in the Vehicle Theft Deterrent SSTS) has been received.	TRUE/FALSE

8.8.6.2 Node Interface Pseudo Code.

```
Powerup States:
SPS_Bad_Key_Counter ← 2
SPS_Security_Key_Allowed ← FALSE
DevCtrl_Bad_Key_Counter ← 2
DevCtrl_Security_Key_Allowed ← FALSE
DeviceContol_Security_Level ← SERVICE

IF ((Vulnerability_flag=$FF) OR (manufacturers_enable_counter > 0))THEN
    Security_Delay_Timer ← 0
ELSE
    Security_Delay_Timer ← 10
ENDIF
Each time a $27 message is received, the following logic is executed:
BEGINFUNCTION Serv_27_Msg_Recvd()
IF (more than P2C ms is needed to process this request) THEN
    send Negative Response ($7F $27 $78) /* RequestCorrectlyReceived-ResponsePending */
ENDIF
IF ( $Level != $00 AND $Level != $FF AND
(($Level MOD 2=$01 AND message_data_length = 2) OR
($Level MOD 2=$00 AND message_data_length = 4))) THEN
    SELECT FIRST
    WHEN ($Level=$01)
        IF (Security_Delay_Timer > 0) THEN
            Send ($7F $27 $37) /* Time Delay Not Expired */
        ELSE
            IF ((Vulnerability_flag=$FF) OR (manufacturers_enable_counter > 0)) THEN
                    Security_Access_Unlocked ← TRUE
            ENDIF
            IF (Security_Access_Unlocked=TRUE) THEN
                TesterPresent_Timer_State ← ACTIVE
                Send ($67 $01 $00 $00) message
            ELSE
                Send ($67 $01 $Seed) message
                SPS_Security_Key_Allowed ← TRUE
            ENDIF
        ENDIF
    WHEN ($Level=$02)
        IF (SPS_Security_Key_Allowed=TRUE) THEN
            SPS_Security_Key_Allowed ← FALSE
            IF ($Key does not match stored key) THEN
                SPS_Bad_Key_Counter ← SPS_Bad_Key_Counter - 1
                IF (SPS_Bad_Key_Counter = 0) THEN
                    Security_Delay_Timer ←10 s
                    SPS_Bad_Key_Counter ← 1
                    send ($7F $27 $36) /* for Exceeded Number Of Attempts */
                ELSE
                    send a ($7F $27 $35) message
                ENDIF
            ELSE
                Security_Access_Unlocked ← TRUE
                TesterPresent_Timer_State ←ACTIVE
                SPS_Bad_Key_Counter ← 2
                Send ($67 $02 ) message.
            ENDIF
        ELSE
            send Negative Response ($7F $27 $22) /*Sequence Error */
        ENDIF
    WHEN ($Level=$03)
        IF (Security_Delay_Timer > 0) THEN
            Send ($7F $27 $37) /* Time Delay Not Expired */
        ELSE IF ((Dev_Cntrl_Active = YES) AND
        (DeviceContol_Security_Level = SERVICE)) THEN
            send ($7F $27 $22) /* conditions not correct or sequence error*/
        ELSE
            IF ((DeviceContol_Security_Level = ASSEMBLY_PLT) OR
            (manufacturers_enable_counter > 0) OR
            (Vulnerability_flag=$FF)) THEN
                DeviceContol_Security_Level ← ASSEMBLY_PLT
                TesterPresent_Timer_State ←ACTIVE
                Send ($67 $03 $00 $00) message
            ELSE
                DevCtrl_Security_Key_Allowed ← TRUE
                Send ($67 $03 $Seed) message
            ENDIF
        ENDIF
    WHEN ($Level=$04)
        IF (DevCtrl_Security_Key_Allowed = FALSE) THEN
            send ($7F $27 $22) /* conditions not correct or sequence error*/
            DevCtrl_Security_Key_Allowed ← FALSE
        ELSE
            DevCtrl_Security_Key_Allowed ← FALSE
            IF ($Key does not match stored key) THEN
                DevCtrl_Bad_Key_Counter ← DevCtrl_Bad_Key_Counter - 1
                IF (DevCtrl_Bad_Key_Counter = 0) THEN
                    Security_Delay_Timer←10 s
                    DevCtrl_Bad_Key_Counter ← 1
                    send ($7F $27 $36) /* for Exceeded Number Of Attempts */
                ELSE
                    send a ($7F $27 $35) message
                ENDIF
            ELSE
```

```
                DeviceContol_Security_Level ← ASSEMBLY_PLT
                TesterPresent_Timer_State ←ACTIVE
                DevCtrl_Bad_Key_Counter ← 2
                Send ($67 $04) message.
            ENDIF
        ENDIF
    WHEN ((($Level > $0A)AND($Level < $FB)) AND ($Level MOD 2 = 1))
        /* vehicle manufacturer specific - request seed */
    WHEN ((($Level > $0A)AND($Level < $FB)) AND ($Level MOD 2 = 0))
        /* vehicle manufacturer specific - send key */
    WHEN ($Level ≥ $FB AND ($Level MOD 2 = 1))
        /* system supplier specific - requestSeed*/
    WHEN ($Level ≥ $FB AND ($Level MOD 2 = 0))
        /* system supplier specific */
    ENDSELECT
ELSE
    send Negative Response ($7F $27 $12) /*Subfunction Not Supported-Invalid Format*/
ENDIF
ENDFUNCTION
```

The following logic is executed periodically (suggested execution rate = 100 ms.):

```
IF (Security_Delay_Timer > 0) THEN
    Decrement the Security_Delay_Timer by the execution rate.
ENDIF
```

8.8.7 Node Verification Procedures.

8.8.7.1 General Procedures.

Procedure 1:

1. Send a security access request message with the sub-function parameter ($Level) equal to a value that is not supported and verify $7F $27 $12 response.
2. If negative response code $78 is supported by the ECU, then create the conditions under which the ECU should return the $7F $27 $78 response and verify that proper response is sent.
3. Repeat step 2 of this procedure for each possible reason an ECU would send the negative response with response code $78. Verify this for each applicable supported sub-function parameter.

8.8.7.2 Verification Procedures for SPS Programming Security Levels ($01, $02). The steps for Procedure 1 below shall be performed within 10 s of the ECU being powered up (executing operational software). For Procedures 2 through 5 below, the verification procedures shall be performed after the ECU has been powered up (executing operational software) for at least 10 s.

Procedure 1:

1. Within 10 s of power-up send a valid security access request message $01 to an ECU when its vulnerability_flag !=$FF and its MEC = $00. Verify that the response is $7F $27 $37.
2. Repeat step 1 of this procedure periodically (approximately every 500 ms) and verify that the $7F $27 $37 response is sent each time until the ECU has been powered for 10 s.
3. Send a valid security access request message $01 within 10 s of power-up to an ECU when the vulnerability_flag = $FF (if supported) and the MEC = $00. Verify the proper positive response message $67 $01 $00 $00.
4. Send a valid security access request message $01 within 10 s of power-up to an ECU when the MEC > $00 and the vulnerability_flag (if supported) is not set to $FF. Verify the proper positive response message $67 $01 $00 $00.

Procedure 2: This procedure is to be performed twice, once while the ECU is secured (manufacturers enable counter = $00, vulnerability flag < $FF, and security has not been accessed (SecurityAccess ($27) request has not been sent)) and once when the ECU is unsecured ((manufacturers enable counter > $00 or vulnerability flag = $FF), and security has been accessed (SecurityAccess ($27) request has been sent)).

1. Send a security access request message $01 with an incorrect number of data bytes and verify $7F $27 $12 response.
2. Send a security access request message $02 without sending a security access request message $01 and verify $7F $27 $22.
3. Send a valid security access request message $01 and verify the security access positive response message $01. Then send a request message $02 with an incorrect number of data bytes and verify the $7F $27 $12 response (only perform this step when the seed response to the level $01 request is a non zero seed).

Procedure 3: This procedure is to be performed while the ECU is unsecured ((manufacturers enable counter > $00 or vulnerability flag = $FF), and security has been accessed (SecurityAccess ($27) request has been sent)).

1. Send a valid security access request message $01 with the device already unsecured and verify $67 $01 $00 $00 response.
2. Repeat step 1 of this procedure and then send a security access request message with level $02. Verify $7F $27 $22 response.
3. After performing steps 1 and 2 of this procedure, verify that the ECU remains unsecured by attempting a secure service and verifying the correct positive response.

Procedure 4: The steps in this procedure are to be performed sequentially. This procedure is to be performed twice, once while the ECU is secured (manufacturers enable counter = $00, vulnerability flag < $FF, and security has not been accessed (SecurityAccess ($27) request has not been sent)) and once while the ECU is unsecured ((manufacturers enable counter > $00 or vulnerability flag = $FF), and security has been accessed (SecurityAccess ($27) request has been sent)).

1. Send a valid security access request message with $Level = $01. Verify the proper positive response ($0000 seed if unlocked and non zero seed if locked).
2. This step is performed when the seed in the above step is a non zero value. Send a valid security access request message $02 with a correct key and verify the proper positive response message.
3. Verify that the device is now unsecured by attempting a secure mode and verifying the appropriate positive response.
4. Send valid $3E message within $P3_C$ ms for 2 minutes. Then verify the device remains unsecured by attempting a secure service and verifying the appropriate positive response.
5. Stop sending TesterPresent ($3E) messages. Verify that after $P3_{Cmax}$ ms without sending a $3E message the device is secured by requesting a secure service and verifying the appropriate negative response.

Procedure 5: This procedure is to be performed when the ECU vulnerability flag is not $FF (if supported) and the MEC = 0.

1. Send a valid security access request message $01 and verify $67 $01 $seedMsb $seedLsb response.
2. After step 1 is complete, send a security access request message with level $02 and an incorrect key. Verify $7F $27 $35 response.
3. Repeat step 1 and 2 of this procedure and verify that the response from step 2 (the second level $02 message) changes to $7F $27 $36. Then send a request for this service with $Level = $01 within 10 s and verify $7F $27 $37 message.
4. After performing steps 1 through 3 of this procedure, verify that the ECU remains locked by attempting a secure service and verifying the correct negative response.

8.8.7.3 Verification Procedures for Device Control Security Levels ($03, $04). The procedures below shall be followed if the ECU supports the Device Control security access levels ($03 and $04). The steps for Procedure 1 below shall be performed within 10 s of the ECU being powered up (executing operational software). For procedures 2 through 6 below, the verification procedures shall be performed after the ECU has been powered up (executing operational software) for at least 10 s.

Procedure 1:

1. Send a valid security access request message $03 within 10 s of power-up to an ECU when the vulnerability_flag !=$FF (if supported) and the MEC = $00. Verify $7F $27 $37 response.
2. Repeat step 1 of this procedure periodically (approximately every 500 ms) and verify that the $7F $27 $37 response is sent each time until the ECU has been powered for 10 s.
3. Send a valid security access request message $03 within 10 s of power-up to an ECU when the MEC > $00 and vulnerability flag not equal to $FF (if supported). Verify the proper positive response message $67 $03 $00 $00.
4. Send a valid security access request message $03 within 10 s of power-up to an ECU when the MEC equals $00 and vulnerability flag is set to $FF (if supported). Verify the proper positive response message $67 $03 $00 $00.

Procedure 2: This procedure is to be performed twice, once while the ECU has the service device control restrictions active and once while the ECU has the assembly plant device control restrictions active.

1. Send a security access request message $03 with an incorrect number of data bytes and verify $7F $27 $12 response.
2. Send a security access request message $04 without sending a security access request message $03 and verify $7F $27 $22.
3. Send a valid security access request message $03 and verify the security access positive response message $03. Then send a request message $04 with an incorrect number of data bytes and verify the $7F $27 $12 response.

Procedure 3: This procedure is to be performed while the ECU has the assembly plant device control restrictions active.

1. Send a valid security access request message $03 while the device has assembly plant restrictions active and verify $67 $03 $00 $00 response.
2. Repeat step 1 of this procedure and then send a security access request message with level $04. Verify $7F $27 $22 response.
3. After performing steps 1 and 2 of this procedure, verify that the ECU keeps the assembly plant device control restrictions active. This can be done by sending a device control command that is valid for the assembly plant restrictions but invalid for service restrictions and verifying a positive response message.

Procedure 4: The steps in this procedure are to be performed sequentially. This procedure is to be started when the ECU has the service device control restrictions active, the MEC = $00, and the vulnerability_flag (if supported) not equal to $FF.

1. Send a valid security access request message with $Level = $03. Verify the proper positive response.
2. Send a valid security access request message $04 with a correct key and verify the proper positive response message.
3. Verify that the device is now enforcing assembly plant device control restrictions by sending a device control command that is valid for the assembly plant restrictions but invalid for service restrictions and verifying a positive response message.
4. Send valid $3E message within $P3_C$ ms for 2 minutes (provided that the device control chosen can be active for 2 minutes - if not, adjust as necessary) while maintaining the conditions which are valid for the assembly plant restrictions and invalid for service restrictions. Verify that the ECU keeps the device control active and does not send any device control limits exceeded $7F $AE $E3 $xx $yy reject messages.
5. Stop sending TesterPresent ($3E) messages. Verify that after $P3_{Cmax}$ ms without sending a $3E message that the device reverts back to the service device control restrictions. This can be done by sending a device control command that is valid for the assembly plant restrictions but invalid for service restrictions and verifying the correct $7F $AE $E3 $xx $yy negative response message.

Procedure 5:

1. Send a valid device control request (mode $AE) and verify the proper positive response. Then send a valid security access request message $03 and verify the negative response $7F $27 $22 (the negative response is sent because device control is already active using service restrictions when the request is received).

Procedure 6: This procedure is to be performed when the MEC = $00 and the vulnerability_flag (if supported) not equal to $FF.

1. Send a valid security access request message $03 and verify $67 $03 $seedMsb $seedLsb response.
2. After step 1 is complete, send a security access request message with level $04 and an incorrect key. Verify $7F $27 $35 response.
3. Repeat step 1 and 2 of this procedure and verify that the response from step 2 (the second level $04 message) changes to $7F $27 $36. Then send a request for this service with $Level = $03 within 10 s and verify $7F $27 $37 message.
4. After performing steps 1 through 3 of this procedure, verify that the service device control restrictions remain active. This can be done by sending a device control command that is valid for the assembly plant restrictions but invalid for service restrictions and verifying the correct $7F $AE $E3 $xx $yy negative response message.

8.8.8 Tester Implications. This test mode will be terminated if a $3E message is not received every $P3_C$ ms. Upon a Mode $3E time-out, the node shall lock itself (also revert back to service device control restrictions if applicable).

An invalid key requires the external tester to start over from the beginning with a mode $27 (level 1) request message. This is a requirement of SAE J2186. The tester cannot send a key to a secured device without first requesting the seed.

8.9 DisableNormalCommunication ($28) Service. The purpose of this service is to prevent a device from transmitting or receiving all messages which are not the direct result of a diagnostic request. The primary use of the service is to set up a programming event. This is a required service that must be supported by all nodes.

8.9.1 Service Description. When this service is activated, a node shall cease transmission of all normal mode (non-diagnostic) messages. Normal messages are halted by invoking a handler function(s) which performs the following tasks:

1. Inhibits the transmission of all non-diagnostic signals/messages. Non-diagnostic frames already loaded into the protocol device do not have to be flushed, but no additional non-diagnostic frames shall be loaded.
2. De-queue all queued messages and inhibit further queuing of non-diagnostic messages.
3. Disable the processing of all received messages except for those corresponding to the specific diagnostic CAN Identifiers supported by the ECU.

A node shall not send out any normal communication frames after sending the positive response to this service, until after this service is no longer active. Disabling the processing of normal mode messages is necessary to facilitate module programming for service and assembly. During the programming process, certain normal communication CAN Identifiers may be temporarily used as diagnostic CAN Identifiers. See the chapter in this specification on programming for further details on the use of these CAN Identifiers for programming.

This service also disables the setting of all diagnostic trouble codes. A device receiving and complying with this request shall take failsoft action on all necessary parameters.

Note: A device which is connected to more than one GMLAN link shall be able to receive, process and respond to a service $28 request on each link to which it is connected.

Note: Use of this service during a programming event should always be targeted to all nodes using the AllNodes functional diagnostic request CANId ($101) and the AllNodes extended address ($FE).

This service requires a periodic TesterPresent ($3E) service message to be transmitted by the tester to remain active.

This service shall be terminated when:

A node receives a $20 request.

A P3$_C$ (TesterPresent) time-out occurs.

8.9.2 Request Message Definition (Table 109).

Table 109: DisableNormalCommunication Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DisableNormalCommunication Request Service Id	M	28	SIDRQ

8.9.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. Because this service is defined as a single operation, there are no sub-function parameters required or valid for the request message.

8.9.2.2 Request Message Data Parameter Definition. There are no data parameters used with this service.

8.9.3 Positive Response Message Definition (Table 110).

Table 110: Disable Normal Communication Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DisableNormalCommunication Positive Response Service Id	M	68	SIDPR

8.9.3.1 Response Message Data Parameter Definition. There are no data parameters used by this service in the positive response message.

8.9.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 111) shall be implemented for this service. The circumstances under which each response code would occur are documented in the table below.

Table 111: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** Used when the request message contains more than the SID byte.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** Used when the node is in a critical normal mode activity. If this return code is implemented, the reasons shall be documented in the CTS.	C	CNCRSE

8.9.5 Message Flow Example DisableNormalCommunication.

8.9.5.1 DisableNormalCommunication Positive Response Example. In the following example (Table 112), it is assumed:

- There is more than one node on the link.
- All nodes can comply with the request.
- Node 1 has a USDT physical response CANId of $641.
- Node n has a USDT physical response CANId of $64F.

Table 112: DisableNormalCommunication - Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$28	---	---	---	---	---
N1(USDT-SF)	$641	$01	$68	---	---	---	---	---	---
				:					
Nn(USDT-SF)	$64F	$01	$68	---	---	---	---	---	---

8.9.6 Node Interface Function.

8.9.6.1 Node Interface Data Dictionary (Table 113).

Table 113: Node Interface Data Dictionary of SecurityAccess Service Pseudo Code

Variable/Meaning	Values
message_data_length **TesterPresent_Timer_State** **normal_message_transmission_status** **Diag_Services_Disable_DTCs** **diagnostic_responses_enabled**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.9.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $28 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_28_Msg_Recvd()
```

```
IF (message_data_length != 1) THEN
    IF (diagnostic_responses_enabled = YES) THEN
        Reject the request with a ($7F $28 $12) for invalid format
    ENDIF
ELSE
    IF (Device cannot disable normal communications at this time) THEN
        /* do not need to check if diag resp enabled here because if they are not, then there would be no
        reason why the device would not already have disabled normal comm */
        Reject the request with a ($7F $28 $22) for conditions not correct
    ELSE
        Diag_Services_Disable_DTCs ← TRUE
        normal_message_transmission_status ← DISABLED
        CALL hnd_halt_normal_comm() /* handler call to halt normal comm */
        TesterPresent_Timer_State ← ACTIVE
        IF (diagnostic_responses_enabled = YES) THEN
            Send ($68) response message
        ENDIF
    ENDIF
ENDIF
ENDFUNCTION
```

8.9.7 Node Verification Procedure. Perform procedures 1 and 2 below on devices that are completely programmed (SPS_TYPE_A ECU).

Procedure 1:

1. Send a DisableNormalCommunication request with data bytes after the Service Id and verify the negative response ($7F $28 $12) message is sent.
2. Send a DisableNormalCommunication request message at a time when the device cannot disable Normal Mode communication and verify the negative response ($7F $28 $22) message is sent. (This is only required for devices which support such a situation.)

Procedure 2:

1. Verify that the device has no **Current** or **History** DTCs stored (use $A9 service).
2. Send a DisableNormalCommunication request message. Keep mode active with Tester present messages (mode $3E) for at least 2 minutes. Verify that all Normal Mode communications are disabled.
3. Stop sending TesterPresent messages and wait $P3_{Cmax}$ ms. Verify that normal communications are reinitialized.
4. Verify that the device has no **Current** or **History** DTCs stored (use $A9 service).

Procedure 3: For SPS_TYPE_C ECUs that do not have diagnostic responses enabled.

1. Send a DisableNormalCommunication functional request with data bytes after the Service Id and verify that there is no response from the SPS_TYPE_C ECU.
2. Next, send a $A2 service request and verify that no response is sent (because a valid $28 service had not been received).

Procedure 4: For SPS_TYPE_B ECUs

1. Send a valid DisableNormalCommunication request and verify the correct positive response from the SPS_TYPE_B ECU.

8.9.8 Tester Implications. Upon receipt of the response to this request, the test device may assume that the link will now be quiet for at least $P3_C$ ms. This mode will be terminated if a TesterPresent ($3E) message is not received at least once every $P3_C$ ms.

The tester is also responsible to send a ReturnToNormalMode ($20) request (to all nodes) if any single node rejects the request for service $28. Otherwise, the device that rejected the $28 service would still be expecting normal communications traffic while the nodes that accepted the $28 request would quit communicating, resulting in erroneous serial data DTCs being set.

8.10 DynamicallyDefineMessage ($2C) Service. This service is used to dynamically define the contents of diagnostic data packets which are formatted as UUDT messages and can be requested via the ReadDataByPacketIdentifier ($AA) service. The use of dynamic data packets allows a test device to optimize its diagnostic routines and bus bandwidth utilization by packing messages to only contain the diagnostic signal/parameter information that is required for the current test.

8.10.1 Service Description. Diagnostic data packets contain a one byte Data Packet Identifier (DPID #) and one to seven bytes of signal/parameter data. UUDT diagnostic messages contain a message number and up to seven bytes of data. The DPID# for a diagnostic data packet shall be the same as the message number used when the diagnostic UUDT message is transmitted (i.e., DPID# $FE will be identified as diagnostic UUDT message # $FE). Dynamically definable data packet Identifiers shall start with $FE and be numbered sequentially backwards. No DPIDs are allowed to occupy the range of UUDT message numbers from $80 through $8F as these UUDT message numbers are reserved for the Diagnostic Trouble Code (DTC) services. Dynamic data packets will be referenced in subsequent sections of this service as a DPID.

Each diagnostic signal/parameter is assigned a 2-byte Parameter Identifier (PID) number which is used to build the dynamic DPIDs. The term PID will be used to reference diagnostic signals/parameters in subsequent sections of this service.

All of the PIDs packed into a dynamic DPID shall be defined with a single request of this service. This request may be a single-frame or multiple-frame message depending on the number of PIDs being packed in the DPID. Once the test device receives a positive response from a $2C request, the PIDs in each subsequent transmission of the corresponding UUDT message (as requested via service $AA) shall match those of the last $2C service request message. A dynamic data packet shall be capable of being initially defined or modified any time a node is capable of communications and shall only require a $2C service request to do so (i.e., no key cycles or any other special events can be required). It shall also be possible to redefine a DPID while it is currently being scheduled by the periodic scheduler. See Service $AA description for more detail on the ability to schedule periodic UUDT messages. If the tester redefines a DPID without stopping the scheduler, the PIDs in the response message may not be correct until the 2nd response of that message. This is the case because a message may be queued for transmission at the time the DPID is being redefined and may not gain access to the link until after successful completion of the $2C service. If the tester wants to guarantee that the next transmission of the corresponding UUDT message contains the correct information, then the tester should remove that DPID from the scheduler prior to redefining it.

If a $3E (TesterPresent) time-out occurs, or upon receipt of a $20 service (ReturnToNormalMode) request message, the device shall retain the contents of all dynamic DPIDs. In addition, the periodic message scheduler may be stopped and started without affecting the contents of any dynamic DPIDs.

A device which implements this service shall be capable of buffering a 3-frame request message. This is the maximum number of frames needed to define a single dynamic DPID.

8.10.2 Request Message Definition (Table 114).

Table 114: DynamicallyDefineMessage Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DynamicallyDefineMessage Request Service Id	M	2C	SIDRQ
#2	DPIDIdentifier = [DPID#]	M	FE thru 90 or 7F thru 01	MN_ DPIDID
 #3 #4 : #n-1 #n	PIDData = [PIDHighByte_1 PIDLowByte_1 : PIDHighByte_m PIDLowByte_m]	 M M : U U	 xx xx : xx xx	PIDDTA_ PIDH1 PIDL1 : PIDHn PIDLn

8.10.2.1 Request Message Sub-function Parameter $Level Definition. Because this service is defined as a single operation, there are no sub-parameters required or valid for the request message.

8.10.2.2 Request Message Data Parameter Definition. Table 115 specifies the data parameter definitions for this service.

Table 115: Request Data Parameter Definition

Definition
DPIDIdentifier The value of the Data Packet to be defined. This is also the value used in the mode $AA service request message to read the dynamic data values of the PIDs assigned to be part of a specific Data Packet.
PIDData[] These are the PID numbers to be assigned as part of the specific Data Packet.

8.10.3 Positive Response Message Definition (Table 116).

Table 116: DynamicallyDefineMessage Positive Response Messages

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DynamicallyDefineMessage Positive Response Service Id	M	6C	SIDPR
#2	DPIDIdentifier = [DPID#]	M	FE thru 90 or 7F thru 01	PRMN_ DPIDID

8.10.3.1 Positive Response Message Data Parameter Definition (Table 117).

Table 117: DynamicallyDefineMessage Data Parameter Definition

Definition
DPIDIdentifier The value of the Data Packet contained within the request message. This is an echo of the request DPIDIdentifier to confirm completion of the DPID assignment.

8.10.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 118) shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 118.

Table 118: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** The length of the request message is such that the request does not contain a DPID number and at least one PID. The length of the request message does not contain an even number of bytes (each PID is a 2-byte value so length must be an even number). If the total number of data bytes which would be reported in the UUDT message exceeds seven data bytes.	M	SFNS-IF
31	**RequestOutOfRange** A PID in the request message is not supported by the Node. The DPID# in the request message is either invalid or cannot be dynamically defined.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending.** See 7.2 Return Code Definition.	C	RCR-RP

8.10.5 Message Flow Example(s) DynamicallyDefineMessage. The examples below (Tables 119 thru 122) show how a dynamic DPID is defined for a Powertrain Control Module (PCM). All of the examples assume the following information to be true for the PCM:

- The diagnostic CAN Identifiers for the PCM are: $241-USDT Request, $541-UUDT Response, and $641-USDT Response.
- $FE thru $F7 is the valid range for dynamically definable DPIDs.
- Engine rpm is a 2-byte parameter and is identified by PID# $000C.
- Vehicle Speed is a 1-byte parameter and is identified by PID# $000D.
- Manifold Pressure is a 1-byte parameter and is identified by PID# $000B.
- Engine Coolant Temperature is a 1-byte parameter and is identified by PID# $0005.
- Engine Mass Airflow is a 2-byte parameter and is identified by PID# $0004.
- Spark Advance is a 1-byte parameter and is identified by PID# $000E.
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.

8.10.5.1 Example 1: Define DPID with Multiple PIDs. In the first example (Table 119), the tool sends a $2C request message to define DPID $FE with Engine rpm, Vehicle Speed, Manifold Pressure, Engine Coolant Temperature, and Mass Airflow data.

Table 119: DynamicallyDefineMessage Example 1 Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	CAN Id	#1	#2	#3	#4	#5	#6	#7	#8
T(USDT-FF)	$241	$10	$0C	$2C	$FE	$00	$0C	$00	$0D
N(USDT-FC)	$641	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$241	$21	$00	$0B	$00	$05	$00	$04	---
N(USDT-SF)	$641	$02	$6C	$FE	---	---	---	---	---

The UUDT response to a $AA service request for message $FE, would look like Table 120:

Table 120: UUDT Response Message Format for Example 1

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
N(UUDT)	$541	$FE	$xx	$xx	$yy	$zz	$aa	$bb	$bb

8.10.5.2 Example 2 Define DPID with Single PID. In this example (Table 121), the tool sends a $2C request message to define DPID $F7 with Spark Advance.

Table 121: DynamicallyDefineMessage Example 2 Message Flow

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$04	$2C	$F7	$00	$0E	---	---	---
N(USDT-SF)	$641	$02	$6C	$F7	---	---	---	---	---

The UUDT response to a $AA service request for message $F7, would look like Table 122:

Table 122: UUDT Response Message Format for Example 2

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
N(UUDT)	$541	$F7	$xx	---	---	---	---	---	---

8.10.6 Node Interface Function.

8.10.6.1 Node Interface Data Dictionary (Table 123).

Table 123: Node Data Dictionary of DynamicallyDefineMessage Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **Security_Access_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
valid_request This is a local flag which is used to keep track of whether or not the request message is OK to process.	YES/NO
invalid_PID This is a flag used to keep track of whether or not all of the PIDs in the request message are supported by the device.	YES/NO
total_length This is a counter used to keep track of the number of data bytes contained within a DPID. A DPID can contain a maximum of seven data bytes. As each PID in the request message is processed, the number of bytes needed to transmit that PID in the frame is added to this counter.	$01 thru $07

Variable/Meaning	Values
max_PID_index This is a constant that is used when indexing through the array of supported PIDs to determine if a PID is supported. Its value is equal to 1 less than the number of PIDs supported by the ECU.	constant determined at compile time
PID_Req_Index This is a local variable used to index through the array of supported PIDs to determine if a PID is supported.	0 to max_PID_index
Requested_PID This is a local variable which is set equal to a specific PID number from the request message when indexing through the list of requested PIDS to determine if all requested PIDs are supported.	$0000 to $FFFF
PID_Found This is a flag used to determine if a requested PID has been found in the list of supported PIDs.	YES/NO
PIDs_Supported[] This is an array of data structures that contains the list of supported PIDs and the length of the data for each supported PID. PIDs_Supported[].PID_Number is used in the pseudo code when checking the PID number to see if it is supported. PIDs_Supported[].length is the number of data bytes associated with the given PID.	N/A
Supported_Dyn_DPIDs[] This is an array that contains all of the supported dynamically definable DPID numbers.	N/A
max_Dynamic_DPID_Index This is a constant that is used when indexing through the array of supported dynamic DPIDs to determine if a DPID is supported. Its value is equal to 1 less than the number of DPIDs supported by the ECU.	constant determined at compile time
DPID_Index This is a local index variable used when indexing through the array of supported dynamic DPIDs to determine if the DPID from the request message is valid.	0 to max_Dynamic_DPID_Index
PidStructArray[] This is an array of structures that contain the data needed to support PIDs. for the purposes of the pseudo code of this service it is assumed that each structure contains the following information: 1. PidSecure - Set to YES if security access is required to retrieve PID data 2. PidSecurityCode – Set to YES if security code access is required to retrieve PID data Other elements may exist in the structure such as a pointer to a function to retrieve the data, memory address and length information (for a dynamic PID, see $2D service), etc. Only the PidSecure and PidSecurityCode are shown here for the purposes of the pseudo code. The pseudo code uses the following convention to access an element of the structure: PidStructArray[0].PidSecure would indicate if security access is required for the first PID structure in the array.	N/A

8.10.6.2 Node Interface Pseudo Code.

Powerup States:

All dynamic DPIDs are initialized to a data length of $00

Each time a $2C message is received, the following logic is executed:

```
BEGINFUNCTION Serv_2C_Msg_Recvd()
valid_request ← NO
IF ((message_data_length < 4) OR   /* message must have at least 4 data bytes */
((message_data_length MOD 2) != 0)) THEN /* length must be even number */
    send Negative Response ($7F $2C $12) /* InvalidFormat */
ELSE
    FOR (DPID_Index ← 0 TO max_Dynamic_DPID_Index BY 1)
        IF ($DPIDIdentifier = Supported_Dyn_DPIDs[DPID_Index]) THEN
            valid_request ← YES
            DPID_Index ← max_Dynamic_DPID_Index
        ENDIF
    ENDFOR
    IF (valid_request = NO) THEN
        send Negative Response ($7F $2C $31) /* RequestOutOfRange */
    ENDIF
ENDIF
IF (valid_request = YES)
    IF (Processing this message will take more than $P2_C$ ms) THEN
        Send ($7F $2C $78) /*for RequestCorrectlyReceived-ResponsePending */
    ENDIF
    invalid_PID ← NO
    total_length ← 0
    FOR (PID_Req_Index ← 0 TO (message_data_length - 4) BY 2)
        Requested_PID ← ((PIDData[PID_Req_Index] << 8) |
        (PIDData[PID_Req_Index + 1]))
        PID_Found ← NO
        FOR (PID_Index ← 0 TO max_PID_index BY 1)
            IF (Requested_PID = PIDs_Supported[PID_Index].PID_Number) AND
            ((PidStructArray[Index].PidSecure = NO) OR Security_Access_Unlocked = TRUE)) AND
            ((PidStructArray[Index].PidSecurityCode = NO) OR (Security_Access_Allowed = TRUE))
                PID_Found ← YES
                total_length ← (total_length + PIDs_Supported[PID_Index].length)
                PID_Index ← max_PID_index
            ENDIF
        ENDFOR
        IF (PID_Found = NO)
            invalid_PID ← YES
            PID_Req_Index ← (message_data_length - 2)
        ENDIF
    ENDFOR
    IF (invalid_PID = YES)
```

```
        send Negative Response ($7F $2C $31) /* RequestOutOfRange */
        valid_request ← NO
    ELSE IF (total_length > 7)
        send Negative Response ($7F $2C $12) /* InvalidFormat */
        valid_request ← NO
    ENDIF
ENDIF
/* process the request if it is valid */
IF (valid_request = YES)
    Assign PIDs to requested DPID for further use with $AA service
    store length of DPID for further use with $AA service
    Send ($6C $DPID#) response message
ENDIF
ENDFUNCTION
```

8.10.7 Node Verification Procedure.

Procedure 1:

1. Define a valid dynamic DPID with a single PID, verify that the correct parameter and length are reported with an $AA request.
2. Define a valid dynamic DPID using PIDs that will result in seven data bytes in the report message. Verify that the correct parameters are reported with a $AA request.
3. After step 2, send a $20 request and verify the positive response. Then send a request for the DPID defined in step 2 using the $AA service. Verify that the contents of the dynamic DPID were not lost by verifying the data in the $AA UUDT response message.

Procedure 2:

1. Define a valid dynamic DPID using PIDs that will result in more than seven data bytes in the report message. Verify the $7F $2C $12 response.
2. Send a request which attempts to define a dynamic DPID which the node under test does not support and verify the $7F $2C $31 response.
3. Send a request which attempts to define a dynamic DPID with a PID that the node under test does not support and verify the $7F $2C $31 response.
4. Send a request message to define a valid dynamic DPID with and include only one byte of data after the DPID number. Verify the $7F $2C $12 response.
5. Send a request message to define a valid dynamic DPID with and include an odd number of bytes (more than two, with the first two being a valid PID#) after the DPID#. Verify the $7F $2C $12 response.
6. Send a request message with no data bytes after the service identifier and verify the $7F $2C $12 response.
7. If applicable, send a request which attempts to define a dynamic DPID with a secure PID. Ensure that this is done at a time when security access has not been granted. Verify the $7F $2C $31 response.
8. If applicable, send a request which attempts to define a dynamic DPID with a secure PID. Ensure that this is done at a time when security access has been granted. Verify that the correct parameter and length are reported with an $AA request.
9. If applicable, send a request which attempts to define a dynamic DPID with a security code (as defined in the Vehicle Theft Deterrent SSTS) required PID. Ensure that this is done at a time when the security code has not been entered. Verify the $7F $2C $31 response.
10. If applicable, send a request which attempts to define a dynamic DPID with a security code required PID. Ensure that this is done at a time when the security code has been entered. Verify that the correct parameter and length are reported with an $AA request.

Procedure 3:

1. If the ECU supports the use of the negative response code $78, then send a request under conditions which would cause the $7F $2C $78 to be sent. Verify that the negative response is sent and that the appropriate last response is sent within the $P2_C$* timing criteria.

8.10.8 Tester Implications.

1. The test device may assume the contents of dynamic DPIDs are retained upon a $3E time-out or after a $20 service request.
2. If the tester redefines a DPID without stopping the scheduler, the PIDs in the response message may not be correct until the 2nd response of that message. This is the case because a message may be queued for transmission at the time the DPID is being redefined and may not gain access to the link until after successful completion of the $2C service.
3. This service should only be requested with physical addressing.

8.11 DefinePIDByAddress ($2D) Service. The purpose of the $2D DefinePIDbyAddress service is to provide the ability to map ECU variables to a dynamic Parameter Identifier (PID) using ECU memory addresses. The resulting PID defined by this service can then be requested via diagnostic service $22 or diagnostic service $AA.

Note: To retrieve a PID via the ReadDataByPacketIdentifier service the PID must first be packed into a dynamic DPID using the $2C service.

This service ($2D) supports defining a single PID per request (multiple PIDs can be defined in a given ECU, using multiple requests).

This service is intended for use during a device's development cycle to allow access to data that may not be available via another diagnostic service. This service is only intended for use in engineering development. It will not be used in field service applications.

8.11.1 Service Description. The DefinePIDbyAddress service allows a tester to dynamically assign an ECU variable to a PID via the address of the variable. The resulting PID data can then be read using either the $22 service or the $AA service. The service is intended to be used to retrieve RAM variables. Calibrations and unprotected areas of EEPROM/Flash memory should not be packed into a dynamic PID. This type of data is static in nature and can be obtained using service $23.

The dynamic PID being defined is identified by parameterIdentifier (PID), memoryAddress (MA) and memorySize (MS). The size of parameterIdentifier is fixed at two bytes; the size of the memoryAddress parameter in the request message can be two, three, or four bytes in length depending on the ECU internal addressing scheme. The number of bytes that a tester shall use in the memoryAddress portion of the request message for a given ECU shall be documented in the appropriate CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents. The memorySize parameter in the request message can range from one to seven bytes. (Seven bytes is the maximum in order to allow the PID to be part of a DPID.)

The DefinePIDbyAddress positive response message includes the parameterIdentifier (PID), specified in request message.

A device may deny access to specific addresses in order to protect their operating system and other secure data. An ECU may also restrict the testers ability to read specific addresses based on the security status of the ECU. If any of the addresses (that fall in the range of the request message) have security restrictions then the request shall be rejected unless the tester had previously successfully accessed security.

Note: Security access is only required if the ECU is locked based on the status of the MEC and/or the vulnerability flag see the SecurityAccess ($27) service for more information.

The range of PIDs that can be dynamically defined by this service in a given ECU shall be documented in the appropriate CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents.

Note: If a tool requests a dynamic PID before this service is called to define the PID then the resulting response is undefined.

8.11.2 Request Message Definition (Table 124).

Table 124: DefinePIDbyAddress Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1 [Note 1]	DefinePIDbyAddress Request Service Id	M	2D	SIDRQ
 #2 #3	parameterIdentifier = [PIDHighByte_1 PIDLowByte_1]	 M M	xx	PID_ B1 B2
 #4 #5 #6 #7	memoryAddress = [memoryAddress (Byte1 - MSB), memoryAddress (Byte2), memoryAddress (Byte3), memoryAddress (Byte4)]	 M M C1 [Note 2] C2 [Note 3]	xx	MA_ B1 B2 B3 B4
#6/7/8 [Note 4]	memorySize = [memorySize]	M	xx	MS_

Note 1: Byte1 in the memoryAddress parameter is always the most significant byte of the address and the last byte of the memoryAddress parameter (Byte2 for 2 byte addressing, Byte3 for 3-byte addressing, or Byte4 for 4-byte addressing) is always the least significant byte of the memoryAddress parameter.
Note 2: C1: Byte 6 is present when the memoryAddress parameter contains three or four bytes.
Note 3: C2: Byte 7 is present when the memoryAddress parameter contains four bytes.
Note 4: MemorySize byte is actual number of data bytes returned by the memory address specified. This value is limited to seven bytes maximum, so the memorySize is only a single byte value (hex) which should never exceed a value of $07.

8.11.2.1 Request Message Sub-Function $Level Parameter Definition. Because this service is defined as a single operation, there are no sub-function levels required or valid for the request message.

8.11.2.2 Request Message Data Parameter Definitions (Table 125).

Table 125: Request Message Data Parameter Definition

Definition
parameterIdentifier The parameterIdentifier is a 2-byte number assigned to individual diagnostic signals or parameters. The parameterIdentifier (PID) number is used by the $2C service to build dynamic DPIDs or the $22 service to request specific diagnostic parameters
memoryAddress The parameter memoryAddress is used to select the starting address of ECU memory from which data is to be retrieved. A portion of the memoryAddress parameter (most significant bits or bytes) can be used as a memoryIdentifier. An example of the use of a memoryIdentifier would be a dual processor ECU with 16-bit addressing and memory address overlap (when a given address is valid for either processor but yields a different physical memory device). In this case, a 3-byte memoryAddress parameter can be specified and the most significant byte of the memoryAddress parameter could then be used to select which physical memory device the tester is attempting to retrieve data from. Any usage of a memoryIdentifier shall be documented in the appropriate CTS, SSTS or supplemental diagnostic specification referenced by the CTS or SSTS.
memorySize The parameter memorySize is used to select the number of consecutive memory addresses (in ascending order) to be read starting at memoryAddress (memorySize includes the starting location). MemorySize shall have a value between one and seven bytes. The maximum value comes from the maximum number of bytes allowed in a UUDT diagnostic response (of the $AA service).

8.11.3 Positive Response Message Definition (Table 126).

Table 126: DefinePIDbyAddress Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DefinePIDbyAddress Positive Response Service Id	M	6D	SIDPR
 #2 #3	parameterIdentifier = [PIDHighByte_1 PIDLowByte_1]	 M M	xx	PID_ B1 B2

8.11.3.1 Positive Response Message Data Parameter Definitions (Table 127).

Table 127: Request Message Data Parameter Definition

Definition
parameterIdentifier The value of the PID contained within the request message. This is an echo of the requested PID number to confirm completion of the PID assignment.

8.11.4 Supported Negative Response Codes (RC_). The following negative response codes (Table 128) shall be implemented for this service.

Table 128: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** Request message is not the correct number of bytes.	M	SFNS-IF
31	**RequestOutOfRange** Note 1 • The PID requested is not located in the range of dynamically definable PIDs. • The memorySize parameter in the request message is greater than 7 or equal to 0. • Any memory address within the interval [$MA, ($MA + $MS -$1)] is invalid. • Any memory address within the interval [$MA, ($MA + $MS -$1)] is restricted. • Any memory address within the interval [$MA, ($MA + $MS -$1)] is secure and the ECU is locked.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** See 7.2 Return Code Definition.	C	RCR-RP

Note 1: A restricted memory address is one that the ECU will not allow a tester to read under any circumstances (e.g., the addresses that contain the security seed and key values). Secure memory addresses are those which the ECU shall not allow the tester to read unless the ECU is unlocked. See service $27 SecurityAccess.

8.11.5 Message Flow Example DefinePIDbyAddress.

8.11.5.1 DefinePIDbyAddress Example with 4-Byte memoryAddress.

In this example (Table 129):

PID $CCFF is dynamically definable,

The ECU uses 4-byte addressing and the tester requests memory data in address $00011102,
The value of the memorySize parameter is $02,
The tester responds with a positive response with PID $CCFF,
The node has a physical request CANId of $241,
The node has a physical USDT response CANId of $641.

Table 129: DefinePIDbyAddress with 4-Byte memoryAddress

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-FF)	$241	$10	$08	$2D	$CC	$FF	$00	$01	$11
N(USDT-FC)	$641	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$241	$21	$02	$02	---	---	---	---	---
N(USDT-SF)	$641	$03	$6D	$CC	$FF	---	---	---	---

8.11.5.2 DefinePIDbyAddress Example with 3-Byte memoryAddress.

In this example (Table 130):
PID $CCFF is dynamically definable,
The ECU uses 3 byte addressing and the tester requests memory data in address $011102,
The value of the MemorySize parameter is $02,
The tester responds with a positive response with PID $CCFF,
The node has a physical request CANId of $241,
The node has a physical USDT response CANId of $641.

Table 130: DefinePIDbyAddress with 3-Byte memoryAddress

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$07	$2D	$CC	$FF	$01	$11	$02	$02
N(USDT-SF)	$641	$03	$6D	$CC	$FF	---	---	---	---

8.11.6 Node Interface Function.

8.11.6.1 Node Interface Data Dictionary (Table 131).

Table 131: Node Interface Data Dictionary of ReadMemoryByAddress Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
Bad_Address Status flag that indicates whether any of the range of addresses in the request is not valid.	TRUE/FALSE

Variable/Meaning	Values
memoryAddress This is the starting address in ECU memory from the request message.	Varies based on ECU
numAddressBytes This is a local variable used to identify the number of bytes in the tester request message for the memoryAddress parameter.	2 to 4, depending on ECU Addressing scheme
memorySize This is the number or consecutive memory addresses that the tester has requested (parameter of request message).	$01 to $07
Memory_Address This is a local variable used to point to ECU memory when indexing through the requested addresses to determine if any portion of the requested memory is invalid, protected, or requires security access. The pseudo code uses Memory_Address.data to reflect the contents of a given memory address.	Varies based on ECU
DynamicPidStructArray[] This is an array of structures that contain the data needed to support dynamic PIDs. Each structure contains the following information: • PidNumber - The dynamic PID number. This is hardcoded in the structure and is not modified by this service. The DynamicPidStructArray contains an array element (which is a structure) for each dynamic PID that the ECU supports. • MemAddr - The starting memory address assigned to this dynamic PID via this service. • Length - The number of bytes to be transmitted starting with the contents of MemAddr when this PID is requested. The pseudo code uses the following convention to access an element of the structure: DynamicPidStructArray[0]. PidNumber would be the PID number of the first dynamic PID structure in the array.	N/A
NumArrayElements This is a local variable that contains the number of elements in the DynamicPidStructArray.	ECU dependent
Index This is a local variable used in the pseudo code to index into the DynamicPidStructArray.	0 to (NumArrayElements -1)
ValidIndex This is a local variable used in the pseudo code to record the value of Index when the PID in the request message matches the PidNumber element of the DynamicPidStructArray.	0 to NumArrayElements

8.11.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $2D message is received, the following logic is executed:

```
BEGINFUNCTION Serv_2D_Msg_Recvd()
Bad_Address ← FALSE
```

```
ValidIndex ← NumArrayElements
IF (message_data_length != (4 + numAddressBytes) ) THEN
    send Negative Response ($7F $2D $12) /*Subfunction Not Supported or Invalid Format */
ELSE
    IF (the response cannot be sent within P2_C ms) THEN
        send Negative Response ($7F $2D $78) /*RequestCorrectlyReceived-ResponsePending */
    ENDIF

    FOR (Index ← $00 TO (NumArrayElements - 1) BY $01)
        IF (parameterIdentifier = DynamicPidStructArray[Index].PidNumber)
            ValidIndex ← Index
            Index ← NumArrayElements /* to exit for loop */
        ENDIF
    ENDFOR

    IF ((ValidIndex = NumArrayElements) OR (memorySize = 0d) OR (memorySize > 7)) THEN
        send Negative Response ($7F $2D $31) /*Request Out Of Range */
    ELSE
        \* Check whether any memory address in the requested interval is protected, restricted (by Security
        Status) or invalid*/
        FOR (Memory_Address ← memoryAddress TO (memoryAddress + memorySize- $01) BY $01)
            IF ( ((Memory_Address is secure) AND (Security_Access_Unlocked = FALSE)) OR
            (Memory_Address is restricted) OR
            (Memory_Address is Invalid) ) THEN
                Bad_Address ← TRUE
                Memory_Address ← (memoryAddress + memorySize) /* exit for loop */
            ENDIF
        ENDFOR
        IF (Bad_Address = TRUE) THEN
            send Negative Response ($7F $2D $31) /*Request Out Of Range */
        ELSE
            DynamicPidStructArray[ValidIndex].MemAddr ← memoryAddress
            DynamicPidStructArray[ValidIndex].Length ← memorySize
            /* send positive response message */
            send ($6D $PID_B1 $PID_B2)
        ENDIF
    ENDIF
ENDIF
ENDFUNCTION
```

Note: The pseudo code above assumes the data contained within the dynamic PID structure is used by the services that provide PID data when these services build a response message.

8.11.7 Node Verification Procedure.

Procedure 1:

1. Send a request to define a valid dynamic PID with one valid memory address (MS = $01) that is not restricted, and non-secured. Verify the proper positive response. Verify that the correct data and number of bytes for the defined PID are reported when requesting the PID with either a $22 or $AA request.
2. Send a request to define a valid dynamic PID with seven consecutive valid memory addresses (MS = $07) that are not restricted, and non-secured. Verify the proper positive response. Verify that the correct data

and number of bytes for the defined PID are reported when requesting the PID with either a $22 or $AA request.

3. If applicable, with the ECU manufacturers enable counter > $00, the vulnerability flag = $FF, and security has been accessed (SecurityAccess ($27) request has been sent), send a request to define a valid dynamic PID with an address range that is completely valid, not restricted, and includes secure locations. Verify the proper positive response. Verify that the correct data and number of bytes for the defined PID are reported when requesting the PID with either a $22 or $AA request.

Procedure 2:

1. Send a request with less than the required number of data bytes (4 + the number of bytes in the memoryAddress parameter), verify negative response ($7F $2D $12).
2. Send a request with more than the required number of data bytes (4 + the number of bytes in the memoryAddress parameter), verify negative response ($7F $2D $12).
3. Send a request with an invalid dynamic PID. Verify negative response ($7F $2D $31).
4. Define a valid dynamic PID with more than seven consecutive valid memory addresses (MS = $08) that are not restricted, and non-secured. Verify negative response ($7F $2D $31).
5. Define a valid dynamic PID with a valid MA but with zero number of memory addresses (MS = $00). Verify negative response ($7F $2D $31).
6. Define a valid dynamic PID with at least one invalid ECU address in the range of addresses specified by MA and MS. Verify negative response ($7F $2D $31).
7. If applicable, define a valid dynamic PID with a valid range of non-secure addresses that includes at least one restricted address. Verify negative response ($7F $2D $31).
8. If applicable, with the ECU manufacturers enable counter = $00, the vulnerability flag < $FF, and security has not been accessed (SecurityAccess ($27) request has not been sent), define a valid dynamic PID with a range of valid and non-restricted addresses that includes at least one secure address. Verify negative response ($7F $2D $31).
9. If applicable, with the manufacturers enable counter > $00 or vulnerability flag = $FF and security has not been accessed (SecurityAccess ($27) request has not been sent), define a valid dynamic PID with a range of valid and non-restricted addresses that includes at least one secure address. Verify negative response ($7F $2D $31).
10. If applicable, with the manufacturers enable counter > $00 or vulnerability flag = $FF and security has been accessed (SecurityAccess ($27) request has been sent), define a valid dynamic PID with a range of valid and non-restricted addresses that includes at least one secure address. Verify positive response.
11. If negative response code $78 is supported by the ECU, then create the conditions under which the ECU should return the $7F $2D $78 response and verify that proper response is sent. Repeat for each possible reason an ECU would send the negative response with response code $78.

8.11.8 Tester Implications. The tester needs to ensure that it transmits the correct number of bytes in the memoryAddress parameter or the message shall be rejected.

If the tester attempts to request a dynamic PID (via the $22, $2C/$AA services) and the PID has not been previously defined via the $2D service, the results are undefined.

If the tester packs a dynamic PID into a dynamic DPID, then attempts to redefine the dynamic PID (using the $2D service) without subsequently redefining the dynamic DPID, then the data transmitted when the DPID is requested is undefined. Anytime a tester has packed a dynamic PID into a dynamic DPID and wants to redefine the dynamic PID, then the tester should first remove the dynamic DPID from the DPID scheduler, redefine the dynamic PID, redefine the dynamic DPID and then put the DPID back into the periodic message scheduler (via the $AA service).

This service should only be requested with physical addressing.

8.12 RequestDownload ($34) Service. This service is used in order to prepare a node to be programmed.

8.12.1 Service Description. The service is used to indicate to the programming routines which encryption and compression techniques are utilized so that the programming routines can correctly decode the data received with subsequent TransferData ($36) services. This is a secured service. Nodes that support security access must be unlocked before a request for this service is accepted (see Mode $27 Request SecurityAccess).

This service is also used by the test device to request a Node to transfer program operation from the application software to the boot software (for nodes that support a boot) so that the node can be programmed. ECUs that utilize the SPS system to program operational software are required to have boot software. See programming chapter within this specification. ECUs that have their software in ROM and utilize the SPS system to program calibrations are not required to support a boot.

The DisableNormalCommunication ($28) and ProgrammingMode ($A5) services must be active prior to the mode $34 request. However, high speed programming (enabled with service $A5 on the Single Wire CAN link) is not required to be active to perform programming. This means that even though the intent is to perform programming in high speed mode (for devices on the low speed link), it will be possible to program these nodes in normal speed if circumstances dictate the need.

Only a single $34 service request is required to initiate a download sequence of multiple software or calibration modules to the node. However, it is also possible to send a $34 service request each time a download of a single software or calibration module starts (clear separation of all downloaded modules).

Note: If Encryption or Compression techniques are utilized, then multiple requests for this service (one per module downloaded) **may** be required as determined by the encryption or compression techniques for each module. Ideally, all software or calibration modules downloaded to a single node should use a single encryption and/or compression technique.

8.12.2 Request Message Definition (Table 132).

Table 132: RequestDownload Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	RequestDownload Request Service Id	M	34	SIDRQ
#2	dataFormatIdentifier	M	xx	DFI_
	unCompressedMemorySize = [			
#3 Note 1	Byte1 - MSB	M	xx	UCMS_B1
#4	Byte2	M	xx	UCMS_B2
#5	Byte3	C1	xx	UCMS_B3
#6	Byte4]	C2	xx	UCMS_B4
Where: **C1:** Byte 3 is present when the unCompressedMemorySize parameter contains 3 or 4 bytes. **C2:** Byte 4 is present when the unCompressedMemorySize parameter contains 4 bytes.				

Note 1: Byte1 in the unCompressedMemorySize parameter is always the most significant byte and the last byte of the unCompressedMemorySize parameter (Byte2 for 2-byte length, Byte3 for 3-byte length, or Byte4 for 4-byte length) is always the least significant byte of the unCompressedMemorySize parameter.

8.12.2.1 Request Message Sub-Function $Level Parameter Definition. There are no sub-function parameters used by this service.

8.12.2.2 Request Message Data Parameter Definition. Table 133 contains the data parameter values defined for this service.

Table 133: Request Data Parameter Definition

Definition
dataFormatIdentifier This data parameter is a one byte value with each nibble encoded separately. The high nibble specifies the **compressionMethod**, and the low nibble specifies the **encryptingMethod**. The value "$00" specifies that neither compressionMethod nor encryptingMethod is used. If no compression or encryption techniques are used, the unCompressedMemorySize bytes are transmitted as part of the request but their values may be ignored by the ECU.
UnCompressedMemorySize This data parameter can vary from two to four bytes based on the needs of the ECU (e.g., length of modules to be downloaded). The same number of bytes shall always be used for this parameter in all RequestDownload service requests for a given ECU. A programming tool determines the number of bytes to transmit on the bus based on data contained in the utility file (reference the programming chapter within this specification and also the Interpreter Specification listed in the Normative References section of this specification for more details). If compression or encryption techniques are used during programming, the data contained in the unCompressedMemorySize parameter shall be used by the Node to compare against the uncompressed size of the total data transferred with subsequent TransferData ($36) service requests. The total data transferred can either be a large block of data that contains all software and calibration modules (single service $34 request), or a block of data that contains a single software or calibration module (multiple service $34 requests, one for each module).

8.12.3 Positive Response Message Definition (Table 134).

Table 134: RequestDownload Response Messages

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	RequestDownload Response Service Id	M	74	SIDPR

8.12.3.1 Positive Response Message Data Parameter Definition. There are no data parameters in a response to a request for this service.

8.12.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 135.

Table 135: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** • Used if the number of data bytes in the request message is incorrect. • Used if the requested dataFormatIdentifier value is not supported by the node. • Used if the unCompressedMemorySize is invalid for the dataFormatIdentifier requested.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** • This return code shall be sent if the DisableNormalCommunication ($28) service is not active when a request is received for this service. • This return code shall be sent if programming mode has not been activated via the ProgrammingMode ($A5) service when a request is received for this service. • This return code shall be sent if the ECU is secure (for ECUs that support the SecurityAccess ($27) service) when a request for this service has been received. • This return code shall be sent if an ECU receives a request for this service while in the process of receiving a download of a software or calibration module. This could occur if there is a data size mismatch between the ECU and the tester during the download of a module.	M	CNCRSE
78	**RequestCorrectlyReceived-ResponsePending** Used when the node cannot immediately process the request.	C	RCR-RP
99	**ReadyForDownload-DTCStored** Used when the node is ready for download and checksum of flash or EEPROM has resulted in a DTC being set.	U	RFD-DS

8.12.5 Message Flow Example.

8.12.5.1 Request Download Positive Response Example (Table 136). In the following example, it is assumed:

- The node has security unlocked.
- Mode $28 is already active.
- Mode $A5 “ProgrammingMode” is already active.
- The node targeted is programmable.
- The node targeted does not support encryption but does support a supplier specific compression technique (dataFormatIdentifier = $10).
- The node targeted uses a 3 byte uncompressedMemorySize parameter with a value of $01 $02 $03 (equals 66051 decimal).
- The target node physical request CANId is $241.
- The target node physical response CANId is $641.

Table 136: RequestDownload - Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$05	$34	$10	$01	$02	$03	---	---
N1(USDT-SF)	$641	$01	$74	---	---	---	---	---	---

8.12.6 Node Interface Function.

8.12.6.1 Node Interface Data Dictionary (Table 137).

Table 137: Node Interface Data Dictionary of RequestDownload Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **TransferData_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
programming_mode_active This is a flag used to keep track of whether or not a programming event has been enabled.	YES/NO

8.12.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $34 message is received and the ECU is currently executing the operational software, the following logic is executed:

```
BEGINFUNCTION Serv_34_Msg_Recvd()
IF ((message_data_length is invalid) OR (dataFormatIdentifier is invalid) OR ((dataFormatIdentifier != $00) AND (unCompressedMemory size is invalid))) THEN
    send Negative Response($7F $34 $12) /* Invalid Format */
ELSE
    /* The security access check below only applies to ECUs which support security access */
    /* the check below for a download in progress is only required for ECUs that do not have boot software. If the ECU supports a boot then this check would be handled out of the service $34 boot code */
    IF ((Security_Access_Unlocked = FALSE) OR (programming_mode_active = NO) OR (a software or calibration module download is in progress)) THEN
        send Negative Response ($7F $34 $22) /* for conditionsNotCorrect */
    ELSE
        IF (more than $P2_C$ ms is needed to process this request) THEN
            send Negative Response ($7F $34 $78) /* RequestCorrectlyReceived-ResponsePending */
        ENDIF
        /* at this time the module shall do whatever is necessary to prepare for transferring program control to the boot software (if the ECU supports boot software) for module programming.*/
        IF (DTC for Flash or EEPROM checksum failure is set) THEN
            send Negative Response ($7F $34 $99) /*ReadyForDownload-DTCStored */
        ELSE
            send ($74) response message
            TransferData_Allowed ← YES

            if applicable, make permanent diagnostic CAN Identifiers available to the boot
            prior to transferring program operation to the boot software.

            if applicable, Transfer program operation to the boot.
    ENDIF
ENDIF
ENDFUNCTION
```

Note: The pseudo code above shows the final response to the $34 service taking place prior to transferring control to the boot software. An ECU may choose to transfer program control to the boot and then send the final response. If the final response is sent before transferring control to the boot, then the utility file should contain a delay op-code to ensure that the programming tool does not attempt to send the next message before the boot software begins executing.

8.12.7 Node Verification Procedure. The procedures below have been written for nodes which support security. Make the appropriate adjustments for nodes which do not support security access.

Procedure 1:

1. Perform security sequence. Send a DisableNormalCommunication request message and verify the positive response. Begin periodic TesterPresent messages. Enable programming mode with the $A5 service (requires two messages). Send RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value. Execute this step when no memory fault exists which would result in a negative response with response code $99. Verify positive response ($74).
2. If the ECU supports the negative response code 99, use development instrumentation to set a memory DTC, Then repeat step one of this procedure and verify the proper negative response ($7F $34 $99, ReadyForDownload-DTCStored).
3. Perform steps 1 and 2 of this procedure a 2nd time with the programming mode enabled in high speed operation (applicable only to devices on Single Wire CAN link).

Procedure 2:

1. Send a RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value without unlocking security, verify the proper negative response message ($7F $34 $22, sequence error) is sent.
2. Perform security sequence. Send a RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value without previously activating mode $28, and verify the proper negative response message ($7F $34 $22, sequence error) is sent.
3. Perform security sequence. Send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Send RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value. Verify that negative response message ($7F $34 $22, conditions not correct) is sent (because programming mode not active).
4. Perform a valid download of data with the download scheme defined, using service $28, $A5, $27 (if applicable), $34 and $36. While the ECU expects more data to be downloaded with the service $36 transmit a service $34 request message and verify the negative response message ($7F $34 $22, conditions not correct) is sent.
5. Send a request message with less than the required amount of data bytes and verify the negative response message ($7F $34 $12) is sent.
6. Send a request message with more than the required amount of data bytes and verify the negative response message ($7F $34 $12) is sent.
7. Perform security sequence. Send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Enable programming mode with the $A5 service. Send RequestDownload message with the correct number of data bytes and an invalid dataFormatIdentifier parameter value. Verify proper negative response reject message ($7F $34 $12, sub-function not supported) is sent.

8. Perform security sequence. Send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Enable programming mode with the $A5 service. Send RequestDownload message with a valid dataFormatIdentifier and an invalid unCompressedMemorySize parameter value. Verify proper negative response reject message ($7F $34 $12, sub-function not supported) is sent.
9. With the manufacturers enable counter > $00 or vulnerability flag = $FF and security has not been accessed (SecurityAccess ($27) request has not been sent), send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Enable programming mode with the $A5 service. Send RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value. Verify that negative response message ($7F $34 $22, conditions not correct) is sent (because security has not been accessed).
10. With the manufacturers enable counter > $00 or vulnerability flag = $FF, perform security sequence. Send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Enable programming mode with the $A5 service. Send RequestDownload message with a valid dataFormatIdentifier and unCompressedMemorySize parameter value. Verify positive response message.

Procedure 3: (If negative response code $78 is supported by the ECU).

1. Perform security sequence. Send a DisableNormalCommunication request message, verify positive response. Begin TesterPresent message periodic. Enable programming mode with the $A5 service. Create the conditions under which the ECU should return a $7F $34 $78 response and send a valid request for this service. Verify that proper response is sent.
2. Repeat previous step of this procedure for each possible reason an ECU would send the negative response with response code $78.

8.12.8 Tester Implications. This service should only be physically addressed (point to point). The number of bytes to use in the uncompressedMemorySize parameter is determined from data within the utility file.

The device must be unlocked prior to requesting data downloads to the device. While this mode does not require TesterPresent messages, the following modes, which are required to be active prior to requesting mode $34, do:

- $27 (to keep the device unlocked if the download block is a secured memory area).
- $28 (to keep normal communications disabled).
- $A5 (to keep programming mode active).

8.13 TransferData ($36) Service. This service is used to transfer and/or execute a block of data, usually for reprogramming purposes.

8.13.1 Service Description. The TransferData service shall be used to download a block of data, download and execute a block of data, execute a resident routine, or execute a previously downloaded block of data. To execute a previously downloaded data block, use a download and execute transfer type (sub-parameter = $80) with zero data bytes being transferred. A node resident routine could be executed using the download and execute transfer type (sub-parameter = $80) and identifying the routine in the startingAddress parameter or by putting a routine number in the dataRecord field.

This service is indirectly secured by the RequestDownload ($34) service. The RequestDownload ($34) service must be performed successfully prior to executing this service in order to prepare the targeted node to accept the TransferData message(s). Once service $34 has been performed successfully, it shall be possible to carry out one or more TransferData service requests.

Note: Successful completion of a RequestDownload ($34) request also requires that security must be unlocked (via SecurityAccess $27 service if applicable), that normal communications have been disabled (via DisableNormalCommunication $28 service) and that the ProgrammingMode ($A5) service has been activated. These services ($27, $28, and $A5) require the tester to send TesterPresent ($3E) requests at least once every $P3_C$ ms in order to keep them active.

A single TransferData message is limited to a maximum of 4095 total data bytes (including the Service Identifier byte, sub-function $Level parameter, address bytes, and data bytes). An ECU with limited RAM resources may further restrict the maximum number of bytes transferred with a single request of this service. During SPS programming, the programming tool determines the maximum number of bytes to transfer from information in the utility file (reference the programming chapter within this specification and also the Interpreter Specification listed in the Normative References section of this specification for more details).

8.13.2 Request Message Definition. Table 138 defines the structure of the TransferData request message.

Note: A request message must always include the sub-function parameter level of operation AND a starting address. Additional data bytes beyond the starting address are optional and would not be present in the case of a request to execute a previously downloaded block of data.

The starting address parameter (startingAddress) can be two, three, or four bytes based on the ECU internal addressing scheme. All TransferData requests to the same ECU shall use the same number of bytes in the startingAddress parameter (independent of the actual address value). During SPS programming, the programming tool determines the number of bytes to use in the startingAddress parameter from information in the utility file.

Table 138: TransferData Request Message

Data Byte Note 1	Parameter Name	Cvt	Hex Value	Mnemonic
#1	TransferData Request Service Id	M	36	SIDRQ
#2	sub-function = [download downloadAndExecuteOrExecute]	M	 00 80	LEV_ DL DLEXOE
 #3 #4 #5 #6	startingAddress[] = [Byte1 - MSB Byte2 Byte3 Byte4]	 M M C1 C2	 xx xx xx xx	 SA_B1 SA_B2 SA_B3 SA_B4
 #5/6/7 : #n	dataRecord[] = [data byte #1 : data byte #m (where m ≤ 4093 minus the number of bytes in the startingAddress)]	C3	 yy : zz	DREC_ DATA_1 : DATA_m

Where:
C1: Byte 3 is present when the memoryAddress parameter contains 3 or 4 bytes.
C2: Byte 4 is present when the memoryAddress parameter contains 4 bytes.
C3: The size of the dataRecord parameter shall be at least one byte if the sub-function value is of type download and may be 0 bytes if the sub-function value is of type downloadAndExecuteOrExecute

Note 1: Byte1 in the memoryAddress parameter is always the most significant byte of the address and the last byte of the memoryAddress parameter (Byte2 for 2-byte addressing, Byte3 for 3-byte addressing, or Byte4 for 4-byte addressing) is always the least significant byte of the memoryAddress parameter.

8.13.2.1 Request Message Sub-Function $Level Parameter Definition. The following definitions apply to the sub-function levels of operation for service $36 TransferData. All other sub-function values are reserved for future definition within this specification. See Table 139.

Table 139: Definition of Sub-function Values

Hex	Definition	Cvt
00	**Download** This sub-parameter level of operation is used to command a node to receive a block transfer and (only) download the data received to the memory address specified in the startingAddress[] parameter.	M
80	**downloadAndExecuteOrExecute** This sub-parameter level of operation is used to command a node to receive a block transfer, download the data received to the memory address specified in the startingAddress[] parameter, and execute the data (program) downloaded. This sub-parameter command can also be used to execute a previously downloaded program by sending the request message with no data in the dataRecord[ ].	U

8.13.2.2 Request Message Data Parameter Definition (Table 140).

Table 140: TransferData Data Parameter Definition

Definition
startingAddress[] The starting address is a 2-, 3-, or 4-byte data parameter (based on ECU internal addressing scheme) which tells the ECU the starting address of product memory to put the data being sent by the tester.
dataRecord[] This is the portion of the request message which contains the data which will be written to ECU memory. The length of this field is variable.

8.13.3 Positive Response Message Definition (Table 141).

Table 141: TransferData Positive Response Messages

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	Transfer Data Positive Response Service Id	M	76	SIDPR

8.13.3.1 Positive Response Message Data Parameter Definition. There are no data parameters in a response to a request for this service.

8.13.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. See Table 142.

Table 142: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This return code shall be sent if: 1. The request message sub-function parameter is not valid. 2. The message length is too short for the sub-function parameter requested.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** This return code shall be sent if: The RequestDownload ($34) service is not active when a request for this service is received. The Boot Software Programming Executive determines that the order of software and/or calibration parts downloaded is invalid. The CTS, SSTS, or appropriate diagnostic specification referenced by either the CTS or SSTS must describe the conditions under which this usage is allowed.	M	CNCRSE
31	**RequestOutOfRange** This return code shall be sent if: The address specified in the request is invalid or the starting address + length of the data equals an invalid address.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** This return code shall be sent if: After receiving a valid TransferData message, the node cannot process the request and send the final response within $P2_C$ ms.	C	RCR-RP
83	**VoltageOutOfRangeFault (High/Low)** This return code shall be sent if: The voltage measured at the primary power pin of the ECU is out of the acceptable range.	M	VOLTRNG
85	**GeneralProgrammingFailure** This return code shall be sent if: • The ECU detects an error when erasing or programming a memory location in the permanent memory device (e.g., Flash Memory). • The programming executive detects a checksum or cyclical redundancy check (CRC) fault with a data file that was downloaded. • The programming executive detects an incompatibility between the boot software, operational software or calibration files (if required.) Refer to paragraph 9.3.2.3.4.	M	PROGFAIL
89	**DeviceTypeError** This return code shall be sent if: The ECU detects an incompatibility between the programming algorithm(s) downloaded and the type of permanent memory device.	M_1	DEV_TYPE_ERR

Where:

M_1 = This return code is required if the SPS utility file contains different routines for the various suppliers of permanent memory that are used for a given ECU.

8.13.5 Message Flow Example Transfer Data.

8.13.5.1 TransferData Positive Response Example (ECU Uses 3-Byte Address). The following example (Table 143) assumes:

- The node uses 3-byte addressing (e.g., startingAddress is three bytes).
- The node targeted is programmable.
- The node has security unlocked.
- Mode $28 is already active.
- Mode $A5 is already active (programming mode enabled).
- A Mode $34 request was sent and a positive response resulted.
- The tester is downloading and executing a routine (sub-function = $80).
- The routine (dataRecord[]) to be transferred contains 250 bytes of data (so message_data_length = 250 + 5 or 255 bytes. The 5 bytes come from the Service ID plus the sub-function parameter byte plus the 3-byte startingAddress).
- The starting address ($00 $23 $FF) plus the number of data bytes all fall into valid ECU memory address space.
- It takes the node longer than P2$_C$ ms to completely download and execute the routine (so the node sends a negative response with response code $78, and eventually sends a positive response (within the enhanced P2$_{CE}$* timing for programming) message which indicates that the block of data is successfully received and processed by the ECU.
- The target node physical request CANId is $241.
- The target node physical USDT response CANId is $641.
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.
- TesterPresent messages are sent (but are not shown in the example).

Table 143: TransferData Positive Response Example

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-FF)	$241	$10	$FF	$36	$80	$00	$23	$FF	aa
N(USDT-FC)	$641	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$241	$21	bb	cc	dd	ee	ff	gg	hh
:									
T(USDT-CF)	$241	$2x	ww	xx	yy	zz	---	---	---
N(USDT-SF)	$641	$03	$7F	$36	$78	---	---	---	---
:									
N(USDT-SF)	$641	$01	$76	---	---	---	---	---	---

8.13.5.2 TransferData Negative Response Example (ECU Uses 2 Byte Address). The following example (Table 144) assumes:

- The node uses 2-byte addressing (e.g., startingAddress is 2 bytes).
- The node targeted is programmable.
- The node has security unlocked.
- Mode $28 is already active.
- Mode $A5 is already active (programming mode enabled).
- A Mode $34 request was sent and a positive response resulted.
- The starting address ($23 $FF) plus the number of data bytes all fall into valid ECU memory address space.
- The programming routines are executing and the address space to be written to has already been erased. The tester is now sending data to be written to permanent memory with the download transfer type (sub-function = $00).
- The data (dataRecord[]) to be transferred contains 250 bytes of data (so message_data_length = 250 + 4 or 254 bytes. The 4 bytes come from the Service ID plus the sub-function parameter byte plus the 2-byte startingAddress).
- It takes the node longer than $P2_C$ ms to completely download and write the data block (so the node sends a negative response with response code $78 to indicate to the tester that the final response shall be sent within the $P2_{CE*}$ enhanced programming timing window). While writing the data to permanent memory, the ECU detects an error so it sends a negative response to indicate that the programming failed.
- The target node physical request CANId is $241.
- The target node USDT physical response CANId is $641.
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.
- TesterPresent messages are sent (but not shown in the example).

Table 144: TransferData Negative Response Example

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-FF)	$241	$10	$FE	$36	$00	$23	$FF	aa	bb
N(USDT-FC)	$641	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$241	$21	cc	dd	ee	ff	gg	hh	ii
:									
T(USDT-CF)	$241	$2x	xx	yy	zz	---	---	---	---
N (USDT-SF)	$641	$03	$7F	$36	$78	---	---	---	---
:									
N (USDT-SF)	$641	$03	$7F	$36	$85	---	---	---	---

8.13.6 Node Interface Function.

8.13.6.1 Node Interface Data Dictionary (Table 145).

Table 145: Node Data Dictionary of TransferData Service Pseudo Code

Variable/Meaning	Values
message_data_length **TransferData_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
C_NumAddrBytes This is a constant that is used in the pseudo code to handle different length values in the startingAddress parameter.	Set to number of bytes in startingAddress parameter

8.13.6.2 Node Interface Pseudo Code. If a node supports boot software then the operational software only needs minimum support of this service. This is true for nodes with boot software because a service $34 transitions the program operation back to the boot software (see service $34) and the $34 service is required before the $36 service is allowed. ECUs with boot software only need to support the Serv_36_Msg_Recvd() pseudo code in the operational software and the Boot_36_Msg_Recvd() in the boot software. ECUs that do not have boot software shall support the functionality of the Boot_36_Msg_Recvd() pseudo code in the operational software.

```
BEGINFUNCTION Serv_36_Msg_Recvd()
send Negative Response ($7F $36 $22)/* conditions not correct since the $34 service is not active if the ECU
is executing the operational software. */
ENDFUNCTION
BEGINFUNCTION Boot_36_Msg_Recvd()
/* The check below for $Level $80 is only needed if that sub function is supported by the ECU */
IF ( (($Level != $00) AND ($Level != $80)) OR
(message_data_length < (2+C_NumAddrBytes)) OR
(($Level = $00) AND (message_data_length < (3+C_NumAddrBytes ))) ) THEN
    send Negative Response ($7F $36 $12) /* invalid format */
ELSE IF (TransferData_Allowed = NO) THEN
    send Negative Response ($7F $36 $22)
ELSE IF(($startingAddress is invalid) OR ($startingAddress + (message_data_length - $02 -
C_NumAddrBytes) is invalid)) THEN
    send Negative Response ($7F $36 $31) /* Request out of range */
ELSE
    SELECT FIRST
    WHEN ($Level = $DL)/* $00 Download */
        IF (the node cannot process the request within P2_C) THEN
            send Negative Response ($7F $36 $78) /* RequestCorrectlyReceived-ResponsePending */
            perform any necessary actions to the data block /*, i.e., program to flash memory etc.*/
        ENDIF
        IF (errors are detected) THEN
            send the appropriate negative response ($7F $36 $22) or ($7F $36 $83) or ($7F $36 $85).
        ELSE
            send ($76) /* programming of block successfully completed */
        ENDIF
    /* The psuedo code from this point until the ENDSELECT statement is only needed if $level $80 is
    supported by the ECU */
    WHEN ($Level = $DLEXOE) /* $80 Download and Execute or just Execute */
        IF (the node cannot process the request within P2C) THEN
            send Negative Response ($7F $36 $78) /* RequestCorrectlyReceived-ResponsePending */
        ENDIF
        /* Execute the code (typically a programming routine)at the downloaded address. */
        CALL (function at $startingAddress)
        IF (errors are detected) THEN
            send the appropriate negative response ($7F $36 $83) or ($7F $36 $85) or ($7F $36 $89)
        ELSE
            send ($76) /* download complete */
        ENDIF
    ENDSELECT
ENDIF
ENDFUNCTION
```

8.13.7 Node Verification Procedure.

Procedure 1: (Enable Programming Session).

1. Perform the steps necessary to activate ProgrammingMode (reference service $A5 for proper procedure). Begin sending periodic TesterPresent ($3E) messages to satisfy the $P3_C$ timing requirements. Perform the necessary steps to unlock ECU security (reference service $27) if applicable. Next, send a valid mode $34 message and verify the positive response.

Procedure 2:

1. Perform Procedure 1 and then send a block of data using mode 36 and verify the $76 positive response.
2. If it takes more than $P2_C$ ms to write the block of data sent in step 1 of this procedure, verify that the node sends a $7F $36 $78 within the $P2_C$ ms required time, followed by a $76 positive response.
3. Repeat step 1 (and 2) as necessary to transfer all data necessary to complete the download of the module or programming of the node and verify the $76 positive response after each subsequent block transfer and the final mode $36 message.
4. If it takes more than $P2_C$ ms to write any data block, the node shall send a $7F $36 $78 within the $P2_C$ ms required time after each transfer of a block of data, followed by a $76 response when the node has completed writing the block of data.

Procedure 3:

1. Perform Procedure 1 and then send a mode $36 with an invalid $Level value; verify the $7F $36 $12 response.
2. Perform Procedure 1 and then send a mode $36 request with ONLY the $Level data byte (starting address bytes missing), verify the node responds with $7F $36 $12.
3. Perform Procedure 1 and then send a mode $36 request with $Level $00, a valid starting address and no additional data. Verify the node responds with $7F $36 $12.
4. Perform Procedure 1 and then send a mode $36 request $Level $00 with valid data and an invalid starting address, verify the node responds with $7F $36 $31.
5. Perform Procedure 1 and then send a mode $36 request with $Level $00, a valid starting address, and a data block size that results in an invalid address. Verify that the node responds with $7F $36 $31.
6. Perform Procedure 1 without sending the valid mode $34 request. Then send a valid mode $36 and verify the $7F $36 $22 response.
7. Create a condition that will cause the programming algorithm to detect an error (e.g., put incorrect checksum or CRC in a calibration file if the programming algorithm includes a module based checksum or CRC verification) and verify the $7F $36 $85 response.

8. Prior to programming, increase/reduce the voltage level to a value where the device is not capable of being programmed. Verify that the node sends back a $7F $36 $83 negative response to a request for this service.

Procedure 4: Optional. These additional steps should be performed if $Level $80 is implemented.

1. Perform Procedure 1 and then send a mode $36 request with ONLY the $Level data byte (starting address bytes missing), verify the node responds with $7F $36 $12.
2. Perform Procedure 1 and then send a mode $36 request with $Level $80 and an invalid starting address, verify the node responds with $7F $36 $31.
3. Perform Procedure 1 and then send a mode $36 request with $Level $80, a valid starting address, and a data block size that results in an invalid address. Verify that the node responds with $7F $36 $31.

Procedure 5: Optional. To be verified if the node requires a specific order in which software/calibration files are downloaded.

1. Obtain an archive file or a utility file which will cause the software/calibration parts to be downloaded in an order that is unacceptable to the boot software programming executive. Attempt to program the node using a programming tool (e.g., DPS Tool or SPS Tool created by Service Operations). Verify that the programming event fails as a result of a $7F $36 $22 response at the appropriate time.

Procedure 6: Optional - to be performed if the ECU supports negative response code $89 (DEVICE_TYPE_ERROR).

1. Obtain ECUs with each possible type of permanent memory device. Perform Procedure number 1 and then download one of the allowed programming algorithms. Verify that the negative response $7F $36 $89 is sent if the algorithm downloaded was not the correct one for that permanent memory device. Verify that the positive response is sent if the correct algorithm was downloaded.
2. Repeat step 1 of this procedure for each possible combination of programming algorithm and permanent memory device.

Procedure 7: Optional. To be verified if the node requires data file compatibility checks.

1. Obtain an archive file that contains data files that are not compatible (either operational software not compatible with the boot software or a calibration file that is not compatible with the operational software). Attempt to program the node using a programming tool (e.g., DPS Tool or SPS Tool created by Service Operations). Verify that the programming event fails as a result of a $7F $36 $85 response at the appropriate time.

8.13.8 Tester Implications. This service should only be physically addressed (point to point). The size of the startingAddress parameter is determined via information in the utility file.

8.14 WriteDataByIdentifier ($3B) Service. The purpose of this service is to provide the ability to change (write/program) the content of pre-defined ECU data referenced by a dataIdentifier (DID) which contains static information like ECU identification data, or other information which does not require **real-time** updates.

8.14.1 Service Description. The tester is required to provide the dataIdentifier value desired, while the target ECU shall be responsible for knowing the address location and block length corresponding to the requested dataIdentifier. An ECU is not required to support all corporate standard DIDs, and may support additional application specific DIDs. Application specific DIDs supported by this service shall be documented in the ECU's CTS or supplemental diagnostic specification referenced in the CTS.

Note: Appendix C of this specification contains a list of corporate standard DIDs.

It shall be possible for Nodes to restrict the tester's ability to write certain dataIdentifiers based on the security status of the node.

8.14.2 Request Message Definition. The length of the request message is dependant upon the size of the data referenced by the dataIdentifier parameter. Only one dataIdentifier which is supported by the ECU shall be included in the request message. See Table 146.

Table 146: WriteDataByIdentifier Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	WriteDataByIdentifier Request Service Id	M	3B	SIDRQ
#2	dataIdentifier	M	xx	DID_
 #3 : #n	dataRecord[] = [data#1 : data#m]	 M : U	 xx : xx	DREC_ DTA_1 : DTA_m

8.14.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. There are no sub-function parameters used by this service.

8.14.2.2 Request Message Data Parameter Definition. Table 147 specifies the data parameter definitions for this service.

Table 147: Request Data Parameter Definition

Definition
dataIdentifier The dataIdentifier (DID) is used within this service in the request message to indicate to the ECU application, which static data the tool is requesting to write. The dataIdentifier parameter in the response message is an echo of the one in the request message to confirm to the tool that the positive response is for the DID requested. If an ECU supports application specific DIDs (non-corporate standard), then the DID number, the data contents, and other information contained within the tables in Appendix C, must be documented within the ECU CTS or within a supplemental diagnostic specification referenced by the CTS.
dataRecord[] This is the portion of the request message which contains the data which will be written to ECU memory. The length of this field is dependant upon the dataIdentifier requested.

8.14.3 Positive Response Message Definition (Table 148).

Table 148: WriteDataByIdentifier Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	WriteDataByIdentifier Positive Response Service Id	M	7B	SIDPR
#2	dataIdentifier	M	xx	DID_

8.14.3.1 Positive Response Message Data Parameter Definition. Table 149 specifies the data parameter definitions for this service.

Table 149: Response Message Data Parameter Definition

Definition
dataIdentifier The dataIdentifier parameter in the response message is an echo of the one in the request message to confirm to the tool that the positive response is for the DID requested.

8.14.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. See Table 150.

Table 150: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This response code shall be sent if the length of the data in the request message does not match the size expected by the node.	M	SFNS-IF
22	**ConditionsNotCorrectOrRequestSequenceError** This response code shall be sent if the operating conditions of the ECU are such that it cannot perform the required action (e.g., the data for a DID is stored in EEPROM and an EEPROM failure has occured). This response code is not intended to be used if the ECU conditions are such that the request temporarily cannot be performed (e.g., the data temporarily cannot be written to EEPROM because another application task is currently writing to EEPROM). Response code $78 is used when conditions are temporarily not met. If this response code is supported by a node, the reason(s) to generate this return code must be documented in the Component Technical Specification (CTS).	C	CNC-RSE
31	**RequestOutOfRange** This response code shall be sent if: • The dataIdentifier in the request message is not supported in the node or the dataIdentifier is supported for read only purposes (via service $1A). • The dataIdentifier, which references a specific address, is secured and the ECU is not in an unlocked state. • Any data transmitted in the dataRecord of the request message is invalid (if applicable to the node).	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** • This response code shall be sent if writing the data to the address(es), which are referenced by the dataIdentifier, takes longer than $P2_C$ ms. (e.g., the block of data sent is large enough that writing of the data to EEPROM cannot be completed within $P2_C$). • This response code shall be sent if the ECU operating conditions are such that the request temporarily cannot be performed (e.g., the data temporarily cannot be written to EEPROM because another application task is currently writing to EEPROM).	C	RCR-RP

8.14.5 Message Flow Example WriteDataByIdentifier Service.

8.14.5.1 WriteDataByIdentifier(dataIdentifier=VIN). The example below shows how the WriteDataByIdentifier(dataIdentifier=VIN) service shall be implemented. The example assumes the following information to be true:

- The point to point request CANId for the ECU is $241.
- The USDT response CANId of the ECU is $641.
- The Data Identifier for the VIN is $90 (as defined in Appendix C).
- The USDT flow control parameters Block Size (BS) and Separation Time (ST) are both $00.

8.14.5.2 WriteDataByIdentifier(dataIdentifier). (Table 151)

Table 151: WriteDataByIdentifier(dataIdentifier=VIN)

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-FF)	$241	$10	$13	$3B	$90	"W"	"0"	"L"	"0"
N(USDT-FC)	$641	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$241	$21	"J"	"B"	"F"	"3"	"5"	"W"	"1"
T(USDT-CF)	$241	$22	"0"	"4"	"2"	"7"	"6"	"5"	---
N(USDT-SF)	$641	$02	$7B	$90	---	---	---	---	---

8.14.6 Node Interface Function.

8.14.6.1 Node Interface Data Dictionary (Table 152).

Table 152: Node Interface Data Dictionary of WriteDataByIdentifier Service Pseudo Code

Variable/Meaning	Values
message_data_length **Security_Access_Unlocked** **Security_Access_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.14.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a $3B message is received, the following logic is executed:

```
BEGINFUNCTION Serv_3B_Msg_Recvd()
IF ($dataIdentifier is NOT supported)
    send Negative Response ($7F $3B $31) /* RequestOutOfRange */
ELSE IF (((message_data_length - 2)!= the expected length for $dataIdentifier) OR (message_data_length <
3)) THEN
    send Negative Response ($7F $3B $12) /* Invalid Format */
ELSE IF ((any Data value in dataRecord[] is invalid for $dataIdentifier) OR
(($dataIdentifier requires security access) AND
(Security_Access_Unlocked = FALSE)) OR (($dataIdentifier requires security code access) AND
(Security_Access_Allowed = FALSE))) THEN
    send Negative Response ($7F $3B $31) /* RequestOutOfRange */
ELSE IF (Conditions are not correct for writing to $dataIdentifier) THEN
    send Negative Response ($7F $3B $22) /* ConditionsNotCorrect */
ELSE
    IF (writing to $dataIdentifier will take more than $P2_C$ ms) THEN
        send Negative Response ($7F $3B $78) /* RequestCorrectlyReceived-ResponsePending */
    ENDIF
    Write data values from dataRecord[] to the memory address associated with the $dataIdentifier
    send ($7B $dataIdentifier) positive response message
```

```
ENDIF
ENDFUNCTION
```

8.14.7 Node Verification Procedure.

Procedure 1:

1. Send a $3B message for each $dataIdentifier supported including good data and verify positive response message (test data formatting).

Procedure 2:

1. Send a $3B message with an invalid $dataIdentifier parameter and verify the negative response ($7F $3B $31 - RequestOutOfRange).
2. Send a $3B message with no data included after the Service Identifier and verify the negative response ($7F $3B $12 - InvalidFormat).
3. Send a $3B message with a valid dataIdentifier and an invalid number of data bytes after the dataIdentifier and verify the negative response ($7F $3B $12 - InvalidFormat).
4. Repeat step 3 for each dataIdentifier supported.

Procedure 3: (If ECU implements this RC_).

1. Send a $3B message with a valid dataIdentifier and associated data bytes at a time when ECU internal conditions would not allow the data to be written (e.g., EEPROM failure) and verify that the ECU sends the correct negative response ($7F $3B $22 - ConditionsNotCorrect).

Procedure 4: (If ECU supports secure DIDs).

1. Send a $3B message with a secured dataIdentifier and associated data bytes after the data - parameter to a secured ECU and verify the negative response ($7F $3B $31 - RequestOutOfRange).
2. Send a $3B message with a secured dataIdentifier and associated data bytes after the dataIdentifier when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has not been accessed (SecurityAccess ($27) request has not been sent) and verify the negative response ($7F $3B $31 - RequestOutOfRange).
3. Send a $3B message with a secured dataIdentifier and associated data bytes after the dataIdentifier when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has been accessed (SecurityAccess ($27) request has been sent) and verify the positive response (to unlock an ECU see SecurityAccess service).
4. Repeat steps 1 and 2 for each DID that requires security.

Procedure 5: (If ECU implements this RC_).

1. Send a $3B message with a dataIdentifier and associated data bytes after the sub-parameter to the ECU when the ECU needs more than $P2_C$ ms to write data bytes to memory and verify the negative response ($7F $3B $78 - RequestCorrectlyReceived-ResponsePending) within $P2_C$ ms and eventually a positive response message.

Procedure 6: (If applicable).

1. If the ECU does any type of validity checks on the data being written, send a request with invalid data and verify the $7F $3B $31 negative response.

Procedure 7: (Only applicable if ECU has DIDs that require the usage of a security code).

Note: Security code as defined in the Vehicle Theft Deterrent SSTS

1. Send a $3B message with a security code required dataIdentifier and associated data bytes after the dataIdentifier to a secured ECU (security code has not been entered) and verify the negative response ($7F $3B $31 - RequestOutOfRange).
2. Send a $3B message with a security code required dataIdentifier and associated data bytes after the dataIdentifier when the security code has been entered and verify the positive response.
3. Repeat steps 1 and 2 for each DID that requires the security code.

8.14.8 Tester Implications. This service should be used with physical addressing.

8.15 TesterPresent ($3E) Service. This service is used to indicate to a node (or nodes) that a tester is still connected to the vehicle and that certain diagnostic services that have been previously activated are to remain active. Some diagnostic services require that a tester send a request for this service periodically in order to keep the functionality of the other service active. Documentation within this specification of each diagnostic service indicates if a given service requires the periodic TesterPresent request to remain active.

8.15.1 Service Description. This service keeps other diagnostic services active by resetting the diagnostic timer (TesterPresent_Timer) each time a request for this service is received. A portion of the ECU diagnostic application executes in the background that modifies and tests the value of the timer (modification based on the processing loop time). When the value of the TesterPresent_Timer meets or exceeds the value of the $P3_C$ application timer, a TesterPresent or $P3_C$ timeout occurs.

Note: See the paragraphs within this specification on application timing requirements for more information on $P3_C$.

When a $P3_C$ timeout occurs, the node shall execute the same logic that would be executed if a ReturnToNormalMode ($20) service request was received. This includes the transmission of a ReturnToNormalMode positive response message. (See note below) Nodes are required to time out no sooner than $P3_C$ and no later than $P3_{Cmax}$.

Note: The unsolicited service $20 positive response is only sent if programming mode was not active prior to the TesterPresent timeout. See service $A5 for more information on programming mode.

There shall only be a response to a request for this service if the request is physically addressed.

Note: Receiving a TesterPresent request message shall not put a node into a diagnostic mode (i.e., the TesterPresent_Timer_State shall not be set to ACTIVE by this message; see pseudo code).

8.15.2 Request Message Definition (Table 153).

Table 153: TesterPresent Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	TesterPresent Request Service Id	M	3E	SIDRQ

8.15.2.1 Request Message Sub-Function $Level Parameter Definition. There are no sub-function parameters used by this service.

8.15.2.2 Request Message Data Parameter Definition. This service does not contain a data-parameter(s).

8.15.3 Positive Response Message Definition. (Table 154) The use of any response, either positive or negative is dependant on how the request message was addressed. If the request message is addressed functionally (to multiple nodes using the AllNodes $101 CANId) then there shall be no response, either positive or negative. If the request message is addressed physically (using the nodes point to point diagnostic request CANId) then there shall be a response, either positive or negative.

Table 154: TesterPresent Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	TesterPresent Positive Response Service Id	M_1	7E	SIDPR
Where: **M_1** = The TesterPresent positive response message shall only be sent if the request message was physically addressed.				

8.15.3.1 Positive Response Message Data Parameter Definition. There are no data parameters used by this service in the positive response message.

8.15.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. See Table 155.

Table 155: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** If the length of the request message is invalid.	M_2	SFNS-IF
Where: $\mathbf{M_2}$ = A negative response message shall only be sent if the request message was invalid for the above reasons and the request was physically addressed.			

8.15.5 Message Flow Example TesterPresent.

8.15.5.1 TesterPresent. The tester sends a TesterPresent request message to a physical node, which requires a response message from the addressed node. See Table 156.

Network parameter:

- Node physicalRequestCANId = $242.
- Node USDTResponseCANId = $642.

Table 156: Physically Requested TesterPresent (response required)

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$242	01	$3E	---	---	---	---	---	---
N(USDT-SF)	$642	$01	$7E	---	---	---	---	---	---

The tester sends a TesterPresent request message to a functional system. See Table 157.

Network parameter:

- Functional System Address (FSA): $FE = All nodes.
- The nodes shall not send a positive response message.

Table 157: Functionally Addressed TesterPresent (no response allowed)

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$3E	---	---	---	---	---
no positive response messages									

8.15.6 Node Interface Function.

8.15.6.1 Node Interface Data Dictionary (Table 158).

Table 158: Node Interface Data Dictionary of TesterPresent Service Pseudo Code

Variable/Meaning	Values
message_address_type **message_data_length** **TesterPresent_Timer_State** **diagnostic_responses_enabled** **TesterPresent_Timer**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.15.6.2 Node Interface Pseudo Code.

Powerup States:

None

Each time a TesterPresent ($3E) message is received, the following logic is executed:

```
BEGINFUNCTION Serv_3E_Msg_Recvd()
IF (message_data_length = 1) THEN
    TesterPresent_Timer ← 0 /*Reset the tester present timer*/
    /* send a pos. resp. if request was physically addressed and diagnostic responses are enabled*/
    IF ((message_address_type = PHYSICAL) AND (diagnostic_responses_enabled = YES)) THEN
        send ($7E) response message
    ENDIF
ELSE
    IF (message_address_type = PHYSICAL) THEN

        send Negative Response ($7F $3E $12) InvalidFormat
    ENDIF
ENDIF
ENDFUNCTION
```

The following logic is used to implement the TesterPresent_Timer in the node’s main processing loop:

```
BEGINFUNCTION Serv_3E_Background_Logic()
IF (TesterPresent_Timer_State = ACTIVE) THEN
    increment TesterPresent_Timer by the length of the main processing loop
    IF (TesterPresent_Timer ≥ $P3_C$) THEN
        Call Exit_Diagnostic_Services() /* function in service $20 */
    ENDIF
ENDIF
ENDFUNCTION
```

8.15.7 Node Verification Procedure.

Procedure 1:

1. Request message length test: Send a physically addressed TesterPresent ($3E) request message with an invalid message length (not equal to 1) and verify that a negative response message with response code $12 is sent by the node (ECU).

Procedure 2:

1. Send a service request message which requires a TesterPresent service to remain active (excluding service $A5).
2. Send functionally addressed TesterPresent ($3E) request messages with an invalid message length (not equal to 1) and verify that no negative response message is sent by the node (ECU).
3. Verify that the node (ECU) sends a ReturnToNormalMode positive response message between $P3_C$ and $P3_{Cmax}$.

Procedure 3: (Uses Functional TesterPresent message).

1. Send a service request message to the node using a service that requires a TesterPresent to keep its functionality active (e.g., schedule periodic DPID via $AA service or use DeviceControl $AE service). Pick a service where it can be easily verified that the functionality remains active if TesterPresent messages are sent.
2. Wait 2 s.
3. Send a valid functionally addressed TesterPresent ($3E) request message using the AllNodes CANId ($101) + AllNodes extended address ($FE).
4. Verify that no TesterPresent positive response message is sent by the node (ECU).
5. Send no further messages and wait for a $P3_C$ timeout to occur. Verify that the node (ECU) sends a ReturnToNormalMode positive response message between $P3_C$ and $P3_{Cmax}$. Also verify that the functionality of the other service has stopped once the ReturnToNormalMode positive response message has been sent.

Procedure 4: (Uses Physical TesterPresent message).

1. Send a service request message to the node using a service that requires a TesterPresent to keep its functionality active (e.g., schedule periodic DPID via $AA service or use DeviceControl $AE service). Pick a service where it can be easily verified that the functionality remains active if TesterPresent messages are sent.
2. Wait 2 s.
3. Send a valid physically addressed TesterPresent ($3E) request message.
4. Verify that a TesterPresent positive response message is sent by the node (ECU).
5. Send no further messages and wait for a $P3_C$ timeout to occur. Verify that the node (ECU) sends a ReturnToNormalMode positive response message between $P3_C$ and $P3_{Cmax}$. Also verify that the functionality of the other service has stopped once the ReturnToNormalMode positive response message has been sent.

Procedure 5:

1. Repeat Procedures 3 and 4 only continue to send TesterPresent messages within the $P3_C$ timing window for at least one minute. Verify that the service activated in step 1 of these procedures continues to function as long as the TesterPresent messages are sent. After the $P3_C$ timeout occurs, verify that the node (ECU) sends a ReturnToNormalMode positive response message (between $P3_C$ and $P3_{Cmax}$) and that the functionality of the other service has stopped once the ReturnToNormalMode positive response message has been sent.

Procedure 6:

1. Send a ReturnToNormalMode ($20) request to an ECU and verify the positive response.
2. Send a valid functionally addressed $3E request to the node and verify that there is no response.
3. Wait $P3_{Cmax}$ and verify that no unsolicited mode $20 response is sent.
4. Send a valid physically addressed $3E request to the node and verify the positive response.
5. Wait $P3_{Cmax}$ and verify that no unsolicited mode $20 response is sent.

8.15.8 Tester Implications. The TesterPresent ($3E) request message shall be sent at least once every $P3_C$ ms. It is recommended that the tester sends a TesterPresent request message at an interval of less than ½ of $P3_C$. It is the tester's responsibility to ensure that this message is received by the device under test. The tester should take into account bus traffic and message priority in scheduling this message.

8.16 ReportProgrammedState ($A2) Service. The reportProgrammedState is used by the tester to determine:

- Which nodes on the link are programmable.
- The current programmed state of each programmable node.

8.16.1 Service Description. Upon boot up, a programmable node checks for the presence of valid operational software and calibrations, and for any possible memory faults. If any portion of the operational software or calibration data (which is programmed via the Service Programming System) is missing or invalid, then the node is considered to be an SPS_TYPE_B (if permanent diagnostic CAN Identifiers are stored in the ECU) or SPS_TYPE_C (if permanent diagnostic CAN Identifiers are not stored within the ECU) ECU. SPS_TYPE_B and SPS_TYPE_C ECUs shall disable normal communication messages.

SPS_TYPE_A (fully programmed SPS ECU) and SPS_TYPE_B ECUs can respond to all of the diagnostic requests for the services that it supports. SPS_TYPE_C ECUs shall only be able to process diagnostic requests which are sent using the AllNode request CANId with the AllNode extended address, and **shall not respond** to any diagnostic request until diagnostic responses are enabled.

Note: Reference the Diagnostic CANId section of this specification for more information on the AllNode request CANId and the AllNode extended address.

An SPS_TYPE_C ECU shall enable diagnostic responses after receiving a request for this diagnostic service while the DisableNormalCommunication ($28) service is active. The SPS_TYPE_C ECU shall respond to the request for this service using the SPS_PrimeRsp CANId. The SPS_PrimeRsp CANId is a special case diagnostic response CANId in the format of 3xx, where the xx is the diagnostic address of the ECU.

Note: Refer to Appendix D of this specification for more information on Diagnostic Addresses.

Once diagnostic responses are enabled, the SPS_TYPE_C ECU shall continue to accept diagnostic requests using the AllNode CANId with the AllNode extended address, or the SPS_PrimeReq CANId. The SPS_PrimeReq CANId is a special case diagnostic request CANId with the format 0xx, where xx is the diagnostic address of the ECU. All subsequent diagnostic responses to requests using the SPS_PrimeReq CANId or the AllNode request CANId with AllNode extended address shall be transmitted using the SPS_PrimeRsp CANId. The SPS_PrimeReq and SPS_PrimeRsp CAN Identifiers shall be available for use until a ReturnToNormalMode ($20) request is received or a $P3_C$ (TesterPresent) timeout occurs. If a $20 request is received or a $P3_C$ timeout occurs before the ECU is completely programmed, an SPS_TYPE_C ECU shall once again disable diagnostic responses and disable the SPS_PrimeReq and SPS_PrimeRsp CAN Identifiers.

Note: An SPS_TYPE_C ECU shall not send a mode $20 response if it receives a mode $20 request message or a when a TesterPresent timeout occurs during the phase when the SPS_PrimeReq and SPS_PrimeRsp CAN identifiers are enabled. Reference the pseudo code of service $20 for more details.

SPS_TYPE_A and SPS_TYPE_B ECUs shall use their permanent diagnostic CAN Identifiers for receiving and responding to requests for this diagnostic service.

If a programmable node receives a $A2 request, it will report to the tester its current state of programming as follows:

- That a memory error has been detected. The memory error could indicate a problem with RAM or permanent memory (e.g., EEPROM or Flash, potentially including the boot memory).
- That it is has only boot software present (without operational software and without calibration data).
- That it has boot software and operational software but without calibration data.
- That it has boot software, operational software, and calibration data present but the calibration data is a defaulted and programming needs to be performed.
- That it is fully programmed with both operational software and calibration data.

Note: A defaulted calibration set may be used if vehicle assembly plant operations require some module functionality at a point in the build process prior to the point where the ECU would be programmed. Default calibrations may also be present in the ECU when delivered to service. For example an engine controller may have a no start calibration when delivered to service.

8.16.2 Request Message Definition (Table 159).

Table 159: ReportProgrammedState Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReportProgrammedState Request Service Id	M	A2	SIDRQ

8.16.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. There are no sub-function parameters in a request for this service.

8.16.2.2 Request Message Data Parameter Definition. There are no data parameters in a request for this service.

8.16.3 Positive Response Message Definition (Table 160).

Table 160: ReportProgrammedState Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReportProgrammedState Positive Response Service Id	M	E2	SIDPR
#2	programmedState = [fully programmed no op s/w or cal data op s/w present, cal data missing s/w present, default or no start cal present General Memory Fault RAM Memory Fault NVRAM Memory Fault Boot Memory Failure Flash Memory Failure EEPROM Memory Failure]	 M M M C U U U U U U	 00 01 02 03 50 51 52 53 54 55	PS_ FP NSC NC SDC GMF RMF NVRMF BMF FMF EEMF

8.16.3.1 Positive Response Message Data Parameter Definition (Table 161).

Table 161: Response Message Data Parameter Definition

Definition
programmedState This data parameter is used to identify for the off board tool which information needs to be programmed into an node as well as provide an indication of whether or not any memory faults have been detected during the boot up process. A node may choose which memory faults are appropriate to implement. It may also be possible that a node may choose to not support the memory faults in this service but provide fault information by setting a DTC which can be read with the $A9 service.

8.16.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 162.

Table 162: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	SubFunctionNotSupported-InvalidFormat Used when the request message contains more than the SID byte.	M	SFNS-IF
78	RequestCorrectlyReceived-ResponsePending This return code is valid if a tester requests this service and the programmedState calculation has not yet completed.	C	RCR-RP

8.16.5 Message Flow Example ReportProgrammedState.

8.16.5.1 ReportProgrammedState Positive Response Example.

In the following example (Table 163), it is assumed:

- There are two SPS programmable nodes on the link.
- The $28 service has already been activated.
- Node 1 is a programmable node that is fully programmed (SPS_TYPE_A) with a USDT physical response CANId of $641.
- Node 2 is an SPS_TYPE_C ECU that has no operational software or calibration data programmed. The diagnostic address of this node is $80 (thus the SPS_PrimeRsp CANId shall be $380 and the SPS_PrimeReq CANId shall be $080).
- TesterPresent messages are sent (but not shown in the example).

Table 163: ReportProgrammedState - Response Example

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	CAN Id	#1	#2	#3	#4	#5	#6	#7	#8
T(USDT-SF)	$101	$FE	$01	$A2	---	---	---	---	---
N1(USDT-SF)	$641	$02	$E2	$00	---	---	---	---	---
N2(USDT-SF)	$380	$02	$E2	$01	---	---	---	---	---

8.16.6 Node Interface Function.

8.16.6.1 Node Interface Data Dictionary (Table 164).

Table 164: Node Interface Data Dictionary of ReportProgrammedState Service Pseudo Code

Variable/Meaning	Values
message_data_length **message_address_type** **normal_message_transmission_status** **diagnostic_responses_enabled**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
programmedState This is a variable which is used to indicate whether or not a programmable ECU is fully programmed, or if any memory errors exist.	See Response Message Definitions Section
Service_Id This is a variable used in the pseudo code of the diagnostic application function which is used to represent the diagnostic service which is being requested (first data byte after the network layer information for diagnostic request messages).	$00 thru $FF

8.16.6.2 Node Interface Pseudo Code.

```
Powerup States:
None
The following logic is executed when a device powers up or reboots after a software reset:
Check memory for errors
Check for operational software and calibrations
Store result of memory, software and calibration checks in $programmedState
IF (permanent diagnostic CAN Identifiers are programmed) THEN
    diagnostic_responses_enabled ← YES
ELSE
    diagnostic_responses_enabled ← NO
ENDIF
```

The following pseudo code is correct for service $A2 provided that the portion of the diagnostic application program interface (API) which handles processing diagnostic received messages is functionally equivalent to the diagnostic API pseudo code below:

```
BEGINFUNCTION Serv_A2_Msg_Recvd()
IF (message_data_length != 1) THEN
    IF (diagnostic_responses_enabled = YES) THEN
        Reject the request with a ($7F $A2 $12) for invalid format
    ENDIF
ELSE
    diagnostic_responses_enabled ← YES
    /* in larger ECUs, it may be possible for a tester to request this service
     after an ECU powers up but before the ECU completes the programmedState
     calculation. If this occurs, an ECU can send the $7F $78 response and then
     send the positive response after the calculation is complete */
    IF (calculation for programmedState is not complete)
        send negative response ($7F $A2 $78 ..)
    ENDIF
    IF (memory fault exists)
        send a ($E2 $programmedState)
    ELSE IF (op software and calibrations are missing)
        send ($E2 $01)
    ELSE IF (calibration data is missing)
        send ($E2 $02)
    ELSE IF (calibration data present is default calibration)
        send ($E2 $03)
    ELSE
        send ($E2 $00) /* fully programmed ECU */
ENDIF
ENDFUNCTION
```

The following pseudo code is the portion of the diagnostic API which handles the processing of diagnostic received messages. The pseudo code assumes that a node has implemented all of the diagnostic services documented within this specification.

```
BEGINFUNCTION Diag_API_Process_Recv_Msg()
IF (diagnostic_responses_enabled = NO)
    IF ((normal_message_transmission_status = DISABLED) AND
     ($Service_Id = $A2) THEN
        CALL Serv_A2_Msg_Recvd()
    ENDIF
    IF ($Service_Id = $28) THEN
        CALL Serv_28_Msg_Recvd()
    ELSEIF ($Service_Id = $3E) THEN
        CALL Serv_3E_Msg_Recvd()
    ENDIF
ELSE
    SELECT FIRST
    WHEN ($Service_Id = $04)
        CALL Serv_04_Msg_Recvd()
    WHEN ($Service_Id = $10)
        CALL Serv_10_Msg_Recvd()
    WHEN ($Service_Id = $12)
        CALL Serv_12_Msg_Recvd()
    WHEN ($Service_Id = $1A)
        CALL Serv_1A_Msg_Recvd()
    WHEN ($Service_Id = $20)
        CALL Serv_20_Msg_Recvd()
    WHEN ($Service_Id = $22)
        CALL Serv_22_Msg_Recvd()
    WHEN ($Service_Id = $23)
```

```
        CALL Serv_23_Msg_Recvd()
    WHEN ($Service_Id = $27)
        CALL Serv_27_Msg_Recvd()
    WHEN ($Service_Id = $28)
        CALL Serv_28_Msg_Recvd()
    WHEN ($Service_Id = $2C)
        CALL Serv_2C_Msg_Recvd()
    WHEN ($Service_Id = $2D)
        CALL Serv_2D_Msg_Recvd()
    WHEN ($Service_Id = $34)
        CALL Serv_34_Msg_Recvd()
    WHEN ($Service_Id = $36)
        CALL Serv_36_Msg_Recvd()
    WHEN ($Service_Id = $3B)
        CALL Serv_3B_Msg_Recvd()
    WHEN ($Service_Id = $3E)
        CALL Serv_3E_Msg_Recvd()
    WHEN ($Service_Id = $A2)
        CALL Serv_A2_Msg_Recvd()
    WHEN ($Service_Id = $A5)
        CALL Serv_A5_Msg_Recvd()
    WHEN ($Service_Id = $A9)
        CALL Serv_A9_Msg_Recvd()
    WHEN ($Service_Id = $AA)
        CALL Serv_AA_Msg_Recvd()
    WHEN ($Service_Id = $AE)
        CALL Serv_AE_Msg_Recvd()
    OTHERWISE
        IF (message_address_type = PHYSICAL) THEN
        Send Negative Response ($7F $Service_Id $11)
        ENDIF
    ENDSELECT
ENDIF
ENDFUNCTION
```

8.16.7 Node Verification Procedure.

Procedure 1: (For Devices which have stored their permanent diagnostic CAN Identifiers.)

1. Send a ReportProgrammedState request with invalid additional data bytes and verify the proper negative response reject message is sent ($7F $A2 $12).
2. Send a ReportProgrammedState request at a time when the device cannot process this request within $P2_C$ (programmedState calculation not done) and verify the proper negative response (This is only required for devices which support such a situation.) followed by the proper positive response within $P2_C$*.

Procedure 2: (For Devices which have stored their permanent diagnostic CAN Identifiers.)

1. Send a request for this service to the ECU when it is fully programmed and verify the proper response.
2. Send a request for this service to the ECU when it only has boot software present and verify the proper response.

3. Send a request for this service to the ECU when it has boot software and operational software present (no calibration data) and verify the proper response.
4. Obtain a controller with default calibrations programmed in (only applicable if the ECU supports $A2 response $03). Send a request for this service and verify that the ECU sends the $E2 $03 response.
5. Create situations for each of the memory failure response codes that the ECU supports and verify the proper response.

Procedure 3: (For SPS_TYPE_C ECUs when operating out of boot software.)

1. Send a request for each diagnostic service supported in the ECU (except services $28 and $A2) and verify that no response is sent.
2. Send an all nodes request for service $28 and verify no response. Wait for a positive response from all ECUs (that have stored permanent diagnostic CAN Identifiers) and then send a request for $A2 service. Verify that a positive response is sent using the ECU's SPS_PrimeRsp CANId ($3xx).
3. Repeat step 2 for each valid positive response value (programmedState) supported by the ECU when running out of boot, and verify the proper positive responses for each case.
4. Send an all nodes request for service $28 and then repeat step 2 of Procedure 1 (if applicable).
5. Repeat step 2 of this procedure. Send TesterPresent $3E messages at least once every $P3_C$ ms while requesting the other supported diagnostic services with the SPS_PrimeReq CANId ($0xx). Verify that the ECU continues to respond properly using the SPS_PrimeRsp CANId ($3xx). Stop sending TesterPresent ($3E) messages and wait $P3_{Cmax}$ ms. Verify that the ECU no longer responds to diagnostic requests.

8.16.8 Tester Implications. For SPS_TYPE_C ECUs, the tester must first send a mode $28 (DisableNormalCommunication) request to all nodes, AND verify all normal communication has ceased before sending the mode $A2 request. Sending the mode $28 to all nodes first ensures that there will be no conflicts with normal message CAN Identifiers and any of the SPS_PrimeRsp response CAN Identifiers.

The tester must send TesterPresent ($3E) messages at an interval less than $P3_C$ to keep an SPS_TYPE_C ECU actively responding to diagnostic service request messages.

8.17 ProgrammingMode ($A5) Service. This service provides for the following levels of operation:

- Verifies that all criteria are met to enable the programming services for all receiving nodes.
- Enables the high speed mode of operation (83.33 kbps) for all receiving nodes on the Single Wire CAN (SWCAN) bus (if high speed programming was requested by the tool).
- Enables programming services for all receiving nodes.

This service shall only be available if normal communications have already been disabled (via service $28).

8.17.1 Service Description. There are two steps required to enter a programming event. The first step (initiating the programming event) is to verify that it is possible for all nodes to begin a programming event. This step includes indicating to the nodes the baud rate that will be used during the programming event. The second step is to actually enable the programming event. During this step, the tester and the nodes shall initialize their CAN protocol converter hardware to the correct baud rate for the duration of the programming event. Once programming mode has been enabled, the tester must send a TesterPresent (mode $3E) message to all nodes (and any sub-nets as required by the programming event) at least once every $P3_C$ ms to keep this mode active.

There are two possible options when initiating a programming event. The first is to request programming in normal speed (sub-parameter $Level = $01 requestProgrammingMode), and the second is to request programming in high speed mode (sub-parameter $Level = $02 requestProgrammingMode_HighSpeed). High speed programming mode is only available on the low speed link (SWCAN). Programming at normal speed shall be supported on any of the links. A request for this service with the sub-parameter value set to requestProgrammingMode ($01), or requestProgrammingMode_HighSpeed ($02), is used to verify that conditions are correct for a programming event to occur. A node can reject a request to initiate a programming event (sub-parameter = $01 or $02) if specific enabling criteria are not met (e.g., The engine control module should reject a request to initiate a programming event if the engine were running. The ABS module should reject a request to initiate a programming event if it detected that the vehicle was moving). The tester must ensure that a positive response is received from all nodes before actually enabling the programming event. Each node shall respond to a request to initiate a programming event. However, the node shall not actually enter programming mode until it receives a request from the tester to enable programming mode (sub-parameter $Level = $03 enableProgrammingMode). There is no positive response message sent by the nodes in response to an enableProgrammingMode request.

If the message to initiate a programming event was sent requesting high speed programming mode, then the request to enable the programming event shall cause all devices (including the tester) on the SWCAN link to switch to the high speed data rate within **30 ms**. The test device shall wait at least **50 ms** after the request has been transmitted before sending any messages at high speed. This provides time for nodes on the low speed link to switch to high speed and avoid error conditions caused by nodes transmitting at different baud rates simultaneously. Once the 50 ms wait time is expired, the test device will be able to program a node (or nodes) at high speed.

Note: All ECUs (including the tester) shall initialize their protocol converter hardware within 30 ms from the time that the request message has been successfully transmitted on the bus. This means that any node which uses a polling loop to service the protocol device shall ensure that the polling loop is fast enough to process the request message and initialize the protocol converter hardware within 30 ms.

The tester can end a programming event by sending a ReturntoNormalMode ($20) request message, or by allowing a $P3_C$ timeout to occur. At the end of the programming event, each node shall transition out of high speed mode (if high speed mode was active) and all devices shall perform a software reset. The software reset allows a node which was just programmed to begin executing the new software/calibrations downloaded, and is also used to synchronize communications among the nodes on the subnet.

Note: When programming mode is active, there shall be no response to the mode $20 request. This minimizes possible link errors from devices switching out of high speed mode. See the description of the ReturnToNormalMode ($20) service for tester timing requirements when exiting high speed mode.

Although devices are not required to be programmable, all nodes must be tolerant of programming mode. This means a node shall not reject a request to initiate a programming event just because that particular node is not programmable. A negative response shall only be sent if a node detects specific vehicle operating criteria which must prevent the programming event from taking place. The reasons why a specific node refuses or is unable to enter Programming mode shall be documented in the device CTS.

Note: Since current draw during a programming event may drain a battery, nodes should failsoft to a state which draws only necessary current.

8.17.2 Request Message Definition (Table 165).

Table 165: ProgrammingMode Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ProgrammingMode Request Service Id	M	A5	SIDRQ
#2	sub-function = [requestProgrammingMode requestProgrammingMode_HighSpeed enableProgrammingMode]	M	 01 02 03	LEV_ RQPM RQPM_HS RQ_EPM

8.17.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. The sub-parameters for this service are indicated in Table 166.

Table 166: Definition of Sub-function Values

Hex	Description	Cvt
00	**reservedByDocument** This value is reserved by this document.	M
01	**requestProgrammingMode** Request by the tester to verify the capability of the node(s) to enter into a normal speed programming event.	M
02	**requestProgrammingMode_HighSpeed** Request by the tester to verify the capability of the node(s) to enter into a high speed programming event.	M_1
03	**enableProgrammingMode** Request by the tester to have the node(s) enter into a programming event. This can only be sent if preceded by one of the valid requestProgrammingMode messages (above).	M
04 thru FF	**ReservedByDocument** These values are reserved by this document for standardized expansion.	M
Where: **M_1** = This level is mandatory for all ECUs on the SWCAN link and not applicable to the DWCAN links.		

8.17.2.2 Request Message Data Parameter Definition. There are no data parameters in a request for this service.

8.17.3 Positive Response Message Definition (Table 167).

Table 167: ProgrammingMode Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ProgrammingMode Positive Response Service Id	M_2	E5	SIDPR
Where: **M_2** = Mandatory if the request message uses a sub-parameter value of $01 or $02. There is no response to a request message with a sub-parameter value of $03.				

8.17.3.1 Positive Response Message Data Parameter Definition. There are no data parameters in a response to a request for this service.

8.17.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. See Table 168.

Table 168: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This return code shall occur if the sub-parameter is not valid and programming mode is not already active (if programming mode is active at the time of a request, the $22 response is sent. See item 4 in the response code $22 section of this table). This return code shall occur if the length of the requested message is incorrect.	M	SFNS-IF
22	**ConditionsNotCorrectorRequestSequenceError** 1. This return code shall occur if the initiate programming request (sub-parameter = $01 or $02) is not requested prior to the “enableProgrammingMode” (sub-parameter = $03) request. 2. This return code shall occur if the $28 service is not active. 3. This return code shall occur if operating conditions are such that the node must not allow a programming event to begin. Reasons for not allowing a programming event to begin must be documented in a CTS, SSTS, or a supplemental diagnostic specification referenced by the CTS or SSTS. 4. This return code shall occur if programming mode is already active when a request for this service is received.	M	CNCRSE
78	**RequestCorrectlyReceived-ResponsePending** See 7.2 Return Code Definition. This response code is only valid if the sub-parameter in the request message is $01 or $02. It shall NOT be used for the “enableProgrammingMode”.	C	RCR-RP

8.17.5 Message Flow Example - ProgrammingMode.

8.17.5.1 RequestProgrammingMode (normal speed) and enableProgrammingMode Example. In the following example (Table 169), it is assumed:

- The tester has already disabled normal communication with mode $28 (with no negative responses received).
- The tester is sending the programming mode commands on the link where the target node resides.
- The programming session is to be performed in the normal speed of the link.
- There is more than one node on the link.
- TesterPresent messages are being sent (but not shown in the example).

Table 169: requestProgrammingMode Positive Response Example

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$A5	$01	---	---	---	---
N-1(USDT-SF)	$641	$01	$E5	---	---	---	---	---	---

:

N-n(USDT-SF)	$64D	$01	$E5	---	---	---	---	---	---

after all positive responses received:

T(USDT-SF)	$101	$FE	$02	$A5	$03	---	---	---	---

No Response Allowed

8.17.5.2 RequestProgrammingMode_HighSpeed and enableProgrammingMode Example.

In the following example (Table 170), it is assumed:

- The tester has already disabled normal communication with mode $28 (with no negative responses received).
- The tester is sending the programming mode commands on the link where the target node resides.
- The programming event is to be performed in high speed mode (valid only for SWCAN low speed link).
- The target node resides on the low speed link.
- There is more than one node on the link.
- TesterPresent messages are being sent (but not shown in the example).

Table 170: requestProgrammingMode Positive Response Example

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$A5	$02	---	---	---	---
N-1(USDT-SF)	$641	$01	$E5		---	---	---	---	---

:

N-n(USDT-SF)	$64D	$01	$E5	---	---	---	---	---	---

after all positive responses received:

T(USDT-SF)	$101	$FE	$02	$A5	$03	---	---	---	---

No Response Allowed.

8.17.6 Node Interface Function.

8.17.6.1 Node Interface Data Dictionary (Table 171).

Table 171: Node Interface Data Dictionary of ProgrammingMode Service Pseudo Code

Variable/Meaning	Values
message_data_length **programming_mode_active** **normal_message_transmission_status** **high_speed_mode_active** **programming_mode_entry_OK** **high_speed_mode_entry_OK**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables

8.17.6.2 Node Interface Pseudo Code.

Powerup States:

/* see global data dictionary for variable states at powerup. */

Each time a ProgrammingMode ($A5) message is received, the following logic is executed:

```
BEGINFUNCTION Serv_A5_Msg_Recvd()
/* The TesterPresent_Timer_State is not checked in the pseudo code below because it must already be ACTIVE in order to have normal communications disabled (which is checked for in both level $01 and $02. Once normal communications have been disabled, the only methods to re-enable normal comm is to send a $20 request or let a P3C timeout occur. In either case, the flags programming_mode_entry_ok and high_speed_mode_entry_ok get reset. Therefore it is also not possible to have normal comm re-enabled between the verification request ($01 or $02) and the request to enable the programming event ($03) so no check is necessary for this in level $03. */
IF ((message_data_length != 2) THEN
    send Negative Response ($7F $A5 $12) /* InvalidFormat */
ELSE IF (programming_mode_active = YES) THEN
    send Negative Response ($7F $A5 $22) /* ConditionsNotCorrect */
ELSE
    SELECT FIRST
    WHEN ($Level = LEV_RQPM) /* ($01) Request Programming in low speed mode */
        /* check if ok to enter programming mode and that $28 service is active */
        /* $28 service disables setting DTCs and inhibits normal communication */
        IF ( (device cannot enter programming mode) OR
        (normal_message_transmission_status = ENABLED)) THEN
            send Negative Response ($7F $A5 $22) /* Conditions Not Correct */
        ELSE
            IF (device temporarily cannot enter programming mode)
                send Negative Response ($7F $A5 $78) /* RequestCorrectlyReceived-ResponsePending */
            ENDIF
            programming_mode_entry_OK ← YES
            high_speed_mode_entry_OK ← NO
            send Positive Response ($E5)
        ENDIF
    WHEN ($Level = LEV_RQPM_HS) /* ($02) Request Programming in high speed mode */
```

```
        /* This level shall be supported by all ECUs on the low speed link */
        /* This level shall not be supported by ECUs on the two wire links and
         shall result in a negative response $7F $A5 $12 if sent */
        /*check if high speed programming mode is ok and $28 service is active*/
        /*$28 service disables setting DTCs and inhibits normal communication*/
        IF ( (device cannot enter high speed programming mode)OR
        (normal_message_transmission_status = ENABLED)) THEN
            send Negative Response ($7F $A5 $22) /* Conditions Not Correct */
        ELSE
            IF (device temporarily cannot enter programming mode)
                send Negative Response ($7F $A5 $78) /* RequestCorrectlyReceived-ResponsePending */
            ENDIF
            programming_mode_entry_OK ← YES
            high_speed_mode_entry_OK ← YES
            send Positive Response ($E5)
        ENDIF
    WHEN ($Level = LEV_EPM) /* ($03) Enable Programming */
        /* verify that a previous request was sent with $Level = LEV_RQPM or */
        /* $Level = LEV_RQPM_HS */
        IF (programming_mode_entry_OK = NO) THEN
            send Negative Response ($7F $A5 $22) /* ConditionsNotCorrect */
        ELSE
            IF (high_speed_mode_entry_OK = YES) THEN
                CALL hnd_Invoke_High_Spd_Mode() /* handler function */
                high_speed_mode_active ← YES
            ENDIF
            programming_mode_active ← YES
        ENDIF
    OTHERWISE
        send Negative Response ($7F $A5 $12) /* Invalid Format */
        IF ((programming_mode_entry_OK = YES) AND
        (programming_mode_active = NO)) THEN
            programming_mode_entry_OK ← NO
            high_speed_mode_entry_OK ← NO
        ENDIF
    ENDSELECT
ENDIF
ENDFUNCTION
```

8.17.7 Node Verification Procedure.

8.17.7.1 General Verification Procedures (any node, any link).

Procedure 1: (Perform this procedure when conditions would allow a programming event).

1. Disable normal communication, send a requestProgrammingMode message ($A5 $01). Verify the positive response ($E5) message.
2. Send an enableProgrammingMode request ($A5 $03). Verify no response is sent.
3. Wait at least 100ms (but less than $P3_C$ ms), send a request for a supported mode (e.g., Mode $34 RequestDownload). Verify the proper response.

4. Keep the device in enableProgrammingMode for at least 2 minutes by sending TesterPresent (mode $3E) messages at an interval less than $P3_C$ ms. Verify that no normal communication messages are transmitted.
5. Verify that the device communicates in Programming Mode by periodically sending valid request messages.
6. If the device is programmable, verify that the device is capable of being programmed using a released utility file and the correct interpreter software.
7. Repeat the first 4 steps of this procedure and then send a ReturnToNormalMode ($20) message. Verify that the ECU performs a software reset.
8. Repeat the above procedure except end the programming event by allowing a $P3_C$ (TesterPresent) timeout to occur (instead of sending a mode $20 request). Verify that the ECU performs a software reset.

Procedure 2:

1. Disable normal communication then send a requestProgrammingMode message ($A5 $01) with extra data bytes. Verify the negative response ($7F $A5 $12) message.
2. Disable normal communication then send a message with no sub-function parameter byte after the service Id. Verify the negative response ($7F $A5 $12) message.
3. Disable normal communication then send a message with an invalid sub-function parameter byte after the service Id. Verify the negative response ($7F $A5 $12) message.
4. Send requestProgrammingMode message ($A5 $01) without disabling normal communication. Verify the negative response ($7F $A5 $22) message.
5. If applicable, create conditions under which the node should not allow a programming event to take place. Then send a requestProgrammingMode message ($A5 $01). Verify the negative response ($7F $A5 $22) message.
6. Send a request to enableProgrammingMode ($03) without having previously sent a requestProgrammingMode ($01) request. Verify the negative response ($7F $A5 $22) message.
7. Disable normal communication, send a requestProgrammingMode message ($A5 $01). Verify the positive response ($E5) message. Send a ReturnToNormalMode ($20) request message and verify the positive response. Then send a request to enableProgrammingMode ($03) and verify the negative response ($7F $A5 $22) message.
8. Disable normal communication, send a requestProgrammingMode message ($A5 $01). Verify the positive response ($E5) message. Allow a $P3_C$ timeout to occur and then send a request to enableProgrammingMode ($03) and verify the negative response ($7F $A5 $22) message.
9. Repeat steps 1 and 2 of Procedure 1 above. Then send a request for this service with a supported sub-parameter. Verify the proper negative response (7F $A5 $22) and that there is no impact to the current programming event. Repeat this procedure for each supported sub-parameter of this service.
10. Repeat steps 1 and 2 of Procedure 1 above. Then send a request for this service with an unsupported sub-parameter. Verify the proper negative response (7F $A5 $22) and that there is no impact to the current programming event.

Procedure 3: (If negative response code $78 is supported by the ECU.)

1. Disable normal communication and verify the positive response. Send TesterPresent requests as necessary to keep normal communications disabled.
2. Create conditions under which the ECU should return the $7F $A5 $78 response to a request for this service. Send a valid request for this service and verify the $7F $A5 $78 response followed by a final response within the $P2_C$* timing window.
3. Repeat steps 1 and 2 of this procedure for each possible reason an ECU would send the negative response with response code $78. Verify this for each applicable supported sub-function parameter.

8.17.7 2 Additional Verification Procedures for Nodes on Mid-Speed or High Speed Busses.

Procedure 1:

1. Disable normal communication, send a requestProgrammingMode_HighSpeed message ($A5 $02), and verify that the ECU sends a negative response ($7F $A5 $12).

8.17.7.3 Additional Verification Procedures for Switch To High Speed Programming Mode. (SWCAN only)

Procedure 1: (Perform this procedure when conditions would allow a programming event.)

1. Disable normal communication, send a requestProgrammingMode_HighSpeed message ($A5 $02). Verify the positive response ($E5) message.
2. Send an enableProgrammingMode request ($A5 $03). Verify no response is sent.
3. Wait at least 30ms (but less than 50 ms), then send a request for a supported mode (e.g., Mode $34 RequestDownload). Verify the proper high speed response.
4. Keep the device in enableProgrammingMode for at least 2 minutes by sending TesterPresent (mode $3E) messages at an interval less than $P3_C$ ms. Verify that no normal communication messages are transmitted.
5. Verify that the device communicates in high speed Programming Mode by periodically sending valid request messages.
6. If the device is programmable, verify that the device is capable of being programmed using a released utility file for high speed programming and the correct interpreter software.
7. Repeat steps 1 thru 4 of this procedure and then send a ReturnToNormalMode ($20) message. Verify that the ECU switches the protocol converter back to low speed operation and then performs a software reset. Wait 1 s and then send a request for a supported diagnostic service at low speed. Verify the proper response.
8. Repeat the above procedure except end the programming event by allowing a $P3_C$ (TesterPresent) timeout to occur (instead of sending a mode $20 request). Verify that the ECU switches the protocol converter back to low speed operation and performs a software reset. Wait 1 s and then send a request for a supported diagnostic service at low speed. Verify the proper response.

Procedure 2:

1. Disable normal communication then send a requestProgrammingMode_HighSpeed message ($A5 $02) with extra data bytes. Verify the negative response ($7F $A5 $12) message.
2. Send a requestProgrammingMode_HighSpeed message ($A5 $02) without disabling normal communication. Verify the negative response ($7F $A5 $22) message.
3. If applicable, create conditions under which the node should not allow a programming event to take place in high speed mode. Then send a requestProgrammingMode_HighSpeed message ($A5 $02). Verify the negative response ($7F $A5 $22) message.
4. Disable normal communication, send a requestProgrammingMode_HighSpeed message ($A5 $02). Verify the positive response ($E5) message. Send a ReturnToNormalMode ($20) request message and verify the positive response. Then send a request to enableProgrammingMode ($03) and verify the negative response ($7F $A5 $22) message.
5. Disable normal communication, send a requestProgrammingMode_HighSpeed message ($A5 $02). Verify the positive response ($E5) message. Allow a $P3_C$ timeout to occur and then send a request to enableProgrammingMode ($03) and verify the negative response ($7F $A5 $22) message.
6. Repeat steps 1 and 2 of Procedure 1 above (for high speed devices). Then send a request for this service with a supported sub-parameter. Verify the proper negative response (7F $A5 $22) and that there is no impact to the current programming event. Repeat this procedure for each supported sub-parameter of this service. In addition, verify that the device remains in high speed operation.

Procedure 3:

1. Disable normal communication, then send a requestProgrammingMode_HighSpeed message ($A5 $02). After receiving responses from all ECUs, send a requestProgrammingMode message ($A5 $01) when conditions are correct to begin a low speed programming event. Verify positive responses from all ECUs to the requestProgrammingMode request. Next send an enableProgrammingMode ($A5 $03) request. Verify no responses are sent to the enableProgrammingMode request and then verify that the ECU stayed in low speed operation by requesting an additional diagnostic service.

8.17.8 Tester implications. The tester should only request this service with the ALL NODES request CANId ($101) and the ALL NODES extended address ($FE).

The tester should not enable a programming event if it has received a negative response (from any node) to the initiate programming request is sent.

After enabling a programming event (sending a requestProgrammingMode_HighSpeed request followed by an enableProgrammingMode message), the tester must not transmit any messages for 50 ms. The test device must also initialize its protocol converter hardware to high speed operation within 30 ms from the time that the enableProgrammingMode request message was successfully transmitted on the data link.

TesterPresent (Mode $3E) messages must be sent to all nodes at least once every $P3_C$ ms to keep the programming mode active, otherwise the nodes shall return to normal operation (same as receiving a ReturntoNormalMode $20 request or a $P3_C$ timeout).

The tester must be capable of returning the bus to normal speed mode by issuing a mode $20 ReturntoNormalMode message. After sending a mode $20 request (when the link is in high speed mode), the tester should wait 1 s before transmitting any messages to allow devices to reset and to prevent possible bus error or bus off conditions from occurring.

8.18 ReadDiagnosticInformation ($A9) Service. This service allows a tester to read the status of node-resident Diagnostic Trouble Code (DTC) information from any controller, or group of controllers within a vehicle. This service allows the tester to do the following:

1. Retrieve the status of a specific DTC and FaultType combination.
2. Retrieve the list of DTCs that match a tester defined DTC status mask.
3. Enable a node resident algorithm which periodically calculates the number of DTCs that match a tester defined DTC status mask. The ECU shall send a response message each time the calculation yields a different result than the one calculated the previous time.

8.18.1 Service Description. This service uses a sub-parameter to determine which type of DTC information is being requested by the tester. The message number in the UUDT response message shall be the same as the sub-parameter byte in the request message. UUDT message numbers ranging from $80 thru $8F have been reserved for UUDT diagnostic responses for this service and shall not be used for UUDT responses to the $AA service. ECU timing requirements for transmitting the UUDT responses are defined in the description sections below.

Note: An ECU will not transmit a USDT-SF message to positively acknowledge a tester's service $A9 request. However, if conditions necessitate, a $7F negative response message (USDT-SF) may be sent in response to the tester's request (see supported negative responses within this service description for more details).

It shall be possible for a tester to send a request to retrieve the status of a single DTC and Failure type combination, or send a request to determine which DTCs match a tester defined status mask, while the node resident send on change DTC count algorithm is running. However, to ease the burden on an ECU, a node may temporarily suspend the DTC count algorithm while processing a request of the other types.

8.18.1.1 Retrieving the Status of a DTC and Failure Type Combination. A tester can retrieve the status of a specific DTC and DTCFailureTypeByte (symptom) combination by sending a request for this service with the sub-parameter set to readStatusOfDTCByDTCNumber ($80). The response to this request is a UUDT message (message number = $80) containing the two byte DTC Number, the one byte DTCFailureTypeByte, and the one byte DTCStatusByte. The definitions of the DTCFailureTypeByte and the DTCStatusByte can be found in Appendix E.

If the ECU diagnostic application determines that the tester request is NOT acceptable (e.g., incorrect format), the ECU shall return an appropriate USDT-SF negative response within the $P2_C$ time window followed by NO ($80) UUDT message.

8.18.1.2 Retrieving the List Of DTCs That Match A Tester Defined Status Mask. The tester can retrieve a list of DTC numbers and DTCFailureTypes which satisfy a tester defined status mask by sending a request with the sub-parameter byte set to readStatusOfDTCByStatusMask ($81). Nodes shall be able to mask on more than one bit of the status byte field at a time via a logical **OR-ing** operation, making it possible for the tester to request a node to transmit all DTCs that are **current** OR **history** OR , etc., with a single request. If a tester specifies a status mask that contains bits that the node does not support, then the node shall process the DTC information using only the bits that it does support. In response to the tester's request, the ECU shall return a UUDT message to the tester for each DTC and DTCFailureTypeByte combination that satisfies the masking criteria specified by the tester (refer to diagram below for response timing). The UUDT response message(s) shall contain the 2-byte DTC number, the one byte DTCFailureTypeByte, and the one byte DTCStatusByte. Once all DTCs satisfying the tester-specified masking criteria have been transmitted, the ECU shall transmit a special endOfDTCReport UUDT message to flag the end of the transmission. Refer to the corresponding response message definition table below for formatting details regarding the endOfDTCReport message. If no DTC/DTCFailureTypeByte combinations match the masking criteria, then the ECU shall only transmit the endOfDTCReport message in response to the tester's request.

The ECU diagnostic application shall transmit the first UUDT response (or USDT-SF negative response) within $P2_C$ of receiving the tester request. Once the first UUDT response has been sent, each subsequent UUDT response message(s) shall be transmitted within $P2_C$ from the previously transmitted ($81) UUDT response message.

Note: The $P2_C$ timing is the maximum interval that an ECU should send the UUDT responses. It is acceptable to send them at a rate faster than $P2_C$.

If the ECU diagnostic application determines that the tester request message is acceptable, but cannot transmit the first UUDT response message within the $P2_C$ time window, it shall transmit a $7F $A9 $78 RequestCorrectlyReceived-ResponsePending to the tester to invoke enhanced timing $P2_C$*. Once the ECU is able to transmit the first ($81) UUDT response message (within the P2$_C$* time window), each subsequent UUDT response message(s) shall be transmitted within P2$_C$.

If the ECU diagnostic application determines that the tester request is NOT acceptable (i.e., formatted incorrectly), the ECU shall return an appropriate USDT-SF negative response within the P2$_C$ time window followed by NO ($81) UUDT message(s).

See Figure 33 for a graphical representation of the ECU UUDT timing requirements for this service.

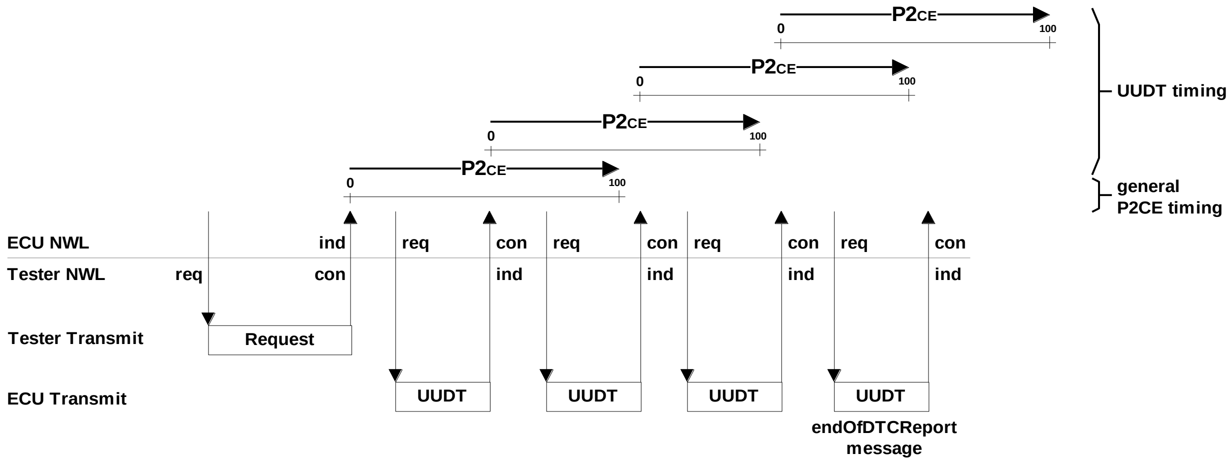


Figure 33: ECU UUDT Timing Parameter Definition for service $A9, $Level $81

8.18.1.3 Enabling the Node Resident DTC Count Algorithm. The tester can enable a node-resident DTC count algorithm by requesting the $A9 service with the sub-parameter set to sendOnChangeDTCCount ($82). If the requested mask value is not $00, the node shall enable a background algorithm that scans through the list of supported DTCs to count the total number of DTCs that match the masking criteria. As soon as the computation is complete, the node shall return a UUDT message back to the tester containing the 2-byte count value. Thereafter, the node resident DTC count algorithm shall count the number of DTCs satisfying the tester defined mask at a periodic rate of approximately 1 s (exact value to be documented in the component technical specification). If the count is different from that which was calculated on the previous execution, the node shall transmit a single UUDT message back to the tester with the updated 2-byte DTC count. The latest count shall then be stored as a reference for the next calculation. Masking shall consist of a logical **OR-ing** operation and shall only be performed on the status byte associated with each DTC/FailureType combination (unsupported status bits are ignored).

Note: If the algorithm which calculates the number of DTCs cannot complete within the P2$_C$ timing window, the ECU shall respond with a $7F $A9 $78 RequestCorrectlyReceived-ResponsePending negative response to institute P2$_C$* timing for the pending UUDT positive response message. The ECU shall always send at least one positive ($82) UUDT response message to the tester's request. Subsequent ($82) UUDT responses will only be returned if and when the background algorithm results in a different count.

A node shall support only one sendOnChangeDTCCount transmission arrangement at a time. Redefinition of the sendOnChangeDTCCount status mask shall occur by sending another sendOnChangeDTCCount request message with a different status mask. A UUDT message with a DTC count of $00 00 shall be returned in response to a request in which the status mask = $00, and the receiving ECU shall halt all background sendOnChangeDTCCount computations.

Once a sendOnChangeDTCCount request message has been sent, the tester must periodically transmit TesterPresent ($3E) messages to keep this level of the service active within the node. If a P3$_C$ (TesterPresent) timeout occurs, the node shall deactivate sendOnChangeDTCCount reporting. The node shall also deactivate this level if the tester transmits a ReturnToNormalMode ($20) service request, or the node is powered down. In each of the three cases described, the node is NOT required to transmit any $A9 specific response message to inform the tester that the send-on-change algorithm has been deactivated.

If a code clear request is received while this level is active, the send-on-change DTC count checking shall remain active. To lessen the burden on the ECU, the send-on-change DTC count algorithm may be temporarily disabled while a DTC clear request is being processed. The algorithm shall be re-enabled after the DTC clear event has completed. The DTC clear does not modify the count value calculated on the previous execution of the algorithm so there is a good chance that a response message will be sent with the new count after the DTC clear has completed.

8.18.2 Request Message Definition.

8.18.2.1 Request Message Definition - $Level $80 (readStatusOfDTCByDTCNumber) (Table 172).

Table 172: ReadDiagnosticInformation(ReadStatusOfDTCByDTCNumber) Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDiagnosticInformation Request Service Id	M	A9	SIDRQ
 #2	sub-function = [readStatusOfDTCByDTCNumber]	M	 80	LEV_ RSDTCBN
 #3 #4 #5	DTCMaskRecord = [DTCHighByte DTCLowByte DTCFailureTypeByte]	M	 xx xx xx	DTCMREC_ DTCH DTCL DTCFT

8.18.2.2 Request Message Definition - $Level $81 (readStatusOfDTCByStatusMask) (Table 173).

Table 173: ReadDiagnosticInformation(readStatusOfDTCByStatusMask) Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDiagnosticInformation Request Service Id	M	A9	SIDRQ
 #2	sub-function = [readStatusOfDTCByStatusMask]	M	 81	LEV_ RSDTCBS
 #3	DTCMaskRecord = [DTCStatusMask]	M	 xx	DTCMREC_ DTCSM

8.18.2.3 Request Message Definition - $Level $82 (sendOnChangeDTCCount) (Table 174).

Table 174: ReadDiagnosticInformation(sendOnChangeDTCCount) Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDiagnosticInformation Request Service Id	M	A9	SIDRQ
 #2	sub-function = [sendOnChangeDTCCount]	M	 82	LEV_ SOCDTCC
 #3	DTCMaskRecord = [DTCStatusMask]	M	 xx	DTCMREC_ DTCSM

8.18.2.4 Request Message Sub-Function Parameter $Level (LEV_) Definition. The following definitions apply to the sub-function levels of operation for service $A9 ReadDiagnosticInformation. All other sub-function values are reserved for future definition within this specification. Future enhancements to this service, which require UUDT responses, shall have sub-function parameter levels within the range from $80 thru $8F as this range of UUDT responses are reserved for this service. See Table 175.

Table 175: Definition of Sub-function Values

Hex	Definition	Cvt
80	**readStatusOfDTCByDTCNumber** Allows a tester to read DTC status information from a node or group of nodes for a tester defined DTC number and FailureType combination.	C
81	**ReadStatusOfDTCByStatusMask** Allows a tester to read DTC status information from a single node or group of nodes for a given status mask.	M
82	**SendOnChangeDTCCount** Allows a tester to retrieve a message which indicates the number of DTCs that match a tester defined status mask. The ECU shall calculate the count information in a background loop and transmit an updated count value each time the computation results in a different value than the last computation.	U
83 thru 8F	**ReservedByDocument** This value is reserved by this document for future definition.	M

8.18.2.5 Request Message Data Parameter Definition. Table 176 specifies the data parameter definitions for this service.

Table 176: Request Data Parameter Definition

Definition
DTCHighByte, DTCLowByte, DTCFailureTypeByte DTCHighByte, DTCLowByte and DTCFailureTypeByte together represent a unique identification number for a specific diagnostic trouble code supported by a device.
DTCStatusMask The DTCStatusMask contains eight DTC status bits. The definitions for each of the eight bits can be found in Appendix E. This byte is used in the request message to allow a tester to request DTC information for the DTCs whose status matches the DTCStatusMask. A DTCs status matches the DTCStatusMask if any one of the DTCs actual status bits is set to 1 and the corresponding status bit in the DTCStatusMask is also set to 1.

8.18.3 Positive Response Message Definition. Positive response(s) to service $A9 requests are of type UUDT. Depending on the sub-parameter in the service request, the receiving node(s) may transmit multiple UUDT response messages containing DTC information. The tester can identify individual UUDT messages by matching the sub-function parameter of the request message and the UUDT message number in the response message. The number of data bytes after the message number depends on the sub-function parameter in the request message.

For sub-function level $81, a UUDT endOfDTCReport message shall be sent after all ($81) UUDT messages containing DTC information have been returned to the tester.

The tables below define the response message formats of each sub-function parameter.

8.18.3.1 Positive Response Message Definition - $Level $80 (readStatusOfDTCByDTCNumber) (Table 177).

Table 177: ReadDiagnosticInformation(readStatusOfDTCByDTCNumber) UUDT Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [readStatusOfDTCByDTCNumber]	M	80	RSDTCBN
	DTCFailureTypeStatusRecord= [	M		DTCFTSR_
#2	DTCHighByte		xx	DTCH
#3	DTCLowByte		xx	DTCL
#4	DTCFailureTypeByte		xx	DTCFT
#5	DTCStatusByte]		xx	DTCST

Note: There are no subsequent UUDT responses associated with $Level $80 of mode $A9.

8.18.3.2 Positive Response Message Definition - $Level $81 (readStatusOfDTCByStatusMask). The following UUDT response to the $A9 $Level $81 request is sent for each DTC that satisfies the tester defined masking criteria. See Table 178.

Table 178: ReadDiagnosticInformation(readStatusOfDTCByStatusMask) UUDT Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [readStatusOfDTCByStatusMask]	M	81	RSDTCBS
	DTCFailureTypeStatusRecord= [	M		DTCFTSR_
#2	DTCHighByte		xx	DTCH
#3	DTCLowByte		xx	DTCL
#4	DTCFailureTypeByte		xx	DTCFT
#5	DTCStatusByte]		xx	DTCSB

The following UUDT endOfDTCReport message shall be sent after all ($81) UUDT messages containing DTC information have been returned to the tester. If no ($81) UUDT messages containing DTC information are sent (because the ECU determined that no DTCs matched the tester-specified masking criteria), the endOfDTCReport message shall still be sent.

8.18.3.3 Positive Response Message Definition - $Level $81 (readStatusOfDTCByStatusMask - endOfDTCReport) (Table 179).

Table 179: ReadDiagnosticInformation(readStatusOfDTCByStatusMask) UUDT endOfDTCReport Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [readStatusOfDTCByStatusMask]	M	81	RSDTCBS
	EndOfDTCReport = [	M		EODTCR_
#2	endOfDTCReportHighByte		00	EODTCRH
#3	endOfDTCReportMiddleByte		00	EODTCRM
#4	endOfDTCReportLowByte		00	EODTCRL
#5	DTCStatusAvailabilityMask]		xx	DTCSAM

8.18.3.4 Positive Response Msg. Definition - $Level $82 (sendOnChangeDTCCount). The following UUDT message shall be sent in response to the tester’s $A9 $Level $82 request. It shall also be sent whenever the count of the number of DTCs satisfying the tester defined mask changes during background execution of the send-on-change DTC count algorithm. See Table 180.

Table 180: ReadDiagnosticInformation(sendOnChangeDTCCount) UUDT Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [sendOnChangeDTCCount]	M	82	SOCDTCC
	DTCCountRecord = [	M		DTCCREC_
#2	DTCCountHighByte		xx	DTCCH
#3	DTCCountLowByte]		xx	DTCCL

8.18.3.5 Positive Response Message Data Parameter Definition. Table 181 specifies the response message data parameter definitions for this service.

Table 181: Request Data Parameter Definition

Definition
MessageNumber This data byte of the UUDT response message contains the sub-function parameter of the request message.
DTCHighByte, DTCLowByte, DTCFailureTypeByte DTCHighByte, DTCLowByte and DTCFailureTypeByte together represent a unique identification number for a specific diagnostic trouble code supported by a device. The DTCHighByte and DTCLowByte represent a circuit or system that is being diagnosed. The DTCFailureTypeByte represents the type of fault in the circuit or system (e.g., sensor open circuit, sensor shorted to ground, algorithm based failure, etc). Definition of the DTCFailureTypeByte can be found in Appendix E of this specification.
DTCStatusByte The status of a particular DTC. (e.g., DTC failed since power up, passed since power up, etc.) The definition of the bits contained in the DTCStatusByte can be found in Appendix E of this specification.
DTCStatusAvailabilityMask A byte whose bits are defined the same as DTCStatusByte and represents the status bits that are supported by the device.
DTCCountHighByte, DTCCountLowByte A count of the number of DTCs that match the DTCStatusMask included in a sendOnChangeDTCCount request message.

8.18.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 182.

Table 182: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This code is returned if the request message length is incorrect. This code is also returned if the requested sub-function is not locally supported.	M	SFNS-IF
31	**RequestOutOfRange** This response code is returned if the DTC and FaultType combination requested with sub-function $80 is not supported by the ECU.	M_1	ROOR
78	**RequestCorrectlyReceived-ResponsePending** This code may be returned if DTC information is being written at the time of the request (depending on memory architecture). The ECU cannot process the request message and send the first (or only) UUDT positive response within the $P2_C$ timing requirements.	C	RCR-RP
Where: **M_1** = Mandatory if sub-function $80 is supported.			

8.18.5 Message Flow Examples - ReadDiagnosticInformation. The examples below illustrate the operation of each sub-function parameter of service $A9 ReadDiagnosticInformation. The following is assumed for all of the examples:

- CAN Identifiers for the ECM (node #1) are $241 (USDT Req) and $541 (UUDT Resp).
- CAN Identifiers for the TCM (node #2) are $243 (USDT Req) and $543 (UUDT Resp).
- The ECM and TCM are the only ECUs connected to the theoretical GMLAN sub-network over which the following messaging examples are conducted.

8.18.5.1 Example #1 - Read Status of DTC by DTC Number. This example (Table 183) demonstrates how a tester can request a node to report the status of DTC P0700 ($0700) FailureType $02. The request with $Level = $80 is followed by a positive ($80) UUDT response message including the DTCH, DTCL, DTCFT, DTCSB.

Table 183: Example of Physical Request for ReadDiagnosticInformation With Sub-Parameter = $80

T = Frame Sent By Tester, N = Frame Sent By Node, shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$05	$A9	$80	$07	$00	$02	---	---
N1(UUDT)	$541	$80	$07	$00	$02	$63	---	---	---

Table 184 is the same as Table 183 example except DTCFailureTypeByte = $00 and the request is functionally addressed with extended address = $FE (all functional systems). It is assumed for the purpose of this example that both ECUs support P0700 DTC with FailureType $00.

Table 184: Example of Functional Request for ReadDiagnosticInformation with Sub-Parameter = $80

T = Frame Sent By Tester, N = Frame Sent By Node, shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$05	$A9	$80	$07	$00	$00	---
N1(UUDT)	$541	$80	$07	$00	$00	$63	---	---	---
N2(UUDT)	$543	$80	$07	$00	$00	$63	---	---	---

8.18.5.2 Example #2 - Functional Read Status of DTC by Status Mask. In this example (Table 185), the tester requests all ECUs to report all DTCs that have a current or history status (status mask = $12). The request with $Level = $81 is followed by a single UUDT message from each ECU for each DTC that matches the masking criteria. Once all DTC information has been sent, each ECU will transmit an endOfDTCReport message to flag the completion of the transfer to the tester.

- Assume the TCM has no DTCs matching the masking criteria. As a result, only the endOfDTCReport message is sent.
- Assume the ECM has two DTCs matching the masking criteria: P0100 (DTCFailureTypeByte $00) and P1864 (DTCFailureTypeByte $00).

Table 185: Example of Functional Request For ReadDiagnosticInformation with Sub-Parameter = $81

T = Frame Sent By Tester, N = Frame Sent By Node, shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$03	$A9	$81	$12	---	---	---
N1(UUDT)	$541	$81	$01	$00	$00	$39	---	---	---
N2(UUDT)	$543	$81	$00	$00	$00	$FF	---	---	---
N1(UUDT)	$541	$81	$18	$64	$00	$07	---	---	---
N1(UUDT)	$541	$81	$00	$00	$00	$FF	---	---	---

Note: Tester should wait for all ECU(s) to return the endOfDTCReport message before transmitting another $A9 $Level $81 request message.

8.18.5.3 Example #3 - Send on Change Reporting of DTC Information. This messaging example (Table 186) shows how the tester can request DTC count information to be sent using a send-on-change message transmission model. The ECM will return the current number of DTCs that satisfy the masking criteria as soon as the tester's request is received. The ECM will then return fresh count information as soon as the node-resident periodic DTC scanning algorithm detects any change in the number of DTCs matching the masking criteria.

Note: The tester must periodically transmit service $3E TesterPresent messages to keep the send-on-change model active.

As was mentioned above, the masking algorithm consists of a logical OR-ing operation between actual DTC statuses and the tester defined filtering mask.

- This example shows that there were four DTCs present in the ECM that matched the status mask $02 at the time of the tester's request (status mask = $02 corresponds to current DTCs).
- At some time later, the ECM returns an updated DTC count indicating that another DTC has satisfied the $02 filtering criteria, bringing the total number of matches to $00 05. (Aside: An off-board tester could use this message from the node to trigger a message $81 request for DTC information by status mask to determine which DTC(s) actually set).
- Note that the count is not monotonically increasing (i.e., If it is determined by a node that fewer DTCs currently match the masking criteria than that which was calculated previously, then the DTC count will decrement). This can be seen below as the count drops from $00 05 back down to $00 04.
- This example also shows how to terminate the send-on-change session by defining a status mask of $00. After positively acknowledging this request with a UUDT message containing a DTC count of $00 00, the ECM will halt all subsequent computation/transmission of send-on-change DTC count information.

Table 186: Example of Physical Request for ReadDiagnosticInformation With Sub-Parameter = $82

T = Frame Sent By Tester, N = Frame Sent By Node, shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$03	$A9	$82	$02	---	---	---	---
N1(UUDT)	$541	$82	$00	$04	---	---	---	---	---

:

N1(UUDT)	$541	$82	$00	$05	---	---	---	---	---

:

N1(UUDT)	$541	$82	$00	$04	---	---	---	---	---

:

T(USDT-SF)	$241	$03	$A9	$82	$00	---	---	---	---
N1(UUDT)	$541	$82	$00	$00	---	---	---	---	---

Table 187 is the same as the example in Table 186, but with negative response message RC_78.

Table 187: Example of Physical Request for ReadDiagnosticInformation With Sub-Parameter = $82 And RC = $78

T = Frame Sent By Tester, N = Frame Sent By Node, shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$03	$A9	$82	$02	---	---	---	---
N1(USDT-SF)	$641	$03	$7F	$A9	$78	---	---	---	---
:									
N1(UUDT)	$541	$82	$00	$04	---	---	---	---	---
:									
N1(UUDT)	$541	$82	$00	$05	---	---	---	---	---
:									
N1(UUDT)	$541	$82	$00	$04	---	---	---	---	---
:									
T(USDT-SF)	$241	$03	$A9	$82	$00	---	---	---	---
:									
N1(UUDT)	$541	$82	$00	$00	---	---	---	---	---

8.18.6 Node Interface Function.

8.18.6.1 Node Interface Data Dictionary (Table 188).

Table 188: Node Interface Data Dictionary of ReadDiagnosticInformation Service Pseudo Code

Variable/Meaning	Values
message_data_length **TesterPresent_Timer_State** **DTC_send_on_change_flag**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
stored_status_mask This variable is used to store the most recently requested status mask in association with a message $82 request.	$00 thru $FF
dtc_found This local flag is used to indicate whether a DTC number/FailureType combination requested via message $80 is locally supported.	TRUE, FALSE

Variable/Meaning	Values
send_on_change_count This variable is used to store the most recently tabulated count of DTCs matching the single tester-defined send-on-change DTC status mask. As soon as service $A9 message $82 is activated, the node will calculate the number of DTCs satisfying the tester-defined masking criteria and store the count in this variable. After acknowledging the tester's request, the node operating in steady state will send updated DTC count information if the current count is different from that which is stored in this variable. This variable will then be updated with the current count of DTCs satisfying the filtering criteria.	$0000 thru $FFFF Signed value.
temp_count This is a temporary storage variable for message $82. It is a 2-byte counter used to store the current count of DTCs that match the tester-defined filtering criteria.	$0000 thru $FFFF Signed value.
temp_count_hi This is a temporary storage variable for message $82. It is the upper byte of temp_count.	$00 thru $FF
temp_count_lo This is a temporary storage variable for message $82. It is the lower byte of temp_count.	$00 thru $FF
total_num_local_dtcs This static variable contains the total number of DTCs any particular node supports. (Note that this number is specific to each node).	$0001 thru $FFFF
M This is a temporary loop index variable that is used in the pseudo code for searching through local DTC lists.	$0000 thru $FFFF
N This is a temporary loop index variable that is used in the pseudo code for searching through local DTC lists.	$0000 thru $FFFF
DTC_Index This is an index variable used by the background loop for sending level $81 responses.	0 to total_num_local_dtcs
Level_81_Active This is a flag which is used to keep track of whether a response to a level $81 request is in progress.	YES/NO
Request_Status_Mask This variable is used to store the requested status mask to a level $81 request and is used by the background function which transmits the responses.	$00 thru $FF
msg_sent This is a local variable used to prevent the ECU from attempting to transmit two messages in response to a level $81 request in the same pass of the background loop.	YES/NO

Variable/Meaning	Values
DTC_Info This is an array of data structures used to house DTC specific information within a node. The following elements comprise this structure: 1. DTCH = most significant byte of diagnostic trouble code. 2. DTCL = least significant byte of diagnostic trouble code. 3. DTCFT = failure type information associated with the DTC. 4. DTCSB = status information associated with the DTC (refer to Table E1: DTC Status Bit Assignments for more details.) To access information in this structure, the following nomenclature is used: DTC_Info[x].element, where x = **m** or **n** as described above in the data dictionary, and **element** is one of the four types listed above.	The values for each element of the DTC_Info structure are described below: 1. \$00 ≤ DTCH ≤ \$FF 2. \$00 ≤ DTCL ≤ \$FF 3. \$00 ≤ DTCFT ≤ \$FF 4. \$00 ≤ DTCSB ≤ \$FF

8.18.6.2 Node Interface Pseudo Code.

Powerup States:

```
DTC_send_on_change_flag ← 0
Level_81_Active ← NO
```

Each time a $A9 message is received, the following logic is executed:

```
BEGINFUNCTION Serv_A9_Msg_Recvd()
/************************************************************************************
* Verify that a valid sub-function is being requested and that the message length is
* valid
************************************************************************************/
IF (($Level not locally supported) OR
(($Level = $LEV_RSDTCBN) AND (message_data_length != $05)) OR /* $Level = $80 */
((message_data_length != $03) AND /* $Level = $82 or $81 and length not 3 */
 (($Level = $LEV_ SOCDTCC) OR ($Level = $LEV_RSDTCBS)))) THEN
    send Negative Response($7F $A9 $12)
ELSE
    IF ((Node cannot service DTC request because read/write memory conflict) OR
    (Tester's request takes longer than P2_C ms to process)) THEN
        send Negative Response($7F $A9 $78)
    ENDIF
    SELECT FIRST
/************************************************************************************
* Handle Message $80 Request "readStatusOfDtcByDtcNumber"
************************************************************************************/
    WHEN ($Level= $LEV_RSDTCBN)
        dtc_found = FALSE
        FOR (m ← 0 TO (total_num_local_dtcs - 1) BY 1)
            IF ((DTC_Info[m].DTCH = $DTCH) AND (DTC_Info[m].DTCL = $DTCL) AND
            (DTC_Info[m].DTCFT = $DTCFT)) THEN
                dtc_found ← TRUE
                /* send UUDT positive response */
                send ($80 $DTC_Info[m].DTCH $DTC_Info[m].DTCL
                 $DTC_Info[m].DTCFT $DTC_Info[m].DTCSB)
                m ← total_num_local_dtcs /* exit FOR loop */
            ENDIF
        ENDFOR
        IF (dtc_found = FALSE)

            send Negative Response($7F $A9 $31)
/*********************************************************************************
* Handle Message $81 Request “readStatusOfDTCByStatusMask”
**********************************************************************************/
    WHEN ($Level= LEV_RSDTCBS)
        DTC_Index ← 0
        Request_Status_Mask ← $DTCSM
        /* Enable Background Loop For Handling UUDT reporting of $A9 $81 DTC information */
        Level_81_Active ← YES
        /* Call background loop to send first UUDT response as acknowledgment */
        call Service_A9_Process_Level_81()
/********************************************************************************
* Handle Message $82 Request “sendOnChangeDTCCount”
* 1. Notice that setting the status mask to $00 terminates the send-on-change
* transmission of DTC count information.
* 2. Also notice that nodes shall always transmit a UUDT acknowledgment message in
* response to a tester’s service $A9 $82 request (even if the tester requests
* the same status mask as the one that is currently being serviced).
**********************************************************************************/
    WHEN ($Level= LEV_SOCDTCC) /* $82 sendOnChangeDTCCount */
        IF ($DTCSM != $00) THEN
            /*set send_on_change_count to -1 ($FFFF), to ensure a UUDT response to the request */
            send_on_change_count ← -1
            TesterPresent_Timer_State ← ACTIVE
            stored_status_mask ← $DTCSM
            /* Enable Background loop for handling UUDT reporting of $A9 $82 DTC data */
            DTC_send_on_change_flag ← 1
            /* Call background loop one time to send the first UUDT acknowledgement */
            call Send_On_Change_DTC_Count()
        ELSE
            /* disable background send on change processing */
            DTC_send_on_change_flag ← 0
            /* Send UUDT positive response with a DTC count of $00 00 */
            send ($82 $00 $00)
        ENDIF
    ENDSELECT
ENDIF
ENDFUNCTION
```

This function handles the sending of UUDT response messages for service $A9 requests with the sub-parameter set to $81. The function shall be executed in a background loop at a rate which satisfies the UUDT timing requirements specified in the Service Description section of this diagnostic service.

```
BEGINFUNCTION Service_A9_Process_Level_81( )
```

```
IF (Level_81_Active = YES) THEN
    msg_sent ← NO
    FOR (n ← DTC_Index TO (total_num_local_dtcs - 1) BY 1)
        IF (($DTC_Info[n].DTCSB & $Request_Status_Mask) != $00) THEN
            /* send $81 UUDT response message containing DTC information */
             send ($81 $DTC_Info[n].DTCH $DTC_Info[n].DTCL $DTC_Info[n].DTCFT
             $DTC_Info[n].DTCSB)
            msg_sent ← YES
            DTC_Index ← n + 1 /* save starting point for next time through FOR loop */
            n ← total_num_local_dtcs /* exit FOR loop */
        ENDIF
    ENDFOR
    IF (n ≥ (total_num_local_dtcs - 1) AND (msg_sent = NO)) THEN
        Level_81_Active ← NO
        /* send $81 UUDT endOfDTCReport response message */
        send ($81 $00 $00 $00 $DTCSAM)
    ENDIF
ENDIF
ENDFUNCTION
```

This function handles the sending of UUDT response messages for service $A9 requests with the sub-parameter set to $82. The function shall be executed in a background loop at a rate which satisfies the UUDT timing requirements specified in the Service Description section of this diagnostic service

```
BEGINFUNCTION Send_On_Change_DTC_Count( )
/***********************************************************************************
* Handle Message $82 Steady State "sendOnChangeDTCCount"
* This background algorithm is halted when any of the following conditions occur:
* 1. Tester transmits service $20 request to return to normal mode
* 2. The node is powered down
* 3. Tester stops sending service $3E tester present requests
* NOTE: this algorithm is intended to be run in parallel to internal node-resident
* diagnostic trouble code testing.
***********************************************************************************/
IF (DTC_send_on_change_flag = 1) THEN
    temp_count ← 0
    FOR (m ← 0 TO (total_num_local_dtcs -1) BY 1)
        IF (($DTC_Info[m].DTCSB & stored_status_mask) != $00) THEN
            temp_count ← temp_count + 1
        ENDIF
    ENDFOR
    IF (temp_count != send_on_change_count) THEN
        send_on_change_count ← temp_count
        temp_count_hi ← ((send_on_change_count & $FF00) >> 8)
        temp_count_lo ← (send_on_change_count & $00FF)
        /* send UUDT response with updated count information */
        send ($82 $temp_count_hi $temp_count_lo)
    ENDIF
ENDIF
ENDFUNCTION
```

8.18.7 Node Verification Procedure.

Procedure 1:

1. Send a request for level $80 (if supported) with less than 3 data bytes after the sub-function parameter in the message. Verify the proper negative response ($7F $A9 $12).
2. Send a request for level $80 (if supported) with more than 3 data bytes after the sub-function parameter in the message. Verify the proper negative response ($7F $A9 $12).
3. Send a request for level $81 with no data bytes after the sub-function parameter. Verify the proper negative response ($7F $A9 $12).
4. Send a request for level $81 with more than 1 data byte after the sub-function parameter. Verify the proper negative response ($7F $A9 $12).
5. Send a request for level $82 (if supported) with no data bytes after the sub-function parameter. Verify the proper negative response ($7F $A9 $12).
6. Send a request for level $82 (if supported) with more than 1 data byte after the sub-function parameter. Verify the proper negative response ($7F $A9 $12).
7. Send a request for each optional sub-function parameter not supported by the ECU ($80 or $82). Verify the proper negative response ($7F $A9 $12).
8. Send a request with a sub-function parameter value which is not supported by this service. Verify the proper negative response ($7F $A9 $12).
9. Send a request for this service (with any valid sub-function parameter) at a time when the ECU cannot send a response within $P2_C$. Verify that the ($7F $A9 $78) response is sent followed by the proper positive response within the time values specified in the Negative Response ($7F) Service Definition section of this specification.

Procedure 2: (Verification Of The readStatusOfDTCByStatusMask $81 Level.)

1. Send a ClearDiagnosticInformation ($04) request. Then send a request for service $A9 with the sub-function parameter equal to $81 and the current bit set in the status mask. Verify that only the $81 UUDT endOfDTCReport positive response is sent, and that no additional UUDT messages follow. For the endOfDTCReport message, verify that the DTC/failure type byte combination is set to $00 00 (i.e., $00 $00 $00), and verify that supported bits in the status availability mask are correct for the ECU.
2. Set a DTC (this can be done by disconnecting a sensor input that is known to cause a single DTC to immediately become current). Then send a request for this service with the sub-function parameter equal to $81 and the current bit set in the status mask. Verify that the DTC information is correctly reported in the UUDT response message, followed by an appropriate endOfDTCReport message.
3. After completing step 2 above, send a request for this service with the sub-function parameter equal to $81. In the request message, define a status mask with the current bit set, and an unsupported status bit also set. Verify that the responses are identical to those from procedure 2.
4. Using instrumentation or other means available to view the status of DTCs, manipulate ECU inputs and outputs to cause DTCs to set each of the status bits supported by the ECU. Send requests for this service with the sub-parameter set to $81 and a status mask of $FF, and verify each time that the correct DTC Numbers, FaultTypes, and Status information are returned in the UUDT response messages. In each case, verify that an appropriate endOfDTCReport message is sent, and that no further UUDT responses follow.

Procedure 3: (Assuming the ECU supports the readStatusOfDTCByDTCNumber $80 Level.)

1. Request the status of a locally supported DTC/FailureType combination: ($A9 $80 $XX $XX $XX). And verify the proper response and status.
2. Repeat step 1 for each locally supported DTC/FailureType combination.
3. Request the status of a DTC/FailureType combination that is not locally supported and verify the proper negative response ($7F $A9 $31).

Procedure 4: (Assuming the ECU supports the sendOnChangeDTCCount $82 Level.)

1. Request a node supporting sub-parameter $82 to report the number of current DTCs (reference Appendix E for definition of DTC status bits) to be sent using send-on-change transmission model ($A9 $82 $02).
2. Verify that the node properly acknowledges the request message with a UUDT response ($82 $DTCCH $DTCCL) containing the correct count of DTCs.
3. Begin periodic transmission of TesterPresent ($3E) messages to keep the send-on-change transmission model active.
4. Select a DTC that is not current, and cause it to transition to a current state (this can be done by disconnecting a sensor input that is known to cause a current code to set immediately).
5. Verify that the node transmits a UUDT message containing an updated count of the number of current DTCs ($82 $DTCCH $DTCCL) within 1.0 s (or whatever execution time is specified in the node's CTS) of the time the code sets.
6. Transmit a request to read the status of all current DTCs by status mask ($A9 $81 $02).
7. Verify that the DTC that was just set is stored.
8. If no other DTCs became current (or discontinued being current) between the time that the previous $Level $82 UUDT response was sent and the time that the $Level $81 UUDT endOfDTCReport response was sent, verify that the number of $Level $81 UUDT response messages received by the tester equals the DTC count reported via the $Level $82 UUDT response.
9. Stop sending TesterPresent messages for $P3_{Cmax}$. Verify that the send on change algorithm has stopped running by causing another DTC to transition to a current status and verifying that no response is sent.
10. Repeat steps 1 through 3 of this procedure. Then send a new request to change the status mask criteria (does not matter whether the resulting background computation will result in a different count value or not). Verify that the ECU sends a $Level $82 UUDT response with an updated count value and verify the time to the response is correct.
11. Repeat steps 1 through 3 of this procedure. Then send a request to change the status mask to $00 (disable the send on change algorithm). Verify that the proper $Level $82 UUDT positive response is sent with a DTC count of $00 00 ($82 $00 $00). Also verify that the algorithm is disabled within the receiving node (by setting another DTC and seeing no send-on-change response).
12. Repeat steps 1 through 3 of this procedure. Then send a ReturnToNormalMode $20 request. Set a DTC and verify that no DTC send on change response is sent.
13. Cause a node to set a history DTC. Then send a mode $A9 $Level $82 request with the history code bit of the status mask set to a logical "1". Verify the $82 UUDT response. Then send a ClearDiagnosticInformation ($04) request. Verify that a $82 UUDT response message is sent with an updated DTC count after the code clear completes (taking into account the execution rate of the background count algorithm).

8.18.8 Tester Implications. The send-on-change status mask and DTC count information are lost upon a $3E time-out, after a $20 service request, or after power is cycled.

The tester shall take into account the execution rate of the background send on change DTC algorithm when determining how long to wait for the UUDT response to a $Level $82 request.

The test device shall use the endOfDTCReport message to determine when all UUDT responses to a $Level $81 request have been sent. For a given node, transmitting another $Level $81 request prior to receiving the endOfDTCReport message may result in undefined operation. Refer to the $A9 $Level $81 Service Description section for more timing details.

8.19 ReadDataByPacketIdentifier ($AA) Service. The purpose of the ReadDataByPacketIdentifier($AA) service is to allow a tester to request data packets that contain diagnostic information (e.g., sensor input or output values) which are packaged in a UUDT diagnostic message format. Refer to paragraph 4.5.1.2 for more information on UUDT diagnostic message format. Each diagnostic data packet includes one byte that contains a Data Packet IDentifier (DPID) number, and one to seven bytes of additional data. The DPID number occupies the message number byte position of the UUDT diagnostic response message and is used by the tester to determine the data contents of the remaining bytes of the message.

This service is intended to be used to retrieve ECU data which is most likely changing during normal operation (e.g., ECU sensor inputs, ECU commanded output states, etc). Static information such as VIN or Part Numbers should be retrieved via the ReadDataByIdentifier ($1A) service.

The DPIDs requested via this service can be sent as a one-time response or scheduled periodically. Each DPID scheduled can be transmitted at one of three predefined periodic rates (slow, medium, or fast). Periodic rates require a TesterPresent ($3E) message to be sent on the bus to keep the Periodic DPID Scheduler (PDS) active (reference $3E service description).

8.19.1 Service Description. The ReadDataByPacketIdentifier service ($AA) request message includes a sub-function parameter and at least one DPID number when the request is for a one time response (sub-function parameter $01) or periodic transmission (sub-function parameters $02 through $04). A single request to stop sending one or multiple periodic DPIDs (sub-function parameter $00) may be sent with or without additional DPID data and the resulting action taken by the ECU shall be different based on whether or not DPID data is attached. It shall be possible to request the transmission of multiple DPIDs with a single request for this service. The controller shall support multiple DPIDs in the request message for all possible data rates implemented (slow, medium, fast, stop periodic, and one time response).

The entire request message shall be validated before any action is taken. If any error in a request message is identified, the whole request shall be rejected and the appropriate negative response shall be transmitted to the tester. A single message validation exception exists for the stopSending sub-function parameter where an invalid DPID in the request message does not result in a negative response as long as the total number of DPIDs in the request is less than the maximum number of items allowed in the periodic scheduler. Refer to the pseudo code and Supported Negative Response Codes section for the required implementation.

A positive response to a request for this service shall always consist of one or multiple UUDT diagnostic messages. The only USDT diagnostic responses allowed with this service are negative responses.

8.19.1.1 Requesting A One Time Response of One or More DPIDs Sub-Function Parameter = $01 (sendOneResponse). The ECU shall send a single UUDT diagnostic response message for each DPID included in a request for this service with a sub-function parameter = $01 (sendOneResponse). The maximum number of DPIDs the ECU is required to support in a sendOneResponse request shall be documented in the ECU’s Component Technical Specification (CTS).

The tester shall wait until all responses to a sendOneResponse DPID request have been received before issuing another sendOneResponse DPID request. The ECU is only required to handle a single sendOneResponse DPID request at a time. If the tester sends another sendOneResponse request before the ECU has finished responding to the first one, the results are undefined (based on the message buffering capabilities of the receiving ECU). After issuing a sendOneResponse request to an ECU, a tester is allowed to send another diagnostic request (except for another sendOneResponse request) to the same ECU after receiving either the first/only UUDT DPID response, or receiving a negative response with a response code value not equal to $78 (RequestCorrectlyReceived-ResponsePending).

The ECU shall support a sendOneResponse DPID request while the periodic scheduler is active (assuming that the ECU supports one or more of the periodic levels of this service and provided that another sendOneResponse request is not in progress).

If multiple DPIDs are included in a valid sendOneResponse DPID request, the ECU shall send the first UUDT DPID response within $P2_{CE}$ (or $P2_{CE}$* if a negative response with response code $78 is sent by the ECU within $P2_{CE}$).

Note: See the application timing requirements section within this specification for further details.

The ECU shall transmit all subsequent DPID responses within $P2_{CE}$ of the previous DPID response transmitted as a result of this request. Any deviations from the sendOneResponse UUDT timing requirements shall be agreed upon by the DRE, Service and Manufacturing serial data engineers, and shall be documented in a CTS, SSTS, or supplemental diagnostic specification referenced by a CTS or SSTS.

8.19.1.2 Requesting Periodic DPID Scheduling - Sub-Function Parameters ($02 sendAtSlowRate, $03 sendAtMediumRate, and $04 sendAtFastRate). If the Node supports any of the levels that allow a tester to schedule periodic DPID transmissions (sub-function parameters $02 through $04), it shall have a Periodic DPID Scheduler (PDS). The maximum number of simultaneously scheduled DPIDs that the Node is required to support shall be documented in the ECU Component Technical Specification (CTS). Each DPID in the PDS shall be capable of being scheduled at one of the allowable periodic rates independent of the rate applied to the other DPIDs in the PDS. The ECU shall be able to support in a single periodic DPID request, as many DPIDs as there are items in the PDS. Refer to parameter PDS_Length in pseudo code.

The default periodic rates (scheduling rates) for the periodic DPID scheduler are **1000 ms** for slow, **200 ms** for medium, and **25 ms** for fast. Periodic rate is defined as the time between any two consecutive messages of the **same DPID** when it is scheduled by this service.

Note: This service does not require an ECU to transmit ALL scheduled DPIDs at the same time. It is allowable to have a minimum time between different DPIDs. The periodic rate is the time between consecutive transmissions of a single given scheduled DPID, NOT the time between transmission of ALL scheduled DPIDs.

The default rates are guidelines and may be modified based on ECU capabilities. The actual data rates associated with slow, medium, and fast, shall be documented in the CTS.

Note: The actual rate at which a **specific** periodic DPID is transmitted can vary from the requested periodic rate based on the number of DPIDs in the PDS, the rate at which the ECU services the PDS, and the amount of bus traffic. Reference the pseudo code (paragraph 8.19.6.2) and the examples (paragraph 8.19.6.3) for specifics on how the number of DPIDs in the PDS and the rate at which the PDS is serviced affects the periodic transmission rate.

Each time a valid $AA request for periodic transmission is received the PDS shall be started (if not already active) and the corresponding DPID(s) shall be put into the PDS or be updated with the requested (new) rate. Multiple copies of the same DPID are not allowed in the PDS (i.e., if the ECU receives a request to schedule a DPID already present in the PDS, only the scheduling rate will be updated). If a $AA request for periodic transmission does not include a DPID already active for periodic transmission, this DPID shall remain unaffected (e.g., if DPIDs 1, 4 and 5 are periodically sent and a new $AA request for periodic transmission only includes DPIDs 1 and 4, the scheduled rate for DPID 5 shall remain unaffected while the scheduling rate for DPIDs 1 and 4 are set to the new requested scheduling rate).

It shall be possible to request additional DPIDs to be scheduled at any allowed rate while the periodic scheduler is active provided that the request does not attempt to place more DPIDs in the PDS then the PDS supports. It shall also be possible to request scheduling DPIDs while a sendOneResponse DPID request is in progress provided that the initial UUDT DPID response to the sendOneResponse DPID request has been received. It shall be possible to redefine the rate of one or several DPID(s) while periodic transmission is active. If dynamic DPIDs are supported by a Node, the Node shall allow redefinition of any/all dynamic DPID(s) (via the $2C service) while periodic transmission is active.

When a $P3_C$ (TesterPresent) time-out occurs, the ECU shall stop periodic transmission of all DPIDs (stop the PDS) and the entire PDS shall be cleared. No UUDT response shall be sent on this action. See TesterPresent ($3E) service regarding actions at tester present time-out.

After issuing a request for this service using one of the supported periodic rates, the tester must wait before issuing any new diagnostic requests until one of the following occur:

Note: The exception to this is the possibility of interleaving a functionally addressed TesterPresent ($3E) request message to keep certain diagnostic services active. See TesterPresent service description and the section on Diagnostic Communication Implementation Rules for more information.

- A UUDT DPID response is received for a DPID which was not previously in the PDS, or
- The tester receives any negative response message, or
- The tester receives at least one previously scheduled UUDT DPID response from the periodic request after $P2_C$.

Note: The tester is allowed to send a new diagnostic request after receiving a negative response that contains a request service Id of $AA and a negative response code $78 (RequestCorrectlyReceived-ResponsePending) because no further USDT response (either positive or negative) shall follow. All positive responses are UUDT messages and no additional USDT negative response shall follow in this case because the ECU is required to verify that the entire request for this service is valid before sending any response. This means that the negative response with response code $78 is only used with this service when it is possible that the ECU may not be able to send any of the requested DPIDs within the P2$_C$ timing window. The ECU is only allowed to transmit a single negative response with response code $78 to a periodic request for this service. Refer to the pseudo code (Process_AA_Msgs function for more details).

Timing of additional UUDT responses is handled in the applications PDS logic.

8.19.1.3 Stopping Scheduled Periodic DPID(s) - Sub-Function Parameter ($00). If a node supports any of the periodic levels of this service, then it shall also support the stopSending ($00) sub-function parameter. The positive response to a stopSending DPID request is a single UUDT diagnostic message with a value of $00 in the DPID/message number position and no additional data bytes.

When a stopSending request is received without any DPIDs after the sub-function parameter, the ECU shall stop the periodic transmission of all DPIDs (stop the PDS) and the entire PDS shall be cleared (cleared = all DPIDs removed from the PDS. Reference the pseudo-code for the required implementation). When a stopSending request is received with one or more DPID(s) after the sub-function parameter, then all DPID(s) in the PDS that match a DPID number in the request shall be stopped. The ECU shall be able to support in a single stopSending DPID request, as many DPIDs as there are items in the PDS. Refer to parameter PDS_Length in pseudo code.

Note: The ECU shall stop only the matching DPIDs. All other scheduled DPIDs shall remain unaffected. A stopped DPID shall not be transmitted after the UUDT positive response with message number $00 has been sent.

8.19.2 Request Message Definition.

8.19.2.1 ReadDataByPacketIdentifier Request Message (Table 189).

Table 189: ReadDataByPacketIdentifier Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	ReadDataByPacketIdentifierRequest Service Id	M	AA	SIDRQ
#2	sub-function parameter = [	M		LEV_
	stopSending		00	SS
	sendOneResponse		01	SOR
	scheduleAtSlowRate		02	SASR
	scheduleAtMediumRate		03	SAMR
	scheduleAtFastRate]		04	SAFR
#3	request_DPIDs[] = [DPID#1	M_1/C_1	xx	DPID1
:	:	:	:	:
#n	DPID#m]	U	xx	DPIDm

Where:

M_1 = Mandatory If value of sub-function parameter = [$01, $02, $03, $04]

C_1 = Included if value of sub-function parameter = $00 AND request is to stop periodic transmission of individual DPID(s) (i.e., DPID(s) included in sub-function parameter=$00 request)

8.19.2.1.1 Sub-Function Parameter $Level (LEV_) Definition. The following sub-function parameter values are defined for service $AA **ReadDataByPacketIdentifier**. The sub-function parameter shall be used in a $AA service request to select the level of functionality. The sub-function parameter is always present in a request message. See Table 190.

Table 190: Definition of Sub-Function Parameter In Request Message

Hex	Description	Cvt
00	**stopSending** Stops periodic transmission of all or certain scheduled DPIDs.	C_2
01	**sendOneResponse** Request for one-shot response message(s) (one UUDT message for each requested DPID).	M
02	**scheduleAtSlowRate** Request to schedule DPID(s) in the request at slow periodic rate.	U
03	**scheduleAtMediumRate** Request to schedule DPID(s) in the request at medium periodic rate.	U
04	**scheduleAtFastRate** Request to schedule DPID(s) in the request at fast periodic rate.	U
05 thru FF	**ReservedByDocument** This value is reserved by this document for future definition.	M
Where: C_2 = Support of this sub-function parameter is mandatory if any of the periodic levels of this service are supported.		

The sub-function parameters supported by a Node are vehicle manufacturer optional and shall be documented in the Component Technical Specification (CTS).

8.19.2.2 Request Message Data Parameter Definition. Tables 191 and 192 specifiy the data parameter definitions and valid ranges for this service.

Table 191: Request Message Data Parameter Definition

Definition
request_DPIDs[] This is an array of the DPID numbers that the tester is requesting. The data values identify to the node which DPID(s) to transmit or stop transmitting (as indicated by the preceding sub-function parameter byte). The number of elements in this array can vary based on the preceding sub-function parameter byte. A further definition of the meaning of the DPID data contained in this array can be found below.

Table 192: Definition and Valid Ranges Of DPID Parameters

Hex	Description	Cvt	Mnemonic
00	**ReservedByDocument** This value is used to indicate that one or more of the scheduled DPIDs are being stopped. No data is ever sent with this DPID number.	M	**RBD**
80 thru 8F	**ReservedByDocument** These values are not allowed because the DPID number occupies the message number position in a UUDT message and the $A9 service also uses the message number position in a UUDT message to provide an echo of the sub-function parameter. The range of message numbers from $80 thru $8F are reserved for DTC responses.	M	RBD
01 thru 7F, 90 thru FE	**DPID** DPIDs can be either static or dynamically definable. Dynamically definable DPIDs (see service $2C for more detail) shall start with DPID number $FE and be numbered sequentially backwards. Static DPIDs (DPIDs whose signal/parameter content is determined at the time of ECU software compilation) shall start with DPID number $01 and be numbered sequentially forwards.	M	DPID
FF	**ReservedByDocument** This value is reserved for future definition.	M	RBD

8.19.3 Positive Response Message Definition. Positive responses to this service are UUDT message(s). The first byte of a UUDT diagnostic message contains a message number. For this service, the DPID# occupies the message number field. The number of data bytes after the message number depends on the sub-function parameter value or the DPID requested. If sub-function parameter ($Level) $00 is requested, then the response message contains only the message number (DPID $00) and no additional data as illustrated by Table 193.

Table 193: ReadDataByPacketIdentifier, $Level = $00 Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [stopSendingPositiveResponse]	M	00	DPID_

If the request message contains any other sub-function parameter ($Level) value, then the positive response message contains the DPID number requested and data bytes associated with that DPID, as illustrated by Table 194.

Table 194: ReadDataByPacketIdentifier, $Level = $01, $02, $03, or $04 Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	MessageNumber = [DPID#]	M	01 thru 7F, or 90 thru FE	DPID_
#2	dataRecord = [Data Byte #1	M_2	xx	DB_1
:	:	:	:	:
#8	Data Byte #7]	U	xx	DB_7

Where:

M2 = This byte shall always be present for static DPIDs. For dynamic DPIDs this byte shall always be present once the DPID contents have been assigned via the $2C service. The number of bytes transmitted in the data record shall be 0 for a given dynamic DPID if that DPID had not previously been defined via the $2C service.

8.19.3.1 Response Message Data Parameter Definition. Table 195 specifies the data parameter definitions for this service.

Table 195: Response Message Data Parameter Definition

Definition
MessageNumber This data byte of a UUDT response from this diagnostic service contains the DPID number which identifies to the tool the meaning of the remaining data bytes in the message.
dataRecord[] These are the bytes containing the actual operating data values for each of the signals (e.g., PIDs) assigned as part of the specific DPID requested.

8.19.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. See Table 196.

Table 196: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** This response code shall be sent if: • Message length of the request is invalid (less than two bytes). • Requested sub-function parameter is invalid (not defined or not supported). • $sub-function parameter = [$01 to $04] and Message length < 3. • The number of DPIDs in a sub-function parameter $01 request exceeds the maximum number of DPIDs supported for one-shot. • The number of DPIDs in a sub-function parameter $00, $02, $03 or $04 request exceed the maximum number of DPIDs that can be scheduled simultaneously in the PDS. Refer to pseudo code.	M	SFNS-IF
31	**RequestOutOfRange** This response code shall be sent if: Any DPID in the request is invalid (reserved or not defined). This check is not required for sub-function parameter $00. Checks for valid DPIDs are performed after the checks for message format and sub-function level.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** This response code shall be sent if: The ECU cannot process the request message and send the first/only UUDT positive response within the $P2_C$ timing requirements.	C	RCR-RP
81	**Scheduler Full** This response code shall be sent if: The PDS would overflow if the request was processed (not enough vacant items in PDS to process request).	C_3	SF
Where: C_3 = Only required if any of sub-function parameter $02, $03 or $04 are implemented			

8.19.5 Message Flow Example ReadDataByPacketIdentifier. For all of the examples of this service below:

- The tester to Node point-to-point request CAN Identifier is $241 **and**
- The Node to tester UUDT response CAN Identifier is $541.

8.19.5.1 ReadDataByPacketIdentifier Scheduled UUDT Positive Responses. In the example below (Table 197):

- The tester sends a ReadDataByPacketIdentifier request with the sub-function parameter = $03 (scheduleAtMediumRate) to schedule DPIDs $10, $23 and $30.
- DPID $10 consists of four bytes data,
- DPID $23 consists of six bytes data **and**
- DPID $30 consists of seven bytes data.

Table 197: ReadDataByPacketIdentifier Request ($Level = $03) - Scheduled UUDT Positive Responses

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$05	$AA	$03	$10	$23	$30	---	---
N(UUDT-SF)	$541	$10	$32	$33	$EF	$44	---	---	---
:									
N(UUDT)	$541	$23	$21	$32	$15	$01	$11	$55	---
:									
N(UUDT)	$541	$30	$33	$33	$11	$45	$98	$A3	$AA

Note: The TesterPresent ($3E) messages Necessary to keep periodic responses active are not shown in the example.

8.19.5.2 ReadDataByPacketIdentifier Stop Periodic Messages Positive Response.

In the example below (Table 198):

- The tester sends a ReadDataByPacketIdentifier request with the sub-function parameter = $00 (StopSending), and no additional data parameters. This causes the Node to stop sending all scheduled DPIDs.
- The Node then confirms the performed action with a UUDT positive response message with the MessageNumber = $00.
- DPIDs active for periodic transmission not shown in example below.

Table 198: ReadDataByPacketIdentifier Request ($Level = $00) - Single UUDT Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$AA	$00	---	---	---	---	---
N(UUDT-SF)	$541	$00	---	---	---	---	---	---	---

Note: The ECU(s) shall first disable the scheduler prior to sending the positive UUDT response message.

8.19.5.3 ReadDataByPacketIdentifier Request Single Shot UUDT Frame(s) Positive Response. In the example below (Table 199):

- The tester sends a ReadDataByPacketIdentifier request with the sub-function parameter = $01 (SendOneResponse), with DPIDs $A1 and $22 as data parameters.
- DPID $A1 contains three bytes of data.
- DPID $22 contains two bytes of data.
- The example below is independent on whether scheduling is active or not at time of request.

Table 199: ReadDataByPacketIdentifier Request ($Level = $01) - UUDT Positive Responses

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$04	$AA	$01	$22	$A1	---	---	---
N(UUDT)	$541	$22	$AD	$12	---	---	---	---	---
N(UUDT)	$541	$A1	$98	$34	$45	---	---	---	---

8.19.5.4 ReadDataByPacketIdentifier Sequence Example.

In the sequence of examples below (Tables 200 thru 204):

- The tester sends a ReadDataByPacketIdentifier request with the sub-function parameter = $03 (scheduleAtMediumRate) to schedule two DPIDs.
- The tester then requests an additional DPID to be scheduled at a fast rate while the previous two DPIDs still are transmitted at medium rate.
- One of the medium rate scheduled DPIDs is stopped by a sub-function parameter $00 with the DPID in the request.
- The tester then sends a ReadDataByPacketIdentifier request with the sub-function parameter = $01 (sendOneResponse) to read one DPID (one DPID is still scheduled at fast rate and another at medium rate).
- Finally the tester sends a ReadDataByPacketIdentifier request with the sub-function parameter = $00 (stopSending) without any DPIDs in the request message.

8.19.5.4.1 First Sequence: Schedule Two DPIDs at Medium Rate.

- DPIDs: $10 (6 data bytes), $30 (4 data bytes).

Table 200: ReadDataByPacketIdentifier Request ($Level = $03) - Scheduled UUDT Positive Responses

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$04	$AA	$03	$30	$10	---	---	---
N(UUDT)	$541	$30	$32	$33	$EF	$44	---	---	---
:									
N(UUDT)	$541	$10	$21	$32	$15	$01	$11	$55	---

8.19.5.4.2 Second Sequence - Schedule 1 DPID at FastRate.

- DPID: $4C (three data bytes)

Table 201: ReadDataByPacketIdentifier Request ($Level = $04) - Scheduled UUDT Positive Responses

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$03	$AA	$04	$4C	---	---	---	---
N(UUDT)	$541	$4C	$32	$33	$EF	---	---	---	---
:									
N(UUDT)	$541	$4C	$32	$34	$FF	---	---	---	---
:									
N(UUDT)	$541	$30	$00	$11	$34	$00	---	---	---
:									
N(UUDT)	$541	$4C	$32	$34	$FF	---	---	---	---
:									
N(UUDT)	$541	$10	$22	$12	$00	$21	$01	$23	---

8.19.5.4.3 Third Sequence - Stop 1 of the MediumRate DPIDs.

- DPID: $30.

Table 202: ReadDataByPacketIdentifier Request ($Level = $00) - StopSending DPID# $30 Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$03	$AA	$00	$30	---	---	---	---
N(UUDT-SF)	$541	$00	---	---	---	---	---	---	---

8.19.5.4.4 Fourth Sequence - One-Shot 3rd DPID.

- DPID: $12 (two data bytes).
- The example below also shows that it is possible for an ECU to transmit UUDT messages for DPIDs which were previously scheduled before sending the one-shot response.

Table 203: ReadDataByPacketIdentifier Request ($Level = $01) - OneShot DPID# $12 Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$03	$AA	$01	$12	---	---	---	---
N(UUDT)	$541	$4C	$32	$33	$EF	---	---	---	---
N(UUDT)	$541	$12	$11	$DC	---	---	---	---	---

:

N(UUDT)	$541	$4C	$32	$34	$FF	---	---	---	---

:

N(UUDT)	$541	$10	$23	$12	$14	$01	$13	$F5	---

8.19.5.4.5 Fifth Sequence - Stop Schedule.

- Stop all scheduled DPIDs.

Table 204: ReadDataByPacketIdentifier Request ($Level = $00) - StopSending All DPIDs UUDT Positive Response

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$02	$AA	$00	---	---	---	---	---
N(UUDT-SF)	$541	$00	---	---	---	---	---	---	---

8.19.6 Node Interface Function.

8.19.6.1 Node Interface Data Dictionary (Table 205).

Table 205: Node Interface Data Dictionary of ReadDataByPacketIdentifier Service Pseudo Code

Variable/Meaning	Values
message_data_length **TesterPresent_Timer_State**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
PDS (Periodic DPID Scheduler) A multi-dimensional array of pointers and data used by the application's periodic scheduler. The array contains the scheduled DPIDs, DPID count down loop counters, and DPID data scheduling rate data.	N/A
PDS[].Transmit_Count This variable is counter which decrements by 1 each time the DecrementPDS_Counters() function is executed until the counter reaches $00. Once the counter hits $00, the DPID shall be transmitted as soon as possible.	$00 to Slow_Rate_Count
PDS[].Scheduling_Rate The scheduling rate for a scheduled DPID in the PDS.	Slow_Rate_Count, Medium_Rate_Count, or Fast_Rate_Count

Variable/Meaning	Values
PDS[].DPID Identifier (DPID) of the scheduled data.	$01 thru $7F, $90 thru $FE
PDS_Length Reference to the maximum length of the PDS (maximum number of DPIDs that can be scheduled simultaneously).	According to CTS
PDS_Number_Active The number of active scheduled DPIDs in the PDS.	0 to PDS_length
PDS_Xmit_Index This is an index variable which is used to keep track of the starting offset into the PDS each time the Process_AA_Msgs() function is executed. This is necessary to make sure that all DPIDs with expired timers will be sent before a single DPID may be sent twice.	0 to (PDS_length -1)
AA_Msg_Rate This is the amount of time between two executions of the Process_AA_Msgs() function.	x ms
all_DPIDs_valid A flag used to indicate whether all DPIDs in a request message are valid or not.	YES/NO
DPID_already_scheduled A flag used to indicate whether a DPID# is already in PDS or not.	YES/NO
vacant_position_found A flag used to indicate that a vacant position to put a new DPID# in the PDS has been found.	YES/NO
vacant_position A flag used to indicate the index into the PDS where the first vacant position to put a new DPID# in the PDS has been found.	0 to (PDS_length -1)
max_No_of_01_DPIDs The max. # of DPIDs that an ECU supports for a sub-function parameter $01 (OneShot) request.	According to CTS
num_new_sched_DPIDs The number of DPIDs in a received request that are not already scheduled at time of request.	0 to 4093
No_of_DPIDs The total number of DPIDs in a received request.	0 to 4093
Request_Valid This is a local flag used to keep track of whether or not the request message is a valid request.	YES/NO
request_DPIDs[] This is an array which contains all of the DPIDs from the request message. The first element (request_DPIDs[0]) of this array contains the first data byte after the sub function parameter in the request message.	Each array element contains a DPID number

Variable/Meaning	Values
Req_Index This is a counter which is used as an index into the array of requested DPIDs when trying to determine if all DPIDs in the request are valid.	0 to PDS_length
PDS_Index This is a counter which is used as an index into the PDS when modifying the DPIDs stored in the PDS. It is also used as an index into the array of supported DPIDs when determining if all DPIDs in the request are supported.	0 to max of either PDS_length or num_supported_DPIDs
Slow_Rate_Count The count refers to the minimum number of times that the Process_AA_Msgs() function shall be executed before sending out an item scheduled at the slow rate in the PDS.	Slow Rate/AA_Msg_Rate
Medium_Rate_Count The count refers to the minimum number of times that the Process_AA_Msgs() function shall be executed before sending out an item scheduled at the medium rate in the PDS.	Medium Rate/ AA_Msg_Rate
Fast_Rate_Count The count refers to the minimum number of times that the Process_AA_Msgs() function shall be executed before sending out an item scheduled at the fast rate in the PDS.	Fast Rate/AA_Msg_Rate
Req_Sched_Rate_Count This is a local variable which is set to Slow_Rate_Count, Medium_Rate_Count, or Fast_Rate_Count based on the rate in the requested message.	Slow_Rate_Count, Medium_Rate_Count, or Fast_Rate_Count
supported_DPIDs[] This is an array which contains all of the DPID numbers that the ECU supports.	N/A
num_supported_DPIDs This is a variable which indicates the number of entries in the supported_DPIDs[] array.	Determined in ECU software based on the number of DPIDs supported
Resp_01_DPIDs[] This is an array which is used to store the DPID numbers for the send one response level of this service. The size of this array shall be at least as large as the maximum number of DPIDs allowed with a send one response $Level = $01 request. This array shall be initialized at powerup to have all elements equal to $00.	This array shall contain at least max_No_of_01_DPIDs element
Resp_01_Index This is a variable used to index into the Resp_01_DPIDs[] array when sending the one shot responses.	$00 to max_No_of_01_DPIDs - 1
Resp_01_Num_To_Send This is a variable which is used to keep track of how many one shot responses still need to be sent. This variable shall be initialized to $00 at powerup.	$00 to max_No_of_01_DPIDs
Num_Loops_For_01 This is a counter which is used by the Process_AA_Msgs() function to determine when it is time to send a one shot response. This variable shall be initialized to $00 at powerup.	$00 to Max_01_Wait_Count

Variable/Meaning	Values
Max_01_Wait_Count This is a counter used to compare against the Num_Loops_For_01 counter to determine when to send a one shot DPID instead of a periodic scheduled DPID. The value of this variable should be set based on the number of times the Process_AA_Msgs() function gets called in the background before a $P2_C$ time interval will have elapsed. The objective is to set this variable to a value which will ensure that a one shot DPID will be sent at a time interval less than $P2_C$.	Value to be determined by ECU serial data design engineer.
Service_AA_Msg_Sent This is a flag used in the Process_AA_Msgs() function to keep track of whether or not a one shot or periodic DPID has been transmitted.	YES/NO
Min_Loops_Since_Periodic This is a counter which can be used to force a minimum amount of time between the transmission of two periodic DPIDs. This can be used to allow one shot DPIDs to be transmitted during loops when the periodics are not allowed.	Value determined at compile time of the ECU software.
Num_Loops_Since_Periodic This is a counter used to keep track of the number of times the Process_AA_Msgs() function gets called in the background without sending a periodic DPID.	$00 to Min_Loops_Since_Periodic

8.19.6.2 Node Interface Pseudo Code.

Powerup States:

```
Call ClearAndDisablePDS()
Resp_01_Num_To_Send ← $00
Num_Loops_For_01 ← $00
For (Resp_01_Index ← $00 TO (max_No_of_01_DPIDs - $01) BY $01)
    Resp_01_DPIDs[Resp_01_Index] ← $00
ENDFOR
```

The following logic is executed when a Service $AA message is received by the device:

```
BEGINFUNCTION Serv_AA_Msg_Recvd()
Request_Valid ← YES
IF (message_data_length > $02)
    No_of_DPIDs ← (message_data_length-$02) /*Calculate number of DPIDs in request*/
ELSE
    No_of_DPIDs ← $00
ENDIF
IF ( ($Level > LEV_SAFR) OR (message_data_length < $02) OR
(($Level > LEV_SS) AND (No_of_DPIDs < $01)) OR
((No_of_DPIDs > max_No_of_01_DPIDs) AND ($Level = LEV_SOR)) OR
((No_of_DPIDs > PDS_Length) AND ($Level != LEV_SOR)) ) THEN
    Send Negative Response ($7F $AA $12) /*Invalid Format*/
    Request_Valid ← NO
ELSE IF ($Level != LEV_SS) THEN
    /*********************************************************************
    Check if all DPIDs are valid (no check performed if $Level =$00) and also
    check how many DPIDs in the request are not already scheduled (if the
    sub-function parameter($Level)is $02, $03 or $04). These checks are performed
    in order to determine whether its possible to execute the request
    ******************************************************************/
    all_DPIDs_valid ← YES
    num_new_sched_DPIDs ← No_of_DPIDs
    Req_Index ← $00 /*local counter to traverse the DPIDs in request*/
    REPEAT
        all_DPIDs_valid ← NO
        FOR (PDS_Index ← $00 TO (num_supported_DPIDs - $01) BY $01)
            IF (request_DPIDs[Req_Index] = supported_DPIDs[PDS_Index]) THEN
                all_DPIDs_valid ← YES
                PDS_Index ← num_supported_DPIDs
            ENDIF
        ENDFOR
        IF ((all_DPIDs_valid = YES) AND ($Level > LEV_SOR)) THEN
            FOR (PDS_Index ← $00 TO (PDS_Length - $01) BY $01)
                IF (request_DPIDs[Req_Index] = PDS[PDS_Index].DPID) THEN
                    num_new_sched_DPIDs ← (num_new_sched_DPIDs - $01)
                    PDS_Index ← PDS_Length
                ENDIF
            ENDFOR
        ENDIF
        Req_Index ← Req_Index + $01
    UNTIL ((all_DPIDs_valid = NO) OR (Req_Index = No_of_DPIDs))
    IF (all_DPIDs_valid = NO) THEN
        Send Negative Response ($7F $AA $31) /*Request Out Of Range*/
        Request_Valid ← NO
    ELSE IF (($Level > LEV_SOR) AND
    (num_new_sched_DPIDs > (PDS_Length - PDS_number_active))) THEN
        /*the PDS would Overflow if the Request was Processed*/
        Send Negative Response ($7F $AA $81) /*Scheduler Full*/
        Request_Valid ← NO
    ENDIF
ENDIF
/******************************************************************
 If Request Message is Valid, Process Service AA Message
*******************************************************************/
IF (Request_Valid = YES) THEN
    SELECT FIRST
    WHEN ($Level = LEV_SS ) /* $00 Stop Sending Periodic Data */
        IF (No_of_DPIDs!= $00) THEN /*Request includes DPID(s)*/
            FOR (Req_Index ← $00 TO (No_of_DPIDs - $01) BY $01)
                FOR (PDS_Index ← $00 TO (PDS_Length - $01) BY $01)
                    IF (request_DPIDs[Req_Index] = PDS[PDS_Index].DPID) THEN
                        PDS[PDS_Index].DPID ← $00
                        PDS_Number_Active ← (PDS_Number_Active - $01)
                        PDS_Index ← PDS_Length
                    ENDIF
                ENDFOR
            ENDFOR
            IF (PDS_Number_Active = $00)
                PDS_Xmit_Index ← $00
            ENDIF
        ELSE
            Call ClearAndDisablePDS()
        ENDIF
        Send Positive Response ($UUDTCANId $00)
    WHEN ($Level = LEV_SOR ) /* $01 Send One Shot Responses*/
        IF (processing the first response will take more than $P2_C$ ms) THEN
            Send Negative Response ($7F $AA $78) /*for request received-response pending */
        ENDIF
        FOR (Req_Index ← $00 TO (No_of_DPIDs - $01) BY $01)
            Resp_01_DPIDs[Req_Index] ← request_DPIDs[Req_Index]
        ENDFOR
        Resp_01_Num_To_Send ← No_of_DPIDs
        Num_Loops_For_01 ← Max_01_Wait_Count /* ensure one shot is sent next time Process_AA_Msgs()
        function is called */
    WHEN ($Level > LEV_SOR ) /* $02, $03, or $04 Periodic Rates */
        IF (processing the first response will take more than $P2_C$ ms) THEN
            Send Negative Response ($7F $AA $78) /*for request received-response pending */
        ENDIF
        TesterPresent_Timer_State ← ACTIVE
        IF ($Level = LEV_SAFR) THEN
            Req_Sched_Rate_Count ← Fast_Rate_Count
        ELSE IF ($Level = LEV_SAMR) THEN
            Req_Sched_Rate_Count ← Medium_Rate_Count
        ELSE
            Req_Sched_Rate_Count ← Slow_Rate_Count
        ENDIF
        /*************************************************************
        * Check for each DPID in request to see if it is already scheduled
        * (present in PDS). If already scheduled, clear the transmit counter
        * so the DPID will be sent and then update the data rate for that DPID.
        * If the DPID is not in the PDS, then put it into the first vacant
        * location in the PDS, set its rate, and initialize its transmit
        * counter to $00 so it will be sent as soon as possible.
        *************************************************************/
        vacant_position ← $00
        Num_Loops_Since_Periodic ← Min_Loops_Since_Periodic
        FOR (Req_Index ← $00 TO (No_of_DPIDs - $01) BY $01)
            PDS_Index ← $00 /*Start with first row in PDS each loop*/
            DPID_already_scheduled ← NO
            vacant_position_found ← NO
```

```
            REPEAT
                IF (request_DPIDs[Req_Index] = PDS[PDS_Index].DPID)
                    DPID_already_scheduled ← YES
                    PDS[PDS_Index].Transmit_Count ← $00
                    PDS[PDS_Index].Scheduling_Rate ← Req_Sched_Rate_Count
                ELSE IF ((PDS[PDS_Index].DPID = $00) AND
                (vacant_position_found = NO)) THEN
                    vacant_position ← PDS_Index
                    vacant_position_found ← YES
                ENDIF
                PDS_Index ← PDS_Index +1
            UNTIL ((DPID_already_scheduled = YES)OR(PDS_Index = PDS_Length))
                IF (DPID_already_scheduled = NO)
                    PDS_Number_Active ← (PDS_Number_Active + $01)
                    PDS[vacant_position].DPID ← request_DPIDs[Req_Index]
                    PDS[vacant_position].Transmit_Count ← $00
                    PDS[vacant_position].Scheduling_Rate ← Req_Sched_Rate_Count
                ENDIF
        ENDFOR
    ENDSELECT
ENDIF
ENDFUNCTION
```

The following function is called from the Process_AA_Msgs() function and is used to control sending the periodic DPIDs in the PDS.

```
BEGINFUNCTION Send_AA_Periodic_Msgs()
/* check if the PDS should be running by looking at the number of DPIDs in the PDS */
IF (PDS_Number_Active > $00)THEN
    /*Decrement transmit count by one unless already $00*/
    CALL DecrementPDS_Counters()
    /* check to see if any DPIDs need to be sent */
    IF (Num_Loops_Since_Periodic ≥ Min_Loops_Since_Periodic) THEN
        FOR (PDS_Index ← $01 TO PDS_Length BY $01)
            IF ( (PDS[PDS_Xmit_Index].DPID != $00) AND
            (PDS[PDS_Xmit_Index].Transmit_Count = $00)
                /* Get current data (DB_1,... DBx) for this DPID and send it*/
                Send positive response ($UUDTCANId $PDS[PDS_Xmit_Index].DPID $DB_1, ....)
                PDS[PDS_Xmit_Index].Transmit_Count ← PDS[PDS_Xmit_Index].Scheduling_Rate
                PDS_Index ← PDS_Length
                Service_AA_Msg_Sent ← YES
                Num_Loops_Since_Periodic ← $00
            ENDIF
        PDS_Xmit_Index ← ((PDS_Xmit_Index + $01) MOD PDS_Length)
        ENDFOR
    ENDIF
ENDIF
ENDFUNCTION
```

The following function, Send_AA_One_Shot(), is called by the background function Process_AA_Msgs() that handles processing one time and periodic responses for the $AA service. This function will send one DPID each time that it is called.

```
BEGINFUNCTION Send_AA_One_Shot()
IF (Resp_01_Num_To_Send > $00) THEN
    FOR (Resp_01_Index ← $00 TO (max_No_of_01_DPIDs - $01) BY $01)
        IF (Resp_01_DPIDs[Resp_01_Index] != $00)
            /* Get current data (DB_1,... DBx) for this DPID and send it*/
            Send positive response ($UUDTCANId $Resp_01_DPIDs[Resp_01_Index] $DB_1, ....)
            Resp_01_Num_To_Send ← (Resp_01_Num_To_Send - $01)
            Resp_01_DPIDs[Resp_01_Index] ← $00
            Resp_01_Index ← max_No_of_01_DPIDs
            Num_Loops_For_01 ← $00
            Service_AA_Msg_Sent ← YES
        ENDIF
    ENDFOR
ENDIF
ENDFUNCTION
```

The following function, Process_AA_Msgs(), is called in a background loop and handles sending one shot ($Level = LEV_SOR) and periodic ($Level = LEV_SAFR, LEV_SAMR, or LEV_SASR) DPID responses. The execution rate of this function shall be fast enough to ensure that each element of the PDS has the opportunity to be transmitted within the $P2_C$* timing window. This ensures a maximum of 1 negative response with response code $78 per periodic request. This execution rate of this function shall also be at an interval less than or equal to half of the fast periodic rate. If the value of the fast periodic rate is close to the value of $P2_C$, then the execution interval must be small enough to ensure that responses to level $01 requests can meet the $P2_C$ timing requirements.

```
BEGINFUNCTION Process_AA_Msgs()
IF ((PDS_Number_Active > $00) OR (Resp_01_Num_To_Send > $00)) THEN
    Service_AA_Msg_Sent ← NO
    IF (Num_Loops_For_01 ≥ Max_01_Wait_Count) THEN
        CALL Send_AA_One_Shot()
        IF (PDS_Number_Active > $00) THEN
            CALL DecrementPDS_Counters()
        ENDIF
    ELSE
        CALL Send_AA_Periodic_Msgs()
        IF (Service_AA_Msg_Sent = NO) THEN
            CALL Send_AA_One_Shot()
        ELSE
            IF (Resp_01_Num_To_Send > $00) THEN
                Num_Loops_For_01 ← (Num_Loops_For_01 + $01)
            ENDIF
        ENDIF
    ENDIF
ENDIF
ENDFUNCTION
```

The following function, DecrementPDS_Counters, is used to decrement the loop counters associated with each DPID in the PDS. The loop counters are used to determine when a DPID is ready for transmission. This function is called from the Send_AA_Periodic_Msgs() function and also from the Process_AA_Msgs() function.

```
BEGINFUNCTION DecrementPDS_Counters()
FOR (PDS_Index ← $00 TO (PDS_Length - $01) BY $01)
    IF ((PDS[PDS_Index].DPID!=$00) AND (PDS[PDS_Index].Transmit_Count>$00)) THEN
        PDS[PDS_Index].Transmit_Count←(PDS[PDS_Index].Transmit_Count-$01)
    ENDIF
ENDFOR
IF (Num_Loops_Since_Periodic < Min_Loops_Since_Periodic) THEN
    Num_Loops_Since_Periodic ← (Num_Loops_Since_Periodic + $01)
ENDIF
ENDFUNCTION
```

The following function, ClearAndDisablePDS, is called by this and other services to stop and clear the PDS

```
BEGINFUNCTION ClearAndDisablePDS()
FOR (PDS_Index ← $00 TO (PDS_Length -$01) BY $01)
    PDS[PDS_Index].DPID ← $00
ENDFOR
PDS_Number_Active ← $00
PDS_Xmit_Index ← $00
```

8.19.6.3 Graphical and Tabular Examples of Mode $AA Periodic Scheduled Rates. This section contains two examples of scheduled periodic data. Each example contains a graphical depiction of what messages are transmitted on the bus, followed by a table which shows the PDS variables and how they change each time the Process_AA_Msgs() function is executed. In the examples below, the following information is given:

- The fast rate is 25 ms.
- The medium rate is 200 ms for Figure 34.
- The medium rate is 300 ms for Figure 35.
- The Process_AA_Msgs() function is executed every 12.5 ms (AA_Msg_Rate = 12.5 ms).
- The PDS can hold a maximum of four scheduled items.
- It is possible to send a DPID any time its counter has expired (Min_Loops_Since_Periodic = 0).

Since the AA_Msg_Rate is 12.5 ms, the fast rate loop counter would be set to 2 (this value is based on the scheduled rate (25 ms) divided by the AA_Msg_Rate (12.5 ms) or 25/12.5) each time a fast rate DPID is sent and the medium rate loop counter is reset to 16 (or 24) (scheduled rate divided by AA_Msg_Rate or 200 (or 300)/12.5) each time a medium rate DPID is sent.

8.19.6.3.1 Example #1 - Single DPID Schedule Rates. In this example, four DPIDs ($01, $02, $03, and $04) are scheduled at the medium rate. At t = 0.0 ms, the tool begins sending the request to schedule the four DPIDs at the medium rate. For the purposes of this example, the ECU receives the request and executes the PDS code for the first time t = 25.0 ms. See Figure 34 and Table 206.

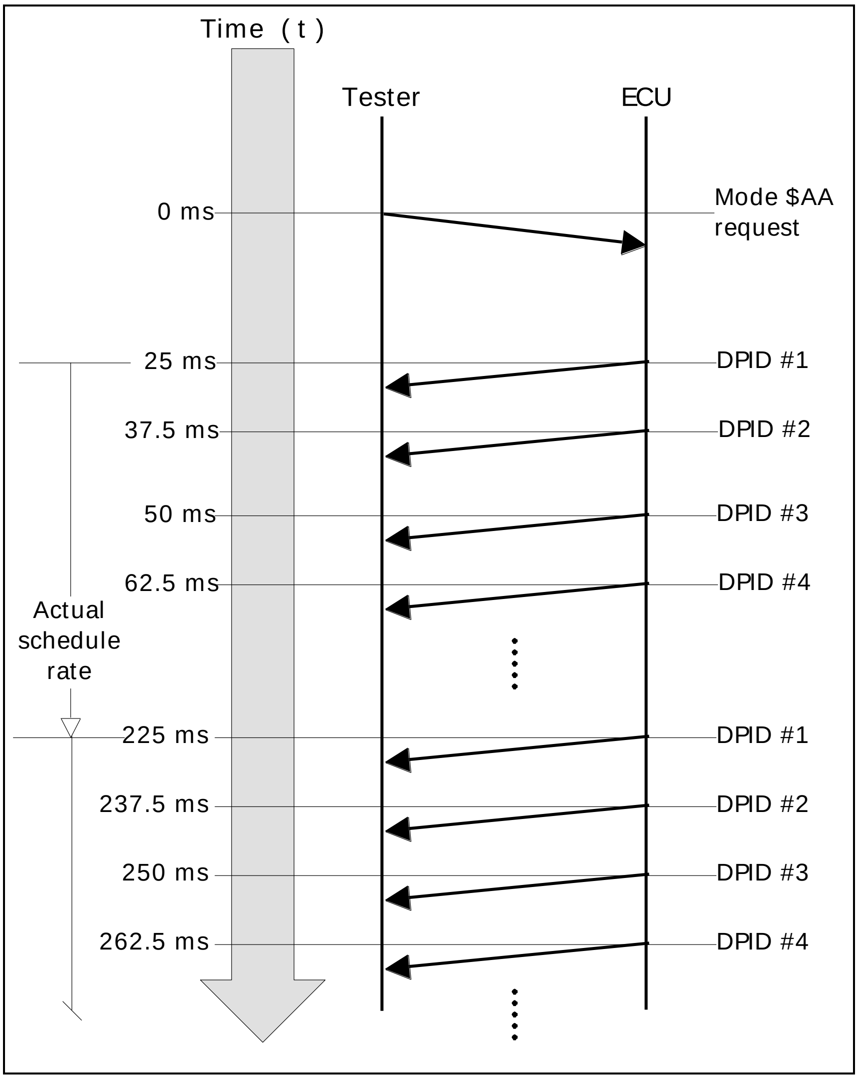


Figure 34: Example #1 - Four DPIDs at Medium Rate (200 ms) with 12.5 ms AA_Msg_Rate Rate

Table 206: Example #1 - Four DPIDs at Medium Rate (200 ms) with 12.5 ms AA_Msg_Rate Rate

t (ms)	PDS Xmit Index	DPID Sent	PDS Loop #	PDS[0] Transmit Count	PDS[1] Transmit Count	PDS[2] Transmit Count	PDS[3] Transmit Count
25.0	0	1	1	0 to 16	0	0	0
37.5	1	2	2	15	0 to 16	0	0
50.0	2	3	3	14	15	0 to 16	0
62.5	3	4	4	13	14	15	0 to 16
75.0	0	None	5	12	13	14	15
87.5	0	None	6	11	12	13	14
100.0	0	None	7	10	11	12	13
112.5	0	None	8	9	10	11	12
125.0	0	None	9	8	9	10	11
137.5	0	None	10	7	8	9	10
150.0	0	None	11	6	7	8	9
162.5	0	None	12	5	6	7	8
175.0	0	None	13	4	5	6	7
187.5	0	None	14	3	4	5	6
200.0	0	None	15	2	3	4	5
212.5	0	None	16	1	2	3	4
225.0	0	1	17	0 to 16	1	2	3
237.5	1	2	18	15	0 to 16	1	2
250.0	2	3	19	14	15	0 to 16	1
262.5	3	4	20	13	14	15	0 to 16

8.19.6.3.2 Example #2 - Mixed DPID Schedule Rates. In this example, three DPIDs ($01, $02, $03) are scheduled at the fast rate and then another request is sent for a single DPID ($04) to be scheduled at the medium rate. For the purposes of this example, the ECU receives the first mode $AA request and executes the PDS code for the first time t = 25.0 ms. The second mode $AA request is received and processed just prior to the device executing the Process_AA_Msgs() function at t = 50.0 ms. See Figure 35 and Table 207.

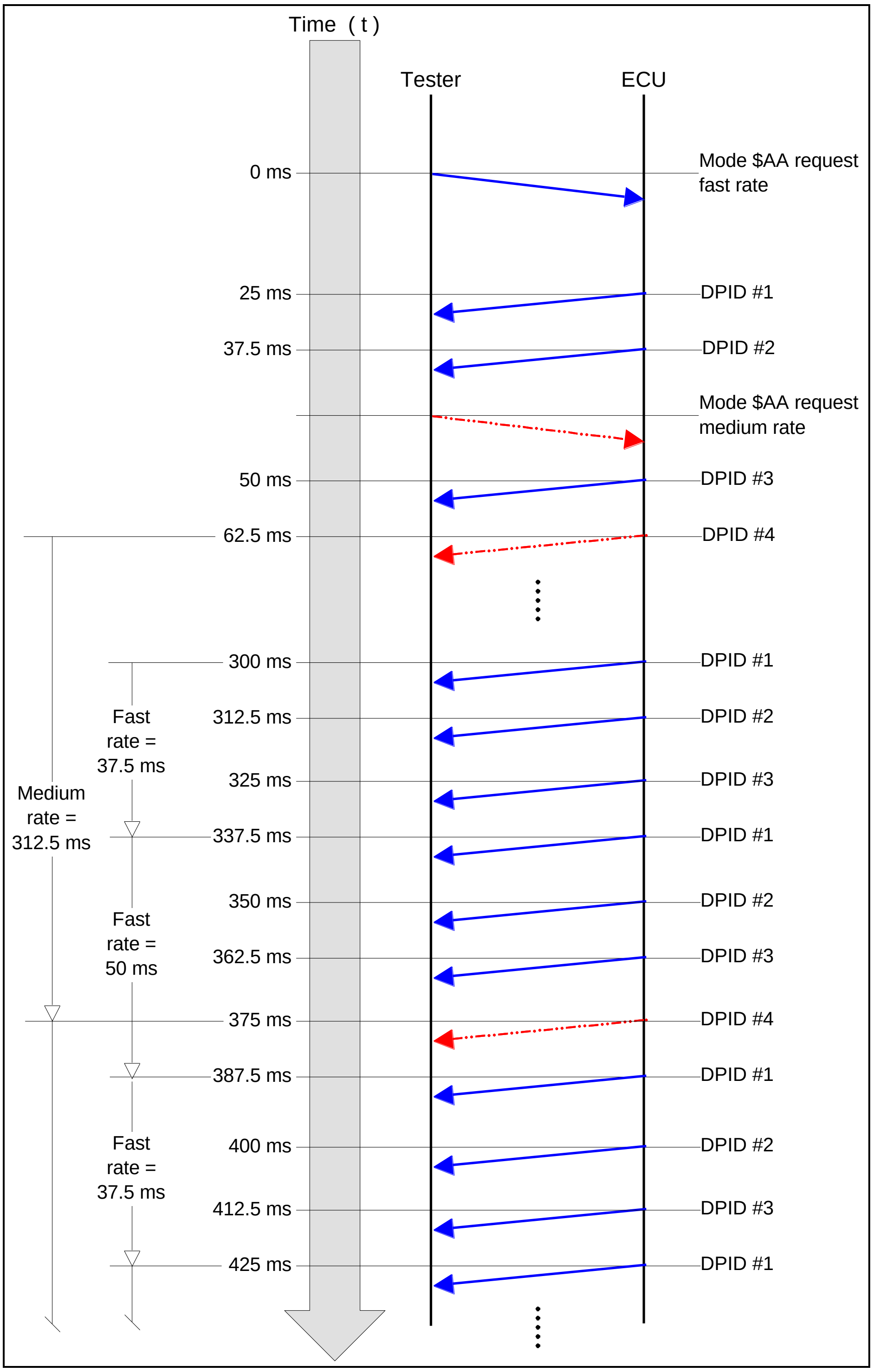


Figure 35: Example #2 - Mixture of DPID Schedule Rates

Table 207: Example #2 - Three DPIDs at Fast Rate (25 ms) and 1 DPID at Medium Rate (300 ms)

t (ms)	PDS Xmit Index	DPID Sent	PDS Loop #	PDS[0] Transmit Count	PDS[1] Transmit Count	PDS[2] Transmit Count	PDS[3] Transmit Count
25.0	0	1	1	0 to 2	0	0	N/A
37.5	1	2	2	1	0 to 2	0	N/A
50.0	2	3	3	0	1	0 to 2	0
62.5	3	4	4	0	0	1	0 to 24
75.0	0	1	5	0 to 2	0	0	23
87.5	1	2	6	1	0 to 2	0	22
100.0	2	3	7	0	1	0 to 2	21
112.5	3	1	8	0 to 2	0	1	20
125.0	1	2	9	1	0 to 2	0	19
137.5	2	3	10	0	1	0 to 2	18
150.0	3	1	11	0 to 2	0	1	17
162.5	1	2	12	1	0 to 2	0	16
175.0	2	3	13	0	1	0 to 2	15
187.5	3	1	14	0 to 2	0	1	14
200.0	1	2	15	1	0 to 2	0	13
212.5	2	3	16	0	1	0 to 2	12
225.0	3	1	17	0 to 2	0	1	11
237.5	1	2	18	1	0 to 2	0	10
250.0	2	3	19	0	1	0 to 2	9
262.5	3	1	20	0 to 2	0	1	8
275.0	1	2	21	1	0 to 2	0	7
287.5	2	3	22	0	1	0 to 2	6
300.0	3	1	23	0 to 2	0	1	5
312.5	1	2	24	1	0 to 2	0	4
325.0	2	3	25	0	1	0 to 2	3
337.5	3	1	26	0 to 2	0	1	2
350.0	1	2	27	1	0 to 2	0	1
362.5	2	3	28	0	1	0 to 2	0
375.0	3	4	29	0	0	1	0 to 24
387.5	0	1	30	0 to 2	0	0	23

8.19.7 Node Verification Procedure.

Procedure 1: All $AA requests for this procedure are with $Level = LEV_SOR ($01).

1. Send a request (including only valid DPIDs) with the total number of DPIDs less than the maximum number supported by the ECU for this $Level. Verify the proper response(s).
2. Send a request (including only valid DPIDs) with the total number of DPIDs requested equal to the maximum number supported by the ECU for this $Level. Verify the proper responses.
3. Send a request (including only valid DPIDs) with the total number of DPIDs requested greater than the maximum number supported by the ECU for this $Level. Verify negative response ($7F $AA $12).
4. Send A request with no DPIDs. Verify negative response ($7F $AA $12).
5. Send a request with an invalid DPID. Verify negative response ($7F $AA $31).
6. Send a request with at least one valid and one invalid DPID. Verify that the only response to this request is the negative response ($7F $AA $31).
7. Verify that it is possible to retrieve all supported DPIDs in the ECU with one or more requests of this service.

Procedure 2:

1. Request a periodic (valid) DPID. Verify proper response, and proper response timing. Repeat this test for each supported periodic rate.
2. After completing step 1 of this procedure, verify that the periodic DPID data stops after $P3_{Cmax}$ ms (with no additional diagnostic requests).
3. With the periodic scheduler inactive, request the maximum number of DPIDs (all items) at one of the supported periodic rates. Verify proper responses and response timing for each scheduled DPID. Send $3E messages at least once every $P3_C$ ms and verify that the responses continue to be properly sent for five minutes.
4. Repeat step 3 of this procedure for each additional supported periodic rate.
5. Send a single periodic request with more DPIDs than the ECU can schedule. Verify the negative response ($7F $AA $12).
6. Request a number of DPIDs to be sent periodically at one supported rate. Request additional DPIDs to be sent at another supported rate (not equal to the first rate) until DPIDs are sent with all supported rates simultaneously (If the number of DPIDs the PDS supports is less than the number of supported periodic rates, then repeat this step for all possible combinations of supported data rates). Verify proper responses and response timing. Send $3E messages at least once every $P3_C$ ms and verify that the responses continue to be properly sent for five minutes.
7. Request an additional (valid) DPID to be scheduled when the DPID scheduler is already full and verify the proper negative response ($7F $AA $81). Verify that the previously scheduled DPIDs continue to be sent at the rates scheduled prior to the last request.
8. Send a periodic request containing a DPID which is already in the PDS, using a sub-function parameter which will result in a change to the data rate of the requested DPID. Perform this test while the scheduler is full. Verify that the requested DPID changes data rates, that the data rate of the other DPIDs in the PDS do not change, and that no negative response for scheduler full is sent.
9. Send a periodic request containing both valid and invalid DPIDs (with the total number of DPIDs in the request less than the maximum number of periodic DPIDs allowed in the PDS). Verify the negative response ($7F $AA $31). Repeat this step for each supported periodic rate.
10. Verify that it is possible to periodically schedule all supported DPIDs in the ECU.
11. Send a periodic request with no DPIDs, verify negative response ($7F $AA $12). Repeat for each periodic rate supported.

Procedure 3:

1. Verify that periodically sent items (DPIDs) can be stopped individually with Service $AA sub-function parameter $00 which includes DPIDs to be stopped. Verify that DPIDs in request are stopped and that DPIDs which are not in the request (but in the PDS) continue to be sent. Verify proper positive response.
2. Verify that all the periodic scheduler items can be stopped with Service $AA sub-function parameter $00 (sent without any DPIDs). Verify proper positive response and that no scheduled DPIDs are sent after the positive response.
3. Send a request with a sub-function parameter equal to $00 and additional DPIDs in the request such that the number of DPIDs exceeds the maximum number that can be scheduled. Verify the proper negative response ($7F $AA $12).

Procedure 4:

1. Request a number of DPIDs to be sent periodically at one supported rate. Request additional DPIDs to be sent at another supported rate (Not equal to the first rate). Repeat until DPIDs are sent with all supported rates simultaneously. Request one valid DPID as One-shot with service $AA sub-function parameter $01. Verify responses, timing.
2. Request a number of DPIDs to be sent periodically at one supported rate. Request additional DPIDs to be sent at another supported rate (Not equal to the first rate). Repeat until DPIDs are sent with all supported rates simultaneously. Request Maximum number of (valid) DPIDs as One-shot with service $AA sub-function parameter $01. Verify responses, timing.
3. Request a number of DPIDs to be sent periodically at one supported rate. Request additional DPIDs to be sent at another supported rate (Not equal to the first rate). Repeat until DPIDs are sent with all supported rates simultaneously. Request the same DPIDs which are scheduled as One-shot with service $AA sub-function parameter ($Level) $01 without stopping the PDS. Verify proper one-shot responses, timing. Verify that proper scheduling is maintained.

Procedure 5:

1. Send a service $AA without sub-function parameter and DPID parameters. Verify negative response ($7F $AA $12).
2. Verify that a service $AA request with all non-supported sub-function parameters is negatively responded to with a negative response ($7F $AA $12).

Procedure 6: (Additional checks for ECUs that support dynamic DPIDs.)

1. Verify that a dynamic DPID can be redefined using the service $2C while that DPID is currently in the periodic scheduler. Verify that the data is correct for the newly defined DPID by the second transmission of that DPID after the $2C positive response. (The first response of that DPID may be invalid since the DPID may have already been queued for transmission prior to the redefinition).
2. Request a dynamic DPID prior to defining its contents via the $2C service. Verify that a positive response is sent with no data after the DPID number.
3. Repeat step 2 for all supported one shot and periodic rates.

Procedure 7: (Additional checks if the ECU supports negative response code $78.)

1. Create conditions under which the ECU should return the $7F $AA $78 response. Send a valid request for this service (including applicable sub-function parameter) and verify that the $7F $AA $78 response is received followed by the appropriate final response within the $P2_{C*}$ timing window.
2. Repeat step 1 of this procedure for each possible reason an ECU would send the negative response with response code $78. Verify this for each applicable supported sub-function parameter.

8.19.8 Tester Implications. It is recommended that a tester use this service with physical addressing. Functionally addressed requests for this service may result in unwanted negative response messages from one or multiple ECUs.

Positive response(s) to this service are formatted as UUDT messages. Since it is possible for multiple UUDT responses to be sent as a result of a single request, the tester must ensure that it properly handles the timing requirements outlined in the description section of this service for each available sub-function.

The test device shall send a Service $3E message at least once every $P3_C$ ms when using any of the periodic rates of service $AA, otherwise the device will exit the periodic transmission as if receiving a Service $AA sub-function parameter $00 request (without DPIDs).

If a tester sends a periodic request with multiple copies of the same DPID in the request, and the DPID (which was included more than once) is not currently in the PDS, then the tester may receive a $7F $81 (scheduler full) negative response (based on the number of open items in the PDS at the time of the request).

8.20 DeviceControl ($AE) Service. The purpose of this service is to allow a test device to override normal output control functions in order to verify proper operation of a component or system, or to reset/clear variables used within normal control algorithms. The tool may take control of multiple outputs simultaneously with a single request message or by sending multiple device control service messages.

8.20.1 Service Description. Device control is performed by manipulating predefined bits and/or bytes within a message to indicate to the device which output(s) or control function(s) the tool wants to override. When a CPID is defined with multiple output controls, the ECU shall support simultaneous control of multiple outputs contained in that CPID as part of a single diagnostic request.

Note: Certain combinations of outputs may not be allowed to be controlled at the same time due to safety or to avoid product damage. Any such restrictions shall be negotiated between the DRE, service and manufacturing engineering and documented in a CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceding documents.

Each device control message shall fit in a single frame and as such contains a maximum of six data bytes after the service Identifier byte. The first data byte (after the service identifier) of a device control request message is a Control Packet Identifier (CPID). The remaining five bytes are used to provide output control information. However, it is not required that a CPID contain five bytes of control information. CPIDs are used to distinguish between different output control packets for a given device. Valid CPID numbers can range from $01 thru $FE. Corporate common CPIDs shall be defined starting at CPID $FE and numbered sequentially backwards (see Appendix B for a list of corporate common CPIDs). A $00 in place of the CPID number in the request message is used to completely cancel all active device controls. If a device supports multiple CPIDs, then receipt of a second device control request with a different CPID number than that of a previous request, shall not result in any change to the functions requested with the previous device control request.

The DeviceControl service shall be given the highest priority in the devices output control logic unless control of the given output could result in vehicle damage, or there is potential risk to the vehicles occupants. If vehicle damage or safety related concerns exist, DeviceControl limitations shall be built into the device’s software. DeviceControl limitations are automatically active upon the receipt of an $AE message. If the device detects that a DeviceControl limitation has been exceeded, all active DeviceControls shall be terminated and the device shall send a $7F $AE $E3 $xx $yy (DeviceControlLimitsExceeded) reject message. The DeviceControlLimitsExceeded reject message may be sent in response to a DeviceControl request message or as an unsolicited response any time the application software detects that a limit has been exceeded. The $xx $yy portion of the reject message shall contain an application specific state encoded value which will allow the tool to determine which device control limit was exceeded. The master list of values for $xx $yy is maintained on the GM Service and Parts Operations web page. The values for the device control limits exceeded return codes implemented for a specific ECU must follow the master list and shall be documented in the ECU CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceeding documents.

Note: If a device control limit is exceeded which warrants an unsolicited negative response message and the ECU is in process of receiving or transmitting a multi-frame USDT diagnostic message, then the unsolicited negative response message shall be delayed until after the multi-frame message (reception or transmission) has completed.

The ECU shall support reactivation of device controls (provided that the additional requests do not result in restrictions being exceeded) after a device control restriction has been exceeded without requiring extraordinary measures such as a key cycle or cycling of power. Any additional limitations imposed on a device control after being initially exceeded (that are beyond those that caused the device control to be initially exceeded) shall be agreed upon by the DRE, service and manufacturing engineering and shall be documented in a CTS, SSTS, or supplemental diagnostic specification referenced by either of the preceeding documents.

Certain DeviceControl restrictions may be bypassed for the purposes of vehicle manufacturing testing. A tester can bypass device control restrictions via the SecurityAccess ($27) service using the sub-parameters assigned for device control limitations.

Note: Security Access levels used for device control restrictions overrides are different than those for SPS security. The SPS security levels are used for reprogramming and with secure CPIDs. The device control security levels are used to accommodate manufacturing device control overrides. See pseudo code of this service for more detail.

All device control limitations supported by an ECU along with the list of restrictions which can be bypassed in manufacturing, must be documented in a CTS, SSTS, or a supplemental diagnostic specification referenced by a CTS or SSTS. The type of DeviceControl restrictions enforced (service or Assembly plant) shall remain unaffected if a DeviceControl restriction is exceeded. Refer to the SecurityAccess ($27) and the ReturnToNormalMode ($20) service descriptions within this specification for additional details on how the type of DeviceControl restriction is determined.

Upon receiving a DeviceControl request message, an ECU shall normally inhibit the setting of Diagnostic Trouble Codes (DTCs). This is done to prevent DTCs from setting as a result of the off board testers modifications to the normal output control algorithms. Once DTCs are inhibited, they will remain inhibited until a TesterPresent ($3E) timeout occurs or until a ReturnToNormalMode ($20) service request is received. In order to facilitate vehicle manufacturing needs to quickly diagnose vehicles during the assembly process, certain ECUs shall be required to keep diagnostic algorithms active while DeviceControl functions are active. The tester can keep the diagnostic algorithms active during DeviceControl by sending the appropriate InitiateDiagnosticOperation (service $10) request prior to activating DeviceControl. See service $10 for more information.

This service requires that a TesterPresent ($3E) service be sent at least once every $P3_C$ ms or the device will cancel all active device controls and resume normal control of its outputs.

If a device has any CPIDs which are used during SPS programming, are required for theft related functions (e.g., deterrent key re-learn), or are somehow safety related, the device must then support the SPS security levels of the SecurityAccess ($27) service.

A given device control is allowed to be packed into more than one CPID if necessary to meet test timing requirements. If a given device control is in more than one CPID, the actions requested by the last message containing that device control shall remain active.

All device controls shall be terminated if any of the following occur:

- A TesterPresent ($P3_C$) time-out occurs.
- A device control request is received with a CPID number of $00.
- A device control limit is exceeded on any active device control.
- A ReturnToNormalMode($20) request message is received.
- The device is powered down.

An individual device control can be terminated in the following manner:

1. A device control request is received with the same CPID number and the enabling criteria turned off.
2. Any of the criteria listed above to cancel all device controls are met.

Note: ECU normal control algorithms determine the state of a given output once the appropriate DeviceControl has been terminated (unless the termination is the result of an ECU power down).

This service shall be allowed while other diagnostic services are also active (e.g., diagnostic periodic data scheduler, etc.) If data is requested which includes the state of a node’s outputs while those outputs are under device control, the data representing the output state shall reflect its commanded state. In other words, if a node’s normal output control algorithm is commanding an output to the off state and a DeviceControl request is processed which turns that same output to the on state, periodic or other data containing the state of the output shall indicate that the output is on.

8.20.2 Request Message Definition (Table 208).

Table 208: DeviceControl Request Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DeviceControl Request Service Id	M	AE	SIDRQ
#2	CPIDNumber	M	00 thru FE	CPID
	CPIDControlBytes[] = [	U		CPIDCB_
#3	ControlByte#1		00 thru FF	CB_1
#4	ControlByte#2		00 thru FF	CB_2
#5	ControlByte#3		00 thru FF	CB_3
#6	ControlByte#4		00 thru FF	CB_4
#7	ControlByte#5]		00 thru FF	CB_5

8.20.2.1 Request Message Sub-Function Parameter $Level (LEV_) Definition. This service does not use a sub-function parameter.

8.20.2.2 Request Message Data Parameter Definition. Table 209 specifies the data parameter definitions for this service.

Table 209: DeviceControl Request Message Data Parameter Definition

Definition
CPIDNumber See message description section for definition of a CPID.
CPIDControlBytes[] The control bytes are mapped such that multiple outputs can be controlled with a single request for this service. Outputs which are likely to be controlled simultaneously during diagnostics should be grouped into the same CPID.

8.20.3 Positive Response Message Definition (Table 210).

Table 210: DeviceControl Positive Response Message

Data Byte	Parameter Name	Cvt	Hex Value	Mnemonic
#1	DeviceControl Positive Response Service Id	M	EE	SIDPR
#2	CPIDNumber	M	00 thru FE	CPID

8.20.3.1 Positive Response Message Data Parameter Definition (Table 211).

Table 211: DeviceControl Response Message Data Parameter Definition

Definition
CPIDNumber This parameter is an echo of the CPID number from the request message.

8.20.4 Supported Negative Response Codes (RC_). The following negative response codes shall be implemented for this service. The circumstances under which each response code would occur are documented in Table 212.

Table 212: Supported Negative Response Codes

Hex	Description	Cvt	Mnemonic
12	**SubFunctionNotSupported-InvalidFormat** • The length of the message is such that there is no CPID number in the request. • The length of the message for a supported CPID does not match the length expected by the device. • See 7.2 Return Code Definition.	M	SFNS-IF
31	**RequestOutOfRange** • This code shall be returned if the CPID requested is not supported by the device. • This code shall be returned if a tester sends a request with a valid secure CPID and the device's security feature is currently active.	M	ROOR
78	**RequestCorrectlyReceived-ResponsePending** See 7.2 Return Code Definition.	C	RCR-RP
E3	**DeviceControlLimitsExceeded** This code shall be returned any time a device exits device control due to its internal protection algorithms. With this code, 2 additional bytes will be included in the reject message to indicate which limit was exceeded.	C	DCLE

8.20.5 Message Flow Example DeviceControl. The example in this section will show how a tool could send device control messages to a Powertrain Control Module (PCM) to control two of its outputs.

8.20.5.1 DeviceControl Example Data. The output control mapping is based on the CPID control byte definitions in Tables 213 and 214, and the brief descriptions that follow:

Table 213: Example Data PCM CPID $01 Definition

Control Byte #	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1								
2							IAC 0 = POS; 1 = rpm	IAC Control Enable
3								
4								
5	IAC Pintle Position (n = counts) or Desired Engine rpm (rpm = n × 12.5)							

Based on the above definition, CPID $01 contains 5 bytes of control data. CPID $01 allows the tool to take control of the Idle Air Control (IAC) motor by placing a 1 in bit 0 of byte 2, and then command a pintle position or desired engine idle speed (based on the value of byte 2 bit 1) by placing the appropriate value in byte 5. The unused bits/bytes are ignored for the purposes of the examples in this section.

Table 214: Example Data PCM CPID $02 Definition

Control Byte #	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1								EGR Enable
2								
3	EGR Duty Cycle: Linear Scaling 0 counts = 0%, 255 counts = 100%							
4								

Based on the above definition, CPID $02 contains 4 bytes of control data. CPID $02 allows the tool to take control of the Exhaust Gas Recirculation (EGR) Valve by making bit 0 of byte 1 a 1, and then placing a value corresponding to the desired duty cycle to the valve in data byte 3. The unused bits are ignored for the purposes of the example in this section.

The example in this section also assumes the following:

- The point-to-point request CAN Id for the PCM is $241.
- The USDT response CAN Id for the PCM is $641.
- The PCM has a device control restriction built in which will not allow the tool to command an idle speed above 1000 rpm if the vehicle is in gear.
- The value of the 2-byte reject code for the rpm limitation while the vehicle is in gear is $0902.

8.20.5.2 DeviceControl (EGR and IAC Control). In this example (Tables 215 thru 217), the tool will send a series of messages to verify the EGR system. The tool will request the PCM to raise the desired idle speed to 1500 rpm and after the rpm has stabilized at the commanded value, the tool will send a second request to open the EGR valve by requesting a 50% duty cycle. The TesterPresent ($3E) messages will not be shown in the example but are assumed to be sent. The message to monitor the engine speed are also not shown in this example but are assumed to be sent on the bus.

Table 215: DeviceControl Example Message Flow to Elevate RPM

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$07	$AE	$01	$00	$03	$00	$00	$78
N(USDT-SF)	$641	$02	$EE	$01	---	---	---	---	---

After the idle speed reaches 1500 rpm and stabilizes the following messages are sent:

Table 216: DeviceControl Example Message Flow to Turn EGR On

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$241	$06	$AE	$02	$01	$00	$80	$00	
N(USDT-SF)	$641	$02	$EE	$02	---	---	---	---	---

At this point the PCM will keep the IAC RPM device control active and command the 50% duty cycle to the EGR valve. The IAC RPM command is kept active because the last device control command received for CPID $01 included the IAC RPM device control and the tester present messages are being sent per the assumptions.

Assume that some time has elapsed and then the operator in the vehicle bumps the gear shift lever resulting in the lever being moved to the overdrive position. The ECU will cancel the IAC RPM and EGR device controls (returning control of those outputs to the PCM) and send the following unsolicited response message to the tool:

Table 217: DeviceControl Example Message Flow of Limit Exceeded

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
N(USDT-SF)	$641	$05	$7F	$AE	$E3	$09	$02	---	---

8.20.6 Node Interface Function.

8.20.6.1 Node Interface Data Dictionary (Table 218).

Table 218: Data Dictionary of DeviceControl Service Pseudo Code

Variable/Meaning	Values
message_data_length **Dev_Cntrl_Active** **security_Access_Unlocked** **TesterPresent_Timer_State** **request_DeviceControl_exit** **Diag_Services_Disable_DTCs** **DTCs_Enabled_In_Device_Control** **Security_Access_Allowed**	Reference Common/Global Pseudo Code Data Dictionary For Definition Of These Flags/Variables
device control status flags Flags associated with each device control function. Indicates whether or not the individual device control function is active.	TRUE/FALSE
Valid_CPID This is a flag used to keep track of whether the requested CPID is supported by the ECU.	YES/NO
Num_Supported_CPIDs This is a constant that is equal to the number of CPIDs supported by the ECU.	Determined at compile time
CPID_Index This is a counter used to index through the array of supported CPIDs when validating the requested CPID.	$00 to (Num_Supported_CPIDs - 1)
CPID_Array[] This is a multi-dimensional array of pointers and data used by the device control application. Data in this array includes the CPID number, whether or not the CPID requires security access, whether or not the CPID requires security code access, the length of the control bytes for a given CPID, and the actual control data bytes. The pseudo code uses the CPID number, security status, and length. These values are accessed in the pseudo code using the following notation:	N/A
CPID_Array[].CPID This variable is the CPID Identifier number	$01 to $FE
CPID_Array[].Security This is a flag used to track whether or not a CPID requires security access before allowing the activation of any of the functions controllable via this CPID.	YES/NO
CPID_Array[].Length This variable defines the expected number of bytes for a given CPID. It is used to compare against the number of control bytes in the request message to validate that the request is correctly formatted.	$00 to $05
CPID_Array[].SecurityCodeAccess This is a flag used to track whether or not a CPID requires security code access before allowing the activation of any of the functions controllable via this CPID.	YES/NO

Variable/Meaning	Values
Secure_CPID This is a flag which is used to keep track of whether or not the requested CPID is one which requires security access.	YES/NO
Valid_CPID_Data_Length This is a flag which is used to keep track of whether or not the number of control bytes in the request message is valid.	YES/NO
Valid_Msg_Length This is a flag used in the initial message length check which verifies that the request is a single frame and includes at least a CPID number.	YES/NO
SecurityCodeAccess_CPID This is a flag which is used to keep track of whether or not the requested CPID is one which requires security code access.	YES / NO

8.20.6.2 Node Interface Pseudo Code.

Powerup States:

Set all the individual device control status flags to FALSE

The following logic is executed each time a Mode $AE message is received by the device:

```
BEGINFUNCTION Serv_AE_Msg_Recvd()
IF ((message_data_length > 7) OR (message_data_length < 2)) THEN
    send Negative Response ($7F $AE $12) /* Invalid Format.*/
    Valid_Msg_Length ← NO
ELSE
    Valid_Msg_Length ← YES
    Valid_CPID ← NO
    Secure_CPID ← NO
    Valid_CPID_Data_Length ← NO
    FOR (CPID_Index ← $00 TO (Num_Supported_CPIDs - $01) BY $01)
        IF (CPID_Array[CPID_Index].CPID = $CPID) THEN
            Valid_CPID ← YES
            IF (CPID_Array[CPID_Index].Security = YES) THEN
                Secure_CPID ← YES
            ENDIF
            IF (CPID_Array[CPID_Index].SecurityCodeAccess = YES) THEN
                SecurityCodeAccess_CPID ← YES
            ENDIF
            IF (CPID_Array[CPID_Index].Length=(message_data_length-$02)) THEN
                Valid_CPID_Data_Length ← YES
            ENDIF
            CPID_Index ← Num_Supported_CPIDs
        ENDIF
    ENDFOR
ENDIF
IF (Valid_Msg_Length = YES) THEN
```

```
    IF ((Valid_CPID = NO) OR
    ((Secure_CPID = YES) AND (Security_Access_Unlocked=FALSE)) OR ((SecurityAccessCode_CPID = YES) AND (Security_Access_Allowed=FALSE))) THEN
        send Negative Response ($7F $AE $31) /* Request Out Of Range. */
    ELSE IF (Valid_CPID_Data_Length = NO)
        send Negative Response ($7F $AE $12) /* Invalid Format. */
        /* Note: there is a flag in mode $27 (DeviceContol_Security_Level) which is used to determine if the assembly plant or service device control restrictions apply */
    ELSE IF (($CPID attempts to control outside device's acceptable range) OR (Device conditions prohibit requested control)) THEN
        send Negative Response ($7F $AE $E3 $xx $yy) /*Device Control Limits Exceeded*/
        Dev_Cntrl_Active ← NO
        Set all the individual device control status flags to FALSE.
    ELSE IF ($CPID = $00) THEN
        Dev_Cntrl_Active ← NO
        Set all the individual device control status flags to FALSE
        Send ($EE $00 )
    ELSE
        TesterPresent_Timer_State ← ACTIVE
        IF (the following logic cannot be completed within $P2_C$ ms.) THEN
            send a Negative Response ($7F $AE $78) /*RequestCorrectlyReceived-ResponsePending */
        ENDIF
        Send ($EE $CPID)
        Dev_Cntrl_Active ← YES
        IF (DTCs_Enabled_In_Device_Control = NO) THEN
            Diag_Services_Disable_DTCs ← TRUE
        ENDIF
        Set the appropriate device control status flags which are used by the
        individual control functions to TRUE.
    ENDIF
ENDIF
ENDFUNCTION
/* The following logic is executed in a background loop and can also be called based on a TesterPresent $3E timeout or upon receipt of a ReturnToNormalMode ($20) service request. */
BEGINFUNCTION Background_DeviceControl_Logic( )
IF ((Dev_Cntrl_Active = YES) AND (request_DeviceControl_exit = NO)) THEN
    /* Note: there is a flag in mode $27 (DeviceContol_Security_Level) which is used to determine if the assembly plant or service device control restrictions apply */
    IF (Device conditions are outside of device's acceptable range) THEN
        send a Negative Response ($7F $AE $E3 $xx $yy) /* Device Control Limit Exceeded */
        Dev_Cntrl_Active ← NO
        set all the individual device control status flags to FALSE
    ELSE
        remain in device control
    ENDIF
ELSE IF ((Dev_Cntrl_Active = YES) AND (request_DeviceControl_exit = YES)) THEN
```

```
    request_DeviceControl_exit ← NO
    Dev_Cntrl_Active ← NO
    set all the individual device control status flags to FALSE
ENDIF
ENDFUNCTION
```

8.20.7 Node Verification Procedure.

Procedure 1:

1. Send a $AE message with less than two data bytes (no CPID in request) and verify the $7F $AE $12 response.
2. Send a multiple frame $AE message and verify $7F $AE $12 response.
3. Send a $AE message with an invalid number of control bytes for a given CPID and verify the $7F $AE $12 response.
4. Send a $AE message with an unsupported CPID and verify $7F $AE $31 response.
5. Send a $AE message with a secure CPID (if applicable) with with the manufacturers enable counter = $00 or vulnerability flag < $FF and security has not been accessed (SecurityAccess ($27) request has not been sent) and verify $7F $AE $31 response.
6. Send a $AE message with a secure CPID (if applicable) when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has not been accessed (SecurityAccess ($27) request has not been sent) and verify $7F $AE $31 response.
7. Send a $AE message with a secure CPID (if applicable) when the manufacturers enable counter > $00 or vulnerability flag = $FF and security has been accessed (SecurityAccess ($27) request has been sent) and verify the appropriate positive response.
8. Send a $AE message with a security code required CPID (if applicable) with the security code (as defined in the Vehicle Theft Deterrent SSTS) has not been entered and verify $7F $AE $31 response.
9. Send a $AE message with a security code required CPID (if applicable) when the security code has been entered and verify the appropriate positive response.

Procedure 2:

1. Send a $AE message with a valid CPID and data and verify the activation of device control. Repeat this step for each device control function supported.

Procedure 3:

1. Send a $AE message with CPID of $00 while device control is active, verify that active device control function(s) are cancelled.
2. Repeat Item 1 for each device control supported.
3. Activate a device control function and then verify that the control function is cancelled after $P3_{Cmax}$ ms without any TesterPresent ($3E) messages sent on the bus.
4. Repeat Item 3 for each device control supported.

Procedure 4:

1. Send a $AE message with a valid CPID and data, cause a device control limit to be exceeded, verify $7F $AE $E3 $xx $yy (limit exceeded) response. Verify that the values of $xx and $yy are correct for the limit which was exceeded.
2. Repeat step 1 for each possible device control restriction.

Procedure 5:

1. Send a valid device control command without the InitiateDiagnosticOperation ($10) service active. Create a fault condition and verify that no DTC is set (via the $A9 service) for the fault condition created.
2. Send the appropriate InitiateDiagnosticOperation ($10) request (if applicable) to allow DTC algorithms to run while DeviceControl is active. Activate a device control and then create a fault condition and verify that the appropriate DTC is logged (via the $A9 service).

Procedure 6: (For Devices that support Device Control Security Access for Restriction Overrides.)

1. Access the device control security level (via $27 service) and repeat procedure 4 for each device control that has a manufacturing restriction that differs from the service restriction and verify that device controls remain active and that no negative response is sent.
2. After step 1 above, send no messages for P3$_C$ and verify that all device controls are cancelled. Then repeat step 1 of procedure 4 (for a single device control) and verify that the device control restrictions are enabled.

Procedure 7: (Additional checks if the ECU supports negative response code $78.)

1. Create conditions under which the ECU should return the $7F $AE $78 response. Send a valid request for this service (including applicable CPID and device control parameters) and verify that the $7F $AE $78 response is received followed by the appropriate final response within the P2$_{C*}$ timing window.
2. Repeat step 1 of this procedure for each possible combination of device control and RC = $78 reason.

8.20.8 Tester Implications. If DTCs are inhibited as a result of this service, a $20 service request message or TesterPresent timeout shall be required to re-enable the DTC logic.

This service should only be requested using physical addressing.

9 ECU Programming Requirements and Process

9.1 This chapter defines the requirements for programming nodes on a GMLAN serial data link using the Service Programming System (SPS) and the utility file concept. The GMLAN programming process is also detailed in this chapter.

The programming requirement sections describe general compliance requirements, minimum diagnostic service implementation requirements, boot memory requirements, and specific vehicle assembly plant performance requirements for programmable ECUs. A section is also included that defines additional requirements for all ECUs (includes non-programmable ECUs). Non-programmable ECUs shall also meet the requirements in this section in order to ensure that they are tolerant of a programming event.

The programming process sections describe the steps needed and the diagnostic services required for those steps. This process was defined in order to create a common utility file for service and assembly plant programming procedures. The programming process described in this chapter provides the possibility for the simultaneous programming of multiple GMLAN ECUs on a single vehicle, independent of subnet location, and within a single programming event. Simultaneous programming of ECUs can be accomplished by having the programming tool process utility files in parallel (which is desirable for vehicle assembly plant usage). This process also allows for sequential programming of multiple ECUs or the programming of an individual ECU.

Note: Although the GMLAN programming procedure supports the simultaneous programming of multiple GMLAN ECUs, the service procedures may only support the programming of an individual ECU.

- ECU Programming General Requirements/Information.

The SPS programming process can be applied to an SPS programmable ECU in order to:

- Reprogram an ECU which has previously been fully programmed.
- Program an ECU which is shipped to the vehicle assembly plant or service facility without some element of its full combination of operational software and calibration data. (See Note Below).

Note: This does not include vehicle option content data written to the device using the WriteDataByIdentifier ($3B) service unless specified in a CTS or supplemental ECU diagnostic specification.

- Reprogram an ECU which has detected an error with memory locations containing software or calibration which forces the ECU to run out of boot software.

SPS programmable ECUs fall into three categories: SPS_TYPE_A, SPS_TYPE_B, or SPS_TYPE_C.

- SPS_TYPE_A:

An SPS_TYPE_A ECU is a programmable ECU that has already been fully programmed (either by the ECU supplier, the vehicle assembly plant, or in the service environment). SPS_TYPE_A ECUs are fully functional and can receive diagnostic requests and respond to them using the appropriate diagnostic CAN Identifiers as outlined in this specification in paragraph 4.4 These CAN Identifiers shall be further referenced as permanent diagnostic CAN Identifiers.

- SPS_TYPE_B/SPS_TYPE_C:

SPS_TYPE_B and SPS_TYPE_C ECUs are programmable ECUs that are missing some element of their full combination of operational software and calibrations, or are executing boot software due to a memory error. An ECU which is missing calibrations (or is missing operational software and calibrations) may, or may not, have permanent diagnostic CAN Identifiers preprogrammed. An SPS programmable ECU which is not fully programmed and is used on a single platform would most likely have its permanent diagnostic CAN Identifiers preprogrammed. An SPS programmable ECU which is not fully programmed and can be used in multiple platforms, may not have the permanent diagnostic CAN Identifiers preprogrammed unless all of the platforms can standardize the CAN Identifiers used by that ECU (or multiple parts are released to accommodate the differences in CAN Identifiers between platforms). An SPS_TYPE_B ECU meets the above criteria and has its permanent diagnostic CAN Identifiers preprogrammed. An SPS_TYPE_C ECU meets the above criteria and does not have its permanent diagnostic CAN Identifiers preprogrammed. SPS_TYPE_B and SPS_TYPE_C ECUs shall not attempt to participate in normal communication message exchange.

Note: An ECU executing boot software due to a memory fault is considered to be SPS_TYPE_B if permanent diagnostic CAN Identifiers are comprehended in the boot software. If the permanent diagnostic CAN Identifiers are not comprehended in boot software, then the ECU is considered SPS_TYPE_C.

If permanent diagnostic CAN Identifiers are preprogrammed, an SPS programmable ECU shall respond to all diagnostic requests which contain one of the permanent diagnostic CAN Identifiers supported by the ECU (SPS_TYPE_A and SPS_TYPE_B). If permanent diagnostic CAN Identifiers are not preprogrammed (SPS_TYPE_C), the ECU shall not respond to diagnostic request messages until diagnostic responses are enabled. While diagnostic responses are disabled, an SPS_TYPE_C ECU shall only receive and process (but not respond to) diagnostic messages addressed to it with the AllNodes request CANId and an extended address of AllNodes. The process of enabling diagnostic responses is described in section 9.1.1 of this specification.

At the conclusion of a programming event, all ECUs on a given subnet shall perform a software reset.

Note: The software reset allows an ECU which was just programmed to begin executing the new operational software and calibration data downloaded. In addition, the reset of all devices synchronizes the start-up of normal communications.

A programming event is considered to have concluded upon receipt of a valid ReturnToNormalMode ($20) request message, or if a TesterPresent timeout occurs. The tester must ensure that the TesterPresent service messages are sent on all GMLAN subnets simultaneously (as close as possible) to ensure that all subnets would exit a programming event at the same time should a TesterPresent timeout occur. The same logic applies to the tester when sending a ReturnToNormalMode message to end a programming event.

9.1.1 Enabling Diagnostic Responses on SPS_TYPE_C ECUs. An SPS_TYPE_C ECU shall not send positive or negative response messages for any diagnostic service until the programming tool enables them. While diagnostic responses are disabled, the ECU shall only process (but not respond to) diagnostic requests sent using the AllNodes request CANId and AllNodes extended address. Diagnostic responses shall become enabled once the ECU receives a DisableNormalCommunication ($28) request followed by a ReportProgrammedState ($A2) request. The SPS_TYPE_C ECU shall only respond to the $A2 request during this sequence, and shall respond to all subsequent diagnostic requests until a TesterPresent timeout occurs or a ReturnToNormalMode ($20) request is received.

Once an SPS_TYPE_C ECU has enabled diagnostic responses, it shall enable two special-case diagnostic CAN Identifiers for programming purposes. The special case CAN Identifiers are defined as:

- SPS_PrimeReq CANId = $0xx, where xx represents the ECU diagnostic address value. This CANId shall be used as the point-to-point (PTP) request CANId.
- SPS_PrimeRsp CANId = $3xx, where xx represents the ECU diagnostic address value. This CANId shall be used as the USDT response CANId.

During the sequence to enable diagnostic responses, the SPS_TYPE_C ECU shall respond to the $A2 request using the SPS_PrimeRsp ($3xx) CANId.

Diagnostic CAN Identifiers and normal mode message CAN Identifiers for a fully programmed ECU are part of the calibration data downloaded to the ECU during a programming event. Upon completion of a software reset, the now completely programmed ECU(s) shall recognize their specific CANId assignments for normal and diagnostic messaging.

9.1.2 Information Contained Within The Utility File. A utility file is used to provide ECU step-by-step programming instructions to a programming tool. The utility file concept was developed to keep the proliferation of tool programming software to a minimum. A utility file is broken into 3 distinct sections; the Header Information section, the Interpreter Instruction section, and a Routine section. The Header Information section contains data that is typically constant during a programming event (e.g., the interpreter type which defines the communications protocol used for programming). The Instruction section contains the Operation Codes (op-codes) used by the programming tool to step through the programming event. The Routine section contains ECU specific programming routines or programming algorithms (e.g., flash erase and write routines).

The Header Information section of the utility file includes data which defines the interpreter type. For GMLAN controllers, the interpreter type indicates that the ECU resides on a GMLAN CAN protocol link (as opposed to a Class 2 or KWP2000 link). The utility file does not contain any information which tells the programming tool which GMLAN subnet the ECU resides on. The programming tool must determine subnet location of programmable ECUs in the steps prior to the execution of a utility file. Refer to paragraph 9.4.1 for further details on how the programming tool determines which subnet the ECU resides on.

The utility file does not contain ECU CANId information. The op-codes within the Interpreter instruction section contain action fields which include the diagnostic address of a GMLAN ECU. Using the diagnostic address within the utility file (instead of the CANId) allows a single utility file to be used on the same ECU across multiple platforms where the CAN Identifiers could not be standardized. The identification of CAN Identifiers assigned to the target ECU(s) must be determined in the steps prior to the execution of a utility file. Refer to paragraph 9.4.1 for further details on how the programming tool correlates the diagnostic address to the appropriate request and response diagnostic CAN Identifiers.

All diagnostic services executed via utility file instructions shall be physically addressed to the ECU to be programmed to allow for parallel programming. See paragraph 9.4 for more details.

9.2 Requirements for All ECUs to Support Programming. The following applies to all ECUs on the link (not limited to the node actively being programmed):

- The ProgrammingMode ($A5) service shall be used by the ECU to recognize the start of a programming event.

1. During a programming event, ECUs shall default their physical input/output (I/O) pins (wherever possible and without risk of damage to the ECU/vehicle and without risk of safety hazards) to a predefined state which minimizes current draw.
2. ECUs shall ensure that they can handle 100% bus utilization at any allowed programming baud rate without dropping frames during the programming event. An ECU may need to modify its hardware acceptance filtering to only receive the AllNodes CANId (non-programmable ECU), or the AllNodes CANId and either the diagnostic point-to-point CANId or SPS_PrimeReq CANId (for programmable ECUs) in order to meet this requirement.

- The DisableNormalCommunication ($28) service shall be used by the ECU to recognize the disabling of normal communication and to ensure that an ECU does not set DTCs while another ECU is being programmed. This includes not only disabling the transmission of normal communication messages, **but also disabling the processing of any received normal communication messages.**

Note: There may be some vehicle link configurations where a mode $10 request must be sent prior to the Mode $28 to ensure no DTCs are set when normal mode is disabled.

- The receipt of a ReturnToNormalMode ($20) request shall be used by the ECU to conclude an active programming event. At the conclusion of a programming event, each ECU shall perform a software reset, and all fully programmed ECUs shall resume normal communications (including re-enabling DTCs). For special issues regarding resets for MSSC devices, see paragraph 4.3.4.
- The TesterPresent ($3E) service shall be used by the ECU to recognize that the active programming event is still in process.
- The ReadDataByIdentifier ($1A) service and the DataIdentifier $B0 (diagnosticAddress) shall be supported by the ECU in order to provide the tester its diagnostic address. The diagnostic address information in conjunction with the ECU CAN Identifiers is used by the tester to recognize the correct response(s) of the ECU(s) connected to the GMLAN subnets during the Pre-Utility File Process. MSSC ECUs shall support this functionality on each GM LAN subnet they are connected to. SPS_TYPE_C ECUs are not required to report the diagnostic address when executing out of the boot software as this information can be obtained from the SPS_PrimeRsp CANId.

Note: An ECU that was SPS_TYPE_C prior to being successfully programmed is required to be capable of reporting its diagnostic address once it is fully programmed and executing operational software.

- The programming tool shall be able to connect to the DLC with the vehicle in any power mode without causing damage to any ECU or components attached to the physical I/O pins of any ECU.
- The sequence of signals during connection to the DLC shall not cause any degradation in performance or damage to the ECU or components attached to the physical I/O pins of any ECU.
- All ECUs shall meet the frame padding requirements as outlined in paragraph 4.6 of this specification.

9.3 Requirements for SPS Programmable ECUs to Support Programming.

9.3.1 Hardware Requirements. All ECUs that are SPS programmable must be able to interface with the programming tools used by development, the assembly plant, and by service via the appropriate GMLAN pins of the vehicle 16-pin DLC specified in GM J1962. The only power required at the DLC for programming shall be vehicle battery power.

Any ECU that is properly installed in the vehicle and is SPS programmable shall be able to be programmed via the DLC. It shall not be required to remove the ECU from the vehicle in order to perform programming.

9.3.1.1 Memory Requirements.

The ECU memory shall be permanently affixed (non-removable) to the circuit board.

There are three different types of memory required that can be categorized into the following retention levels.

Examples of memory that is typically used for each memory type is given in parenthesis. All voltage levels (V_p) are at the ECU connector and apply to the primary power pin.

1. Type 1 memory (typically RAM) contents shall be capable of being retained while **(V_p) is ≥ 9.0 V_{DC}.**

- The ECU shall be capable of reading and writing to Type 1 memory with **9.0 V_{DC} ≤ (V_p) ≤ 16.0 V_{DC}.**
- An ECU must provide enough Type 1 memory to hold the programming routines and provide adequate buffer space to meet the programming timing requirements specified in the General Assembly Programming And Test section of the GM Bill Of Process (BOP).

2. Type 2 memory (typically EEPROM) contents shall be retained at all times even without (V_p) voltage present.

- The ECU shall be capable of reading and writing to Type 2 memory with **9.0 V_{DC} ≤ (V_p) ≤ 16.0 V_{DC}.**
- The number of erase/write cycles shall be documented in a CTS or SSTS. Type 2 memory used solely for programming (i.e., not updated by operational software during normal vehicle operation) requires a minimum of 500 erase/write cycles.

3. Type 3 memory (typically Flash) contents shall be retained at all times even without (V_p) voltage present and it shall be protected from unauthorized erasure or modification.

- The ECU shall be capable of reading Type 3 memory with **9.0 V_{DC} ≤ (V_p) ≤ 16.0 V_{DC}.**
- It shall be possible via the GMLAN bus to erase and program the contents of Type 3 memory of an ECU with **9.0 V_{DC} ≤ (V_p) ≤ 16.0 V_{DC}.**
- The number of programming cycles shall be documented in a CTS or SSTS. Type 3 memory used solely for programming (i.e., not updated by operational software during normal vehicle operation) requires a minimum of 100 erase/write cycles.

Note: The voltage levels for retaining memory contents specified in this section are for use during a programming event. Product performance requirements will most likely result in more stringent requirements.

9.3.2 Software Requirements. An SPS_TYPE_A ECU shall be capable of reporting the base model part number (DIDs $CC/$DC), end model part number (DIDs $CB/$DB), and all supported software module part numbers (DIDs $C1 thru $CA/$D1 thru $DA, plus $DD and any application specific DIDs if more than

10 software/calibration parts exist) via the ReadDataByIdentifier ($1A) service. Support of other DIDs (e.g., $92, $97, $98, $99) is optional and shall be specified in the CTS if needed.

9.3.2.1 Operational Software. If the operational software is programmable via SPS, then the operational software shall be capable of being programmed separately from the calibrations. This allows for assembly plant programming of only calibrations. Deviations from this requirement must be agreed upon by appropriate release responsible engineer, service operations and manufacturing representatives and shall be documented in the CTS.

The operational software shall have its own unique identifier retrievable via the ReadDataByIdentifier ($1A) service.

9.3.2.1.1 Requirements for SPS ECUs Where Operational Software Is Not Programmable. SPS_TYPE_C ECUs executing operational software (only valid if operational software is not programmable via SPS) shall be able to receive diagnostic messages using the AllNodes request CANId with AllNodes extended address. SPS Programmable ECUs with preprogrammed permanent diagnostic CAN Identifiers (SPS_TYPE_A or SPS_TYPE_B) shall also be able to receive messages addressed with its PTP request CANId. SPS_TYPE_C ECUs (which do not have preprogrammed permanent diagnostic CAN Identifiers) shall respond to requests sent with the SPS_PrimeReq ($0xx) CANId once diagnostic responses have been enabled.

An ECU shall be capable of using the same diagnostic CAN Identifiers for the duration of a programming event. This means that an ECU which stores the permanent diagnostic CAN Identifiers in calibration and is fully programmed at the beginning of a programming event (SPS_TYPE_A ECU) shall use the permanent diagnostic CAN Identifiers even after the calibrations have been erased. The operational software is not required to retain the permanent diagnostic CAN Identifiers if the programming event is interrupted prior to its completion and the ECU has performed a software reset.

An ECU that is only capable of calibration programming (e.g., the operational software is part of ROM and calibrations stored in EEPROM) shall have in the operational software the equivalent functionality specified for the boot software in the subsequent sections of this chapter.

9.3.2.2 Calibrations. Calibrations shall be capable of being programmed separately from the operational software. This allows for assembly plant programming of only calibrations. Deviations from this requirement must be agreed upon by appropriate release responsible engineer, service operations and manufacturing representatives and shall be documented in the CTS.

Each calibration module shall have its own unique identifier retrievable via the ReadDataByIdentifier ($1A) service.

The ECU shall support either one or both of the following methods of programming calibrations.

- Programming an individual calibration module.
- Programming multiple (or all) of the calibration modules during a single programming event.

9.3.2.3 Boot Software Description and Requirements. All SPS programmable ECUs that support programming of the operational software shall contain boot software in a boot memory region. ECUs that support boot software shall continue to execute out of the boot until a complete set of Operational software and calibrations are programmed.

The boot memory shall be protected against inadvertent erasure such that a failed attempt to modify program calibrations or operational software does not prohibit the ability of the ECU to recover and be programmed after the failed attempt. The ECU shall be able to recover and be reprogrammed if any of the following error conditions occur during the programming process.

1. Loss of supplied power connection.
2. Loss of the ground connection.
3. Disruption of GMLAN communication.
4. Over or under voltage conditions.

Boot software resides in the boot memory region and is the software that an ECU begins executing upon powerup. Transfer of program control to the boot software also occurs once the ECU is informed that it is about to be programmed (refer to the $34 service). All SPS Programmable ECUs operating out of Boot memory shall not transmit any normal communication messages or any unsolicited diagnostic messages.

The boot software can be broken down into the following components: Power Management, System Initialization, Message Handler, Programming Executive/Program Loader, Diagnostic Service Manager, and Programming Routines. See Figure 36.

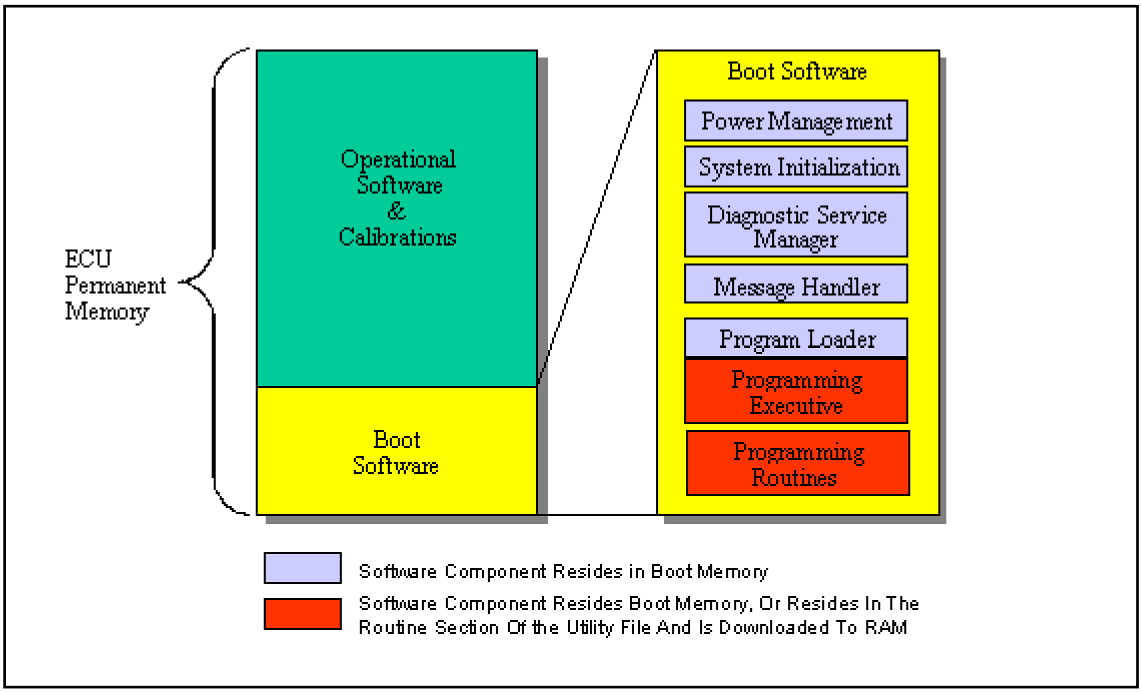


Figure 36: GMLAN Boot Software

9.3.2.3.1 Boot Software Power Management. The power management portion of the boot software handles the wake-up mechanism supported by the ECU (if applicable). The wake-up mechanism for SWCAN ECUs is defined in GMW3104. Dual Wire CAN (DWCAN) ECUs that support a wake-up mechanism must document the mechanism used in the CTS.

The power management component of the boot software also interfaces with the programming executive. If the programming executive does not detect the start of a programming event within 1 minute of the receipt of a wake-up, then the programming executive shall inform the power management logic which in turn shall cause the ECU to enter the low power state.

The start of a programming event is defined to be the point where the programming_mode_active flag is set to YES. Refer to paragraph 8.17.6.2 (service $A5 pseudo code) for additional details. Once the programming event has started, the ECU shall remain awake until the programming event concludes (via the receipt of the $20 service or a $P3_C$ timeout). The ECU shall perform a software reset at the conclusion of the programming event.

9.3.2.3.2 Boot Software System Initialization. The system initialization component of the boot software is responsible for ECU initialization upon powerup or after a reset. Tasks of this component include (but are not limited to) I/O initialization and determining if valid operational software and calibrations are present in the ECU. The system initialization software shall be able to determine if the ECU has only boot software present, boot software and operational software present, or if boot software, operational software and calibrations are all present. The exact method used to determine if valid operational software and calibrations are present must be documented in a CTS, GM provided bootloader specification or ECU specific diagnostic implementation specification. If valid operational software and calibrations are present, then the system initialization component completes its initialization and transfers control to the operational software. Otherwise, the ECU transfers program control to the boot software programming executive or program loader.

If valid operational software and calibrations are not present, then the initialization component of the boot software shall initialize the ECU I/O in a manner that meets the requirements specified in paragraph 9.2.

9.3.2.3.3 Boot Software Message Handler. The boot software message handler is responsible for transferring information between the programming tool and the boot software programming executive. Included in this component of the boot software are the CAN driver, the CAN hardware message filtering logic, support of the Network Transport Layer, and high speed support for SWCAN ECUs.

The Network Layer supported in the ECU boot software shall maximize the size of the network layer buffer and minimize the size of the network layer parameter ST_{min} to maximize the efficiency of data block transfers. The ECU shall allocate some or all of its RAM resources used for normal operation for the network layer buffer when operating out of the boot code. The value of ST_{min} during normal operation can be different than the value used during programming.

The CAN hardware message filtering logic in the boot software shall meet the requirements specified in paragraph 9.2 - Requirements for All ECUs to Support Programming.

9.3.2.3.4 Boot Software Programming Executive and Program Loader. The boot software programming executive coordinates the sequence of events during a programming session. The programming executive has knowledge of the memory map including the allowed application program area, RAM location, and logic for handling flash sectors (if applicable).

The programming executive keeps track of the various software and calibration modules, their starting address, length, and checksum. The executive interfaces with the boot software diagnostic service manager and the boot software message handler to verify incoming data from the programming tool and to provide responses to the programming tool. The executive handles transferring the programming routines to RAM and interfaces with the programming routines to transfer the software and calibration modules to permanent memory.

During the download of a data file (operational software or calibration data) the programming executive interfaces with the programming routines to verify the successful erase/write performed on a section of memory, The programming executive shall also perform a verification of successful programming at the conclusion of each downloaded software or calibration module (data file) by calculating a running checksum or CRC as a module is being downloaded. When the last TransferData request of a module has been processed (confirmation from the programming routine that the data was successfully written to permanent memory), the programming executive shall compare the checksum or CRC against the value in the product memory file header (refer to paragraph 9.3.3.1). If the calculated value does not match the value in the product memory file header, a negative response shall be sent indicating a programming fault (Negative response code $85 General Programming Failure). The method used to calculate the checksum or CRC of the data files shall be specified in a CTS, SSTS, supplemental diagnostic specification, or bootloader specification referenced by either the CTS or SSTS.

The programming executive is also responsible for performing compatibility checks (if required) to ensure that the operational software will work with the bootloader and that each calibration data file is compatible with the operational software that is programmed in the ECU. Performing compatibility checks prevents the possibility of continuous running resets due to a mismatch in the data files (e.g., calibration file size changes due to software change and programming done with new software module and old calibration file) which can leave the part unrecoverable. The programming executive performs the compatibility checks using the DCID values as defined in paragraph 9.3.3.1. For an operational software file the DCID in the data file header is compared against a constant stored in the bootloader. If the two compatibility values do not match, the data file is not downloaded to permanent memory. For calibration data files the DCID is compared against a corresponding value in the operational software header (e.g., DCID_am1) and if the two do not match the data file is not downloaded to permanent memory. Any compatibility check failure shall result in a negative response message with response code $85 General Programming Failure. The appropriate CTS, SSTS, supplemental diagnostic specification, or bootloader specification referenced by either the CTS or SSTS shall document whether or not compatibility checks are required for a particular ECU and any additional requirements relative to their values or implementation.

When the programming executive detects the end of the programming event (e.g., a service $20 request is received or a $P3_C$ timeout occurs), it shall inform the message handler to reset back to the normal baud rate (if applicable for SWCAN) and trigger a software reset. The executive also interfaces with the power management component of the boot software to allow an ECU to run out of the boot software to enter a low power state if a programming event has not begun within 1 minute of a wake-up or power being applied to the ECU.

The programming executive may reside in boot memory or it may be contained within the routine section of the ECU utility file, downloaded into RAM, and then executed. The advantage of placing the programming executive in RAM is that it allows the boot to be smaller and it also allows updates to the programming executive without requiring a new hardware part. The disadvantage to this approach is a slight increase in programming time in order to transfer the programming executive to RAM via the serial data bus and begin its execution. If the programming executive is located in the utility file then a portion of the programming executive (called the program loader) remains in the boot memory. The program loader component gets program control from the boot software initialization component if some portion of the software or calibration data is not present in the ECU. The program loader interfaces with the power management component to allow an ECU to enter a low power state if no programming session has been established within 1 minute of a wake-up. The program loader also interfaces with the boot software diagnostic service manager and the boot software message handler in order to receive the download of the programming executive, and then transfer program control to the programming executive. If the programming executive resides in boot memory then the program loader and programming executive are considered as one component. See Figure 36.

9.3.2.3.5 Boot Software Diagnostic Service Manager. The boot software diagnostic service manager contains the logic for handling diagnostic requests and responses during programming. The level of support of diagnostic services within the boot shall be minimized in order to keep the size of the boot software as small as possible. Refer to paragraph 9.3.2.4 for specific requirements on the implementation of diagnostic services within the boot software.

The diagnostic service manager interfaces with the boot software programming executive which controls the sequence of events during programming.

9.3.2.3.6 Boot Software Programming Routines. The boot software programming routines are those hardware specific routines needed to program the ECU permanent memory. Examples include memory erase, memory write, routines to turn programming voltage on or off, and checksum calculation routines. These routines provide pass or fail indications back to the boot software programming executive so that it can make a decision whether or not to continue the programming event.

At the end of writing/erasing a section of memory, the memory that has been erased/written to shall be checked to ensure that it was done successfully. The success or failure of the erase/write operation(s) shall be communicated back to the programming executive. The programming executive shall then interface with the diagnostic service manager and the message handler to communicate the result to the programming tool in the form of a TransferData ($36) positive response message or through a negative response ($7F) message including the appropriate response code (other than response code $78).

Although the programming routines can technically either reside in the boot software or in the routine section of the programming Utility File, it is recommended that these routines be stored in the Utility File to minimize the size of the boot software.

If the programming routines are stored in the Utility File and more than one supplier of permanent memory exists for that type of programmable ECU, then the Utility file shall contain all of the programming routines for all types of permanent memory allowed.

9.3.2.3.7 Boot Software General Requirements. All ECUs operating out of boot memory shall be able to receive diagnostic messages using the AllNodes request CANId with AllNodes extended address. SPS Programmable ECUs with preprogrammed permanent diagnostic CAN Identifiers (SPS_TYPE_A or SPS_TYPE_B) shall also be able to receive messages addressed with its PTP request CANId. SPS_TYPE_C ECUs (which do not have preprogrammed permanent diagnostic CAN Identifiers) shall respond to requests sent with the SPS_PrimeReq ($0xx) CANId once diagnostic responses have been enabled.

An ECU shall be capable of using the same diagnostic CAN Identifiers for the duration of a programming event. This means that an ECU which is fully programmed at the beginning of a programming event (SPS_TYPE_A ECU), shall use its permanent diagnostic CAN Identifiers during the programming event even if the boot code only contains the SPS_Prime CANIds. To accomplish this, the permanent diagnostic CAN Identifiers must be provided to the boot software from the normal operational software when program control is transferred back to the boot software. The boot software is not required to retain the permanent diagnostic CAN Identifiers passed from the operational software if the programming event is interrupted prior to its completion and the ECU has performed a software reset (this is valid if the boot software only supports the SPS_Prime CAN Identifiers).

The boot software shall be protected. The boot software can be protected via hardware (e.g., via settings in a control register which prevents certain sectors of the memory from being erased or written to) or software (e.g., address range restrictions in the programming routines). It is recommended that the boot software is not capable of being modified by the same programming erase/write routines that are used to modify the operational software and calibrations. Programming the boot software as part of the SPS programming process may be allowed provided that a mechanism is in place to ensure that there is **no** possibility that the ECU could fail at a point in the programming process where it cannot recover and be programmed with a subsequent programming event. The ECU supplier and the appropriate GM Engineer(s) (Release, Service, and Manufacturing) must review the mechanism implemented and agree that it meets the requirements if the boot is to be programmed as part of the SPS process. If the boot software can be reprogrammed as part of the SPS process, then the boot software should be a unique software module with a unique part number.

Changes to the boot software in a production ECU shall result in a change to the ECU hardware part number (DID $CC) in a module where the boot software is not reprogrammable via the SPS system (e.g., the boot may be capable of being modified at the supplier). However, changes to the boot software in a production ECU shall not result in a change to the ECU hardware part number (DID $CC) in a module where the boot software is reprogrammable via the SPS system. The part number of the software module containing the boot software would reflect the change to the boot software.

9.3.2.4 Boot Software Diagnostic Service Requirements. The following are GMLAN Enhanced Diagnostic Service requirements for the boot software diagnostic service manager of a GMLAN SPS programmable ECU. An ECU that is programmed with operational software and calibrations must also meet the requirements listed in its CTS and/or SSTS. See the appropriate section of this specification for complete definition of each diagnostic service.

Note: Additional diagnostic services may be implemented in an unprogrammed or partially programmed ECU. These additional services must be defined in the ECU CTS.

In each of the diagnostic services described below, only the minimum required sub-function levels and responses are listed.

9.3.2.4.1 Service $10 - InitiateDiagnosticOperation. This service is only required for a gateway ECU connected to a GMLAN subnet which utilizes a wake-up mechanism that is not available to the tool (e.g., wake-up wire not brought out to the DLC). In this case, the gateway ECU shall support the sub-function parameter $04 (wakeUpLinks).

9.3.2.4.1.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot_10_Msg_Recvd()
IF ($Level = LEV_WUPLNK ) THEN
    IF (more than P2_C is needed to process this request) THEN
        send Negative Response ($7F $10 $78) /* request correctly received - response pending */
    ENDIF
    generate wake-up on all GMLAN links where wake-up mechanism is supported
    Send ($50) positive response message
ENDIF
ENDFUNCTION
```

9.3.2.4.2 Service $1A - ReadDatabyIdentifier. This service shall be supported so a programming tool can determine the correct file(s) needed for programming.

The following DIDs shall be supported when executing out of the Boot Software:

- DID $B0 for determining the ECU Diagnostic Address (except for SPS_TYPE_C ECUs).
- DIDs as required for determining the ECU Base Model part number. Minimum DIDs $CC and $DC. Support of DID $92 is optional and shall be specified in the CTS (or SSTS or supplemental diagnostic specification referenced by the CTS or SSTS) if needed.
- The boot software shall be capable of reporting the Software Module Identifier (SWMI) and the Software Module Identifier Alpha Code (SWMIAC) DIDs associated with operational software module(s) once the operational software is programmed into the ECU.

- If the utility file is designed to allow individual calibration modules to be programmed, then the boot software shall be capable of reporting the SWMI and SWMIAC for each calibration module once it has been programmed into the ECU.
- Support of Traceability data (DID $B4) and Broadcast Code (DID $B5) may also be required based on regional practices. If any/all of these DIDs are required, they shall be specified in the CTS (or SSTS or supplemental diagnostic specification referenced by the CTS or SSTS).
- The boot software shall be capable of reporting the Software End Model part number (DID $CB).

9.3.2.4.2.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGIN FUNCTION Boot_1A_Msg_Recvd()
IF ($dataIdentifier is NOT supported) THEN
    send Negative Response ($7F $1A $31) /* Request Out Of Range */
ELSE
    IF (device cannot respond with block contents within P2C ms) THEN
        send Negative Response ($7F $1A $78) /* RequestCorrectlyReceived-ResponsePending */
    ENDIF
    /*Get data values from memory address associated with the $dataIdentifier*/
    send ($5A $dataIdentifier $Data) /* send response message with DID data */
ENDIF
ENDFUNCTION
```

Note: The above pseudo code assumes that the ECU is unlocked when operating out of the boot software.

9.3.2.4.3 Service $20 - ReturnToNormalMode. This mode is required to conclude a programming event. Receipt of a mode $20 (during a programming event) requires the ECU to perform a software reset. If a high speed programming event was enabled on the low speed SWCAN link when a request for this service is received, then all ECUs (including the tester) shall re-initialize their protocol converter hardware. The low speed ECUs shall perform the software reset after re-initializing the protocol converter hardware.

There are no responses allowed to this service during a programming event.

9.3.2.4.3.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot_20_Msg_Recvd()
CALL Boot_Exit_Diagnostic_Services() /* procedure defined below to gracefully exit all active diagnostic
services */
ENDFUNCTION
```

Each time a valid $20 message is received, or if a TesterPresent timeout occurs, the following function is called:

```
BEGINFUNCTION Boot_Exit_Diagnostic_Services()
Service_28_MsgReceived ← NO
TesterPresent_Timer_State ← INACTIVE
TesterPresent_Timer ← 0
IF (permanent diagnostic CAN Identifiers are not programmed) THEN /* For SPS_TYPE_C ECUs */
    diagnostic_responses_enabled ← NO
ENDIF
IF (programming_mode_active = YES) THEN
    TransferData_Allowed ← NO
    programming_mode_active ← NO
    /***************************************************************************
    * Reset CAN Protocol Device if high speed mode was active before doing
    * S/W Reset
    ************************************************************************/
    IF (high_speed_mode_active = YES)
        high_speed_mode_active ← NO
        CALL hnd_Init_CAN_Device() /* handler call to reinit the baud rate in order to prevent bus errors which
        could be caused if reset times differ greatly*/
    ENDIF
    CALL Invoke_Sw_Reset() /* causes a software reset to occur */
ENDIF
ENDFUNCTION
```

Note: The above pseudo code assumes that the ECU is unlocked when operating out of the boot software.

9.3.2.4.4 Service $27 - Security Access. This service shall be supported in the boot software if the ECU is theft, emission, safety related, or if the ECU contains functionality which requires the use of this service to meet legal requirements. If implemented, the minimum sub-function levels required are:

1. Level $01 – SPSrequestSeed.
2. Level $02 - SPSsendKey - This sub-function is only required for ECUs that require this service and are sent to the field as an SPS_TYPE_B or SPS_TYPE_C ECU.

Note: An ECU that is sent to the field with operational software and a dummy calibration (fully programmed but not completely functional) is considered to be an SPS_TYPE_A ECU. In this case it is assumed that the dummy calibration would activate the security feature and it would not be necessary to support the Level $02 in the boot software.

9.3.2.4.4.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot_27_Msg_Recvd()
IF ($Level = $01) THEN
    Send ($67 $01 $00 $00) message
ENDIF
ENDFUNCTION
```

Note: The above pseudo code assumes that the ECU is unlocked when operating out of the boot software.

9.3.2.4.5 Service $28 - Disable Normal Message Transmission. This mode is required to be supported to allow a common utility file for Service and Manufacturing, and to assure no conflicts between programming messages and normal communication messages.

- An SPS_TYPE_C ECU which only has knowledge of its SPS_Prime CAN Identifiers does not respond unless the SPS_Prime CAN Identifiers have already been enabled. Refer to paragraph 9.1.1 - Enabling Diagnostic Responses on SPS_TYPE_C ECUs for more detail.

9.3.2.4.5.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot _28_Msg_Recvd()
Service_28_MsgReceived ← YES
TesterPresent_Timer_State ← ACTIVE
IF (diagnostic_responses_enabled = YES) THEN
    Send ($68) response message
ENDIF
ENDFUNCTION
```

9.3.2.4.6 Service $34 - Request to Download Data Block. Receipt of a valid request for this service causes an SPS_TYPE_A ECU to transfer program control back to the boot software. Transferring control back to the boot software allows the ECU to allocate the necessary resources to support being reprogrammed. Examples of resource allocation include a larger buffer for the Network layer and use of a smaller STmin value to optimize data transfer. The ECU shall support in the boot software the same dataFormatIdentifier values as supported by the operational software.

9.3.2.4.6.1 Pseudo Code (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot_34_Msg_Recvd()
IF ( (dataFormatIdentifier is invalid) OR ((dataFormatIdentifier != $00) AND (unCompressedMemory size is invalid)) ) THEN
    send Negative Response($7F $34 $12) /* Invalid Format */
ELSE
    IF ( (programming_mode_active = NO) OR (a software or calibration module download is in progress)) THEN
        send Negative Response ($7F $34 $22) /* for conditionsNotCorrect */
    ELSE
        IF (more than P2C ms is needed to process this request) THEN
            send Negative Response ($7F $34 $78) /* RequestCorrectlyReceived-ResponsePending */
        ENDIF
        /* at this time the module shall do whatever is necessary to prepare for a DataTransfer service request. */
        IF (DTC for Flash or EEPROM checksum failure is set) THEN
            send Negative Response ($7F $74 $99) /*ReadyForDownload-DTCStored */
        ELSE
            send ($74) response message
        TransferData_Allowed ← YES
    ENDIF
ENDIF
ENDFUNCTION
```

Note: The $34 pseudo code assumes that the ECU is unlocked when operating out of the boot software.

9.3.2.4.7 Service $36 - TransferData. This service is used to transfer programming routines as well as software and calibration data to the ECU during a programming event. This service shall be fully implemented as described in paragraph 8.13.

9.3.2.4.8 Service $3B - WriteDataByIdentifier. This service is required to be supported in boot if it is necessary to write information (e.g., option content) into an ECU before the software reset at the end of the programming event. If implemented in the boot software, then the boot implementation shall be as described in paragraph 8.14.

9.3.2.4.9 Service $3E - TesterPresent. The ECU shall support the receipt of the $3E message in order to maintain the programming event. No positive or negative responses are required.

During a programming event, an ECU executing out of RAM while erasing or writing to type 3 memory, shall reset and suspend its $P3_C$ timer if it is not able to process incoming TesterPresent requests during the erase/write process. Upon completion of the erase/write process, the $P3_C$ timer shall be enabled and synchronized with the other ECUs upon receipt of a TesterPresent request.

Note: Reference the Application Timing Parameters Definition section of this specification for detailed requirements regarding the $P3_C$ timer.

9.3.2.4.9.1 Pseudo Code (Refer to the data dictionary of the application implementation of this service for variable definitions.)

```
BEGINFUNCTION Boot_3E_Msg_Recvd()
TesterPresent_Timer ← 0 /*Reset the tester present timer*/
ENDFUNCTION
```

The following logic is used to implement the TesterPresent_Timer in the node’s main processing loop:

```
BEGINFUNCTION Boot_3E_Background_Logic()
IF (TesterPresent_Timer_State = ACTIVE) THEN
    increment TesterPresent_Timer by the length of the main processing loop
    IF (TesterPresent_Timer ≥ $P3_C$) THEN
        Call Boot_Exit_Diagnostic_Services() /* function in service $20 */
    ENDIF
ENDIF
ENDFUNCTION
```

9.3.2.4.10 Service $A2 - ReportProgrammedState. The ReportProgrammedState is used by the tester to determine which nodes on the link are programmable, and the current programmed state of each programmable node. This service is also used as part of the sequence to enable the SPS_Prime CAN Identifiers for SPS_TYPE_C ECUs.

While operating in boot, the ECU shall evaluate the Presence Pattern at the time of the Service $A2 request to report the current programmed state of the ECU.

9.3.2.4.10.1 Pseudo Code. (Refer to the data dictionary of the application implementation of this service for variable definitions.)

The following logic is executed when a device powers up or reboots after a software reset:

```
Check memory for errors
Check for operational software and calibrations
Store result of memory, software and calibration checks in $programmedState
IF (permanent diagnostic CAN Identifiers are programmed) THEN
    diagnostic_responses_enabled ← YES
ELSE
    diagnostic_responses_enabled ← NO
ENDIF
```

Note: If the program operation is transferred from the operational software to the boot software when the service $34 is received, the permanent diagnostic CAN Identifiers must be made available to the boot software. See requirement in paragraph 9.3.2.3.7. In this case, the flag diagnostic_responses_enabled shall be set to YES for the purposes of the pseudo code.

```
BEGINFUNCTION Boot_A2_Msg_Recvd()
diagnostic_responses_enabled ← YES
IF (calculation for programmedState is not complete)
    send negative response ($7F $A2 $78 ..)
ENDIF
IF (memory fault exists)
    send a ($E2 $programmedState) /* memory faults shall be reported first */
ELSE IF (only calibration data is missing)
    send ($E2 $02)
ELSE IF (software and calibrations are missing)
    send ($E2 $01)
ELSE
    send ($E2 $00) /* this can occur if a request is sent to determine the programmed state after the ECU is
    programmed but before the software reset occurs concluding part 1 of the utility file */
ENDIF
ENDFUNCTION
```

9.3.2.4.11 Service $A5 - ProgrammingMode. This mode allows a tester to verify that conditions are correct for a programming event and to enable the programming mode in order to begin the programming event. This mode also allows the tester to request high speed programming mode to be enabled on the GMLAN low speed (SWCAN) link.

This service shall be fully implemented as described in paragraph 8.17.

9.3.2.4.12 Boot Software Receive Message Handler. The following pseudo code defines the receive message handling logic within the boot software:

```
BEGINFUNCTION Boot_Process_Recv_Msg()
IF (diagnostic_responses_enabled = NO)
    IF ((Service_28_MsgReceived = YES) AND ($Service_Id = $A2) THEN
        CALL Boot_A2_Msg_Recvd()
    ENDIF
    IF ($Service_Id = $28) THEN
        CALL Boot_28_Msg_Recvd()
    ELSEIF ($Service_Id = $3E) THEN
        CALL Boot_3E_Msg_Recvd()
    ENDIF
ELSE
    SELECT FIRST
    WHEN ($Service_Id = $10)
        CALL Boot_10_Msg_Recvd()
    WHEN ($Service_Id = $1A)
        CALL Boot _1A_Msg_Recvd()
    WHEN ($Service_Id = $20)
        CALL Boot _20_Msg_Recvd()
    WHEN ($Service_Id = $27)
        CALL Boot_27_Msg_Recvd()
    WHEN ($Service_Id = $28)
        CALL Boot_28_Msg_Recvd()
    WHEN ($Service_Id = $34)
        CALL Boot_34_Msg_Recvd()
    WHEN ($Service_Id = $36)
        CALL Boot_36_Msg_Recvd()
    WHEN ($Service_Id = $3B)
        CALL Boot_3B_Msg_Recvd()
    WHEN ($Service_Id = $3E)
        CALL Boot_3E_Msg_Recvd()
    WHEN ($Service_Id = $A2)
        CALL Boot_A2_Msg_Recvd()
    WHEN ($Service_Id = $A5)
        CALL Boot_A5_Msg_Recvd()
    OTHERWISE
        IF (message_address_type = PHYSICAL) THEN
            Send Negative Response ($7F $Service_Id $11)
        ENDIF
    ENDSELECT
ENDIF
ENDFUNCTION
```

Note: Services $10, $27, and $3B are optional and may not be present based on the type of ECU.

Note: The pseudo code flag Service_28_MsgReceived is used by the Boot_Process_Recv_Msg( ) function, the Boot _28_Msg_Recvd( ) function, and the Boot_Exit_Diagnostic_Services( ) function to illustrate enabling of the special case diagnostic CAN Identifiers used by SPS_TYPE_C ECUs.

9.3.2.5 Security Requirements. All programmable ECUs that have emission, safety, or theft related features shall employ a seed and key security feature, accessible via the SecurityAccess ($27) service, to protect the programmed ECU from inadvertent erasure and unauthorized programming from the GMLAN interface. All field service replacement ECUs that meet the above criteria, shall be shipped to the field with the security feature activated (i.e., a programming tool cannot gain access to the ECU without first gaining access through the SecurityAccess service).

All ECUs that support security for SPS shall implement the seed and key relationship supplied by GM Service and Parts Operations. Once a security algorithm is assigned, a change to the algorithm shall result in a change to the ECU Base Model Part Number. See DID $CC in Appendix C of this specification.

Note: A change to the security algorithm shall also result in a change to the utility file needed to program the ECU.

All development ECUs shall use the ones-complement of the seed as the valid key.

9.3.2.6 The Vulnerability Flag And Manufacturers Enable Counter (MEC). The Manufacturers Enable Counter (MEC) and Vulnerability Flag are variables which are used to bypass a nodes security system during development or during portions of the node and vehicle manufacturing process. The variables manufacturers_enable_counter and vulnerability_flag are used in the pseudo code of the applicable diagnostic services described within this specification to represent the MEC and Vulnerability Flag respectively. Reference the SecurityAccess ($27) service within this specification for more information on accessing node security.

9.3.2.6.1 The Vulnerability Flag. The Vulnerability Flag is a single byte in the node’s calibration area. The node shall remain unlocked as long as the Vulnerability Flag equals $FF. Any value other than $FF will cause the node to lock if this is the only security system bypass mechanism (see section below on the Manufacturers Enable Counter). This use of the Vulnerability Flag is optional but it is shown in the pseudo code of each of the applicable diagnostic services within this specification. Nodes which do not implement this variable can follow the path in the pseudo code as if the value were other than $FF. The Vulnerability Flag shall only be used to bypass a node’s security system during the development process. All fully programmed (SPS_TYPE_A) production ECUs that implement the Vulnerability Flag shall have its value set to something other than $FF to ensure that the security system is enabled once the MEC is set to $00.

9.3.2.6.2 The Manufacturers Enable Counter (MEC). The MEC shall be supported during ECU development and production by all nodes which implement the SecurityAccess ($27) service. The MEC is a single byte in permanent (EEPROM or equivalent) memory which allows a node to remain unlocked as long as its value is not $00. When the value of the MEC becomes $00, security shall be enabled (provided that the Vulnerability Flag is not $FF) and the SecurityAccess ($27) service shall be required to access security.

Production ECUs shipped to the vehicle assembly plants shall have the value of the MEC initialized by the ECU supplier to a value specified in a CTS, SSTS, or supplemental diagnostic specification referenced by a CTS or SSTS. Service Replacement ECUs shall be shipped to the dealerships with the MEC already set to $00.

The MEC must be programmed to $00 at some point in the vehicle assembly plant process (typically, the MEC is programmed to $00 at the conclusion of a passed Dynamic Vehicle Test).

Note: The MEC is programmed with the WriteDataByIdentifier ($3B) service. Refer to Appendix C to determine the Data Identifier (DID) number for the MEC.

A node shall not allow the value of the MEC to change once it becomes $00, unless SecurityAccess ($27) is successfully initiated. The ability to allow writing a new value to the MEC for any specific ECU which supports the MEC shall be negotiated by representatives from Service and Assembly Verification, and the responsible DRE, and documented in the CTS. The platforms should employ a backup mechanism to ensure that a node will lock itself in the event that the vehicle somehow manages to make its way out of the assembly plant with one or more nodes unlocked.

9.3.2.6.3 Example Implementation of the Manufacturers Enable Counter (MEC). In this example, the MEC is used as a counter which decrements by 1 each time the ignition is cycled. The MEC is initialized to a calibratable value ($FE for this example) by the supplier before the node is shipped to the vehicle assembly plant. In this scenario, any node which was not locked prior to the vehicle leaving the assembly plant would become locked after a calculated number of ignition cycles (254 - the number of ignition cycles which took place at the vehicle assembly plant).

If all nodes on the vehicle which support a MEC also have an ignition input, then each node would decrement the MEC by 1 each time its ignition input transitions from a high voltage state to a low voltage state (e.g., Run to Off transition). If the vehicle has one or more nodes which support a MEC, do not have an ignition input, and can remote off, then a more complex method of decrementing the MEC is required.

Note: It would not be desirable to decrement the MEC each time the node remotes off as this approach greatly increases the possibility that the node would lock itself before all assembly plant programming and testing processes have been completed.

To solve the issue of decrementing the MEC for the nodes which have no ignition input and remote off, a MEC master is used. The MEC master is a node which does have an ignition input and is capable of communication after the ignition input goes to a low voltage state. The MEC master detects the ignition transition and generates a bus wake-up. The master then transmits a message which contains the value of the MEC. (See Note below.) The nodes which were remoted off would wake-up and then synchronize their MEC with the value provided by the MEC master. When synchronizing the MEC, nodes which already have their MEC at $00 would not modify the value. This prevents the possibility of replacing a single node on the vehicle (the MEC master) and unlocking all nodes on the vehicle.

Note: This message may need to span subnets based on the vehicle configuration.

9.3.3 Product Memory (Operational Software and Calibration) File Requirements. Product memory files released to the assembly plant or service shall be in binary format.

All files (e.g., software, calibration, utility file, drawing files, archives, etc.) distributed to GMNA assembly plants or GMNA service facilities shall meet all of the requirements specified in the GM World Wide Software and Calibration Parts Application Program Interface specification. In addition, all binary files (e.g., software, calibration, utility file) and their associated drawing files shall have the file base name the same as the part number released through the Product Description System (PDS).

The Module ID for the Operational Software data file shall be $01.

Note: If the ECU has more than one microprocessor, then Module Id $01 shall be assigned to the operational software module of the processor that handles serial data for reprogramming.

The ECU software and calibration files shall have a header as defined in paragraph 9.3.3.1. The header information is transferred to the appropriate ECU during the programming event. Portions of the header information are optionally programmed into the ECU permanent memory.

9.3.3.1 Software and Calibration File Header Requirements (Table 219).

Table 219: ECU Software and Calibration Module Header Definition

Offset	Number of Bytes	Name	Description	Cvt
$00	2	CS	Module **c**heck**s**um or CRC (hex)	M
$02	2	MID	**M**odule **ID** (hex)	M
$04	1 to 2	HFI	This field contains a **H**eader **F**ormat **I**dentifier (HFI). The HFI is used to indicate the length of the Software Module Identifier (SWMI) field, the length of the Design Level Suffix field, and whether or not the Product Memory Address and Module Length fields are included in the header. Refer to **Table 220 and Table 221** for the detailed definition of the HFI.	M

Offset	Number of Bytes	Name	Description	Cvt
$05	4 to 16	SWMI	This is a variable-length field that contains the **SWMI** (refer to DID $C1 thru $CA definitions in Appendix C) data reported via the $1A service. The Header Format Identifier field contains length information for this field of the header.	M
End of SWMI + 1	2 to 3	DLS	This is a variable length field that contains the modules **D**esign **L**evel **S**uffix (**DLS**) or Alpha Code (ASCII)	M
End of previous field + 1	2	DCID_	This optional field is present if the H_N$_{Size}$ bit of the HFI is 1, and the DCID bit in the HFI field is also a 1. This field is two bytes in length and contains a Data Compatibility identifier value. For an operational software module header this value is compared against a constant in the bootloader to ensure that the software being downloaded is compatible with the boot software in the ECU. For a calibration file header this value is compare by the programming executive against a corresponding value in the operational software header to ensure that there is no mismatch between the calibration file and the operational software.	U
End of previous field + 1	1 to 2	NOAR	This optional field is present if the **P**roduct **M**emory **A**ddress (PMA) bit of the HFI is set to a one. If present, this field is one byte in length if the H_N$_{Size}$ bit of the HFI is 0, and two bytes in length if the H_Nsize bit of the HFI is 1. This field contains the **N**umber **O**f **A**ddress **R**egions (**NOAR**) within the software/calibration module. This field is used to accommodate programming of a software or calibration module that has data that is broken up into separate memory regions in the ECU. The number in this field represents the number of different address regions addressed when the module is downloaded.	U
End of previous field + 1	4	PMA#1	This optional field contains the starting PMA of the first (or only) Address Region where the module is loaded. This is an optional field which is present only if the HFI indicates that PMA is used. If the product uses less than four bytes for addressing, then the most significant byte(s) shall be set to $00.	U
End of previous field + 1	4	NOB#1	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address from the previous field.	U
End of previous field + 1	4	PMA#2	This optional field is present if the PMA bit of the HFI is set to a one and the NOAR field indicates that more than one address region is contained within the module. If present, this field contains the starting address of the 2nd memory region.	U
End of previous field + 1	4	NOB#2	This optional field is present if the PMA bit of the HFI is set to a one and the NOAR field indicates that more than one address region is contained within the module. If present this field contains the **n**umber **o**f **b**ytes to be sent starting with the address of the 2nd memory region.	U
:	:		:	U

Offset	Number of Bytes	Name	Description	Cvt
End of previous field + 1	4	PMA#n	This optional field is present if the PMA bit of the HFI is set to a one and the NOAR field indicates that there are (n) address regions (where n > 2) contained within the module. This field would contain the starting address of the nth memory region.	U
End of previous field + 1	4	NOB#n	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address of the nth memory region.	U
End of previous field + 1	2	NOAM	The **N**umber **O**f **A**dditional **M**odules optional field is only present if this header includes the address information of all software/calibration modules to be downloaded into the ECU (MPFH and PMA bits of HFI must be a “1”). This 2-byte value represents the number of additional calibration modules for which this header includes the address information.	U
End of previous field + 1	2	DCID_am1	This optional field is present if the H_N_{Size} bit of the HFI is 1, and the DCID bit in the HFI field is also a 1. This field is two bytes in length and contains a Data Compatibility identifier value for the first additional module. This value is compared by the programming executive against a corresponding value in the calibration data file header (DCID_) to ensure that there is no mismatch between the calibration file and the operational software.	U
End of previous field + 1	2	NOAR_am1	This field is optional and is only present if the NOAM field is ≥ 1. The data represents the number of address regions of the first additional module.	U
End of previous field + 1	4	PMA#1_am1	This optional field contains the starting **P**roduct **M**emory **A**ddress of the first (or only) Address Region of the first additional module (NOAM ≥ 1). If the product uses less than four bytes for addressing, then the most significant byte(s) shall be set to $00.	U
End of previous field + 1	4	NOB#1_am1	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address from the PMA#1_am1 field.	U
End of previous field + 1	4	PMA#2_am1	This optional field is present if the NOAR_am1 field of the header is present and set to a value greater than 1. This field contains the starting address of the 2nd memory region of the first additional module.	U
End of previous field + 1	4	NOB#2_am1	This optional field is present if the PMA#2_am1 field of the header is present. This field contains the **n**umber **o**f **b**ytes to be sent starting with the address of the 2nd memory region of the first additional module.	U
:	:		:	U

Offset	Number of Bytes	Name	Description	Cvt
End of previous field + 1	4	PMA#n_am1	This optional field represents the starting address of the nth memory region within the first additional module.	U
End of previous field + 1	4	NOB#n_am1	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address of the PMA#n_am1 field.	U
:	:		:	U
End of previous field + 1	2	DCID_am^x	This optional field is present if the H_N$_{Size}$ bit of the HFI is 1, and the DCID bit in the HFI field is also a 1. This field is two bytes in length and contains a Data Compatibility identifier value for the xth additional module. This value is compared by the programming executive against a corresponding value in the calibration data file header (DCID_) to ensure that there is no mismatch between the calibration file and the operational software.	U
End of previous field + 1	2	NOAR_am^x	This field is optional and is only present if the NOAM field is > 1. The data represents the number of address regions of the xth additional module.	U
End of previous field + 1	4	PMA#1_am^x	This optional field contains the starting **P**roduct **M**emory **A**ddress of the first (or only) Address Region of the xth addiotnal module (NOAM > 1). If the product uses less than 4 bytes for addressing, then the most significant byte(s) shall be set to $00.	U
End of previous field + 1	4	NOB#1_am^x	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address from the PMA#1_am^x field.	U
End of previous field + 1	4	PMA#2_am^x	This optional field is present if the NOAR_am^x field of the header is present and set to a value greater than 1. This field would contain the starting address of the 2nd memory region of the xth additional module.	U
End of previous field + 1	4	NOB#2_am^x	This optional field is present if the PMA#2_am^x field of the header is present. This field would contain the **n**umber **o**f **b**ytes to be sent starting with the address of the 2nd memory region of the xth additional module.	U
:	:		:	U
End of previous field + 1	4	PMA#n_am^x	This optional field represents the starting address of the nth memory region within the xth additional module.	U
End of previous field + 1	4	NOB#n_am^x	This optional field contains the **n**umber **o**f **b**ytes to be sent starting with the address of the PMA#n_am^x field.	U

The Header Format Identifier (HFI) byte is defined as follows (Tables 220 and 221):

Table 220: Header Format Identifier Definition (Only/MSB)

B7	B6	B5	B4	B3	B2	B1	B0
L_{SWMI}	L_{SWMI}	L_{SWMI}	L_{SWMI}	L_{SWMI}	L_{DLS}	H_N_{Size}	PMA

Table 221: Header Format Identifier Definition (LSB - If Supported)

B7	B6	B5	B4	B3	B2	B1	B0
MPFH	DCID						

Where:

L_{SWMI} = The length of the SWMI field (bit 7 the MSB and bit 3 the LSB).

L_{DLS} = The length of the Design Level Suffix or Alpha Code field (“0” = 2-byte, “1” = 3-byte).

H_N_{Size} = This bit is used to determine the size of the HFI field and also the size of the NOAR field(s) if present. A “0” in this bit indicates that the HFI field in the header is a single byte with bit definitions as defined in **Table 220**. A “1” in this bit indicates that the HFI field in the header is two bytes in length with the bits in **Table 220** representing the bits of the most significant byte and the bits in **Table 221** representing the bits of the least significant byte. A “1” in this bit shall also mean that all NOAR field(s) present in the header are 2 bytes in length. A “0” in this bit shall mean that the only NOAR field (if present based on value of PMA bit) is a single byte in length.

PMA = A “1” in this bit indicates that the NOAR field(s), the Product Memory Address field(s), and the Memory Length field(s) are used. The value contained within the NOAR field determines the number of PMA fields and Memory Length fields contained for a given module within the module header. A “0” in this bit indicates that there are no NOAR fields, PMA fields, or Memory Length fields included in the module header.

MPFH = This bit is used when a controller wants to include the address information for all of the calibration files into the header of the operational software module (multi-part file header). A “1” in this bit means that the address information of all files is included in this module header. This bit shall be set to a “0” if the LSB of the HFI is included in the header and the PMA bit is set to “0”.

DCID = This bit indicates that the data file header contains a data compatibility identifier. The value contained in this field is checked to see if a software data file is compatible with the version of boot software in the ECU or to see if a calibration data file is compatible with the operational software in the ECU.

Note: Unused Bits in the LSB of the HFI field (if LSB is included in the header) are reserved for future definition and shall be set to 0 until defined.

9.3.3.1.1 Examples of Software/Calibration File Headers. In the first example, the boot software is utilizing the PMA information provided in the header of the calibration module to determine the ECU physical addresses for programming. The data within a calibration module will occupy two memory regions in the ECU according to Figure 37. The start address of memory region 1 is $8000 and the length is $1000 bytes. The start address of memory region 2 is $A500 and the length is $0B00 bytes. The calibration module part number is used as the SWMI and is stored in the header in 4 byte hexadecimal format. The Design Level Suffix or Alpha Code is stored in 2-byte ASCII format. See Figure 37.

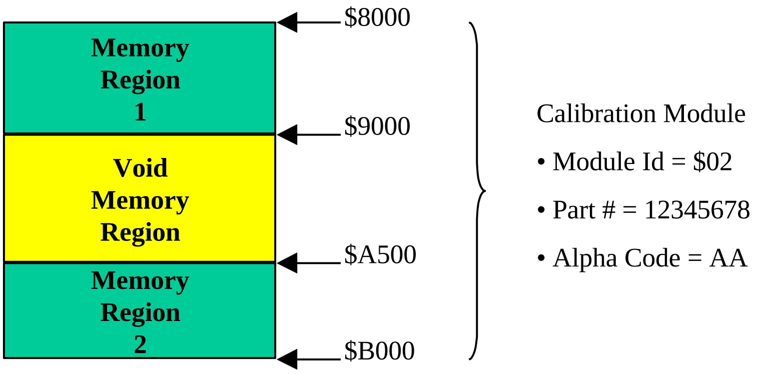


Figure 37: Calibration Module Consisting Of Two Memory Regions

The header for this example would contain the following information (Table 222):

Table 222: Calibration Module Header Data for Example 1

Offset	Number of bytes	Description
$00	2	Module checksum or CRC (hex) = $xx xx
$02	2	Module ID = $00 02
$04	1	Header Format Identifier = $21
$05	4	SWMI (part # 12345678) = $00 BC 61 4E
$09	2	Alpha Code = AA ($41 41)
$0B	1	NOAR = $02
$0C	4	Start address of 1st memory region = $00 00 80 00
$10	4	Length of 1st memory region = $00 00 10 00
$14	4	Start address of 2nd memory region = $00 00 A5 00
$18	4	Length of 2nd memory region = $00 00 0B 00

In the next example, the boot software determines the address information for the software and calibration modules from information stored in the software module header. The ECU is broken up into three modules (one operational software module and two calibration modules) as depicted in Figure 38.

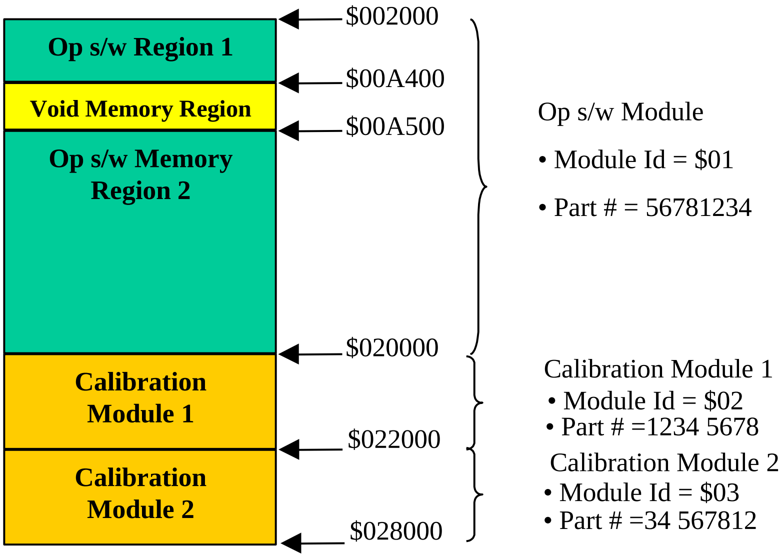


Figure 38: Example 2 - ECU with One Software Module and Two Calibration Modules

In addition, the following information is given for this **example:**

- The part number stored in the SWMI field of the headers is in 4-byte hexadecimal format.
- The Design Level Suffix or Alpha Code is stored in a 2-byte format and has a value of $AA for all modules.
- The tool set used to create the headers does so using word size parameters so the NOAR and HFI fields are two bytes in length for all modules.

The header for the operational software module in this example would contain the following information (Table 223):

Table 223: Operational Software Module Header Data for Example 2

Offset	Number of Bytes	Description
$00	2	Module checksum or CRC (hex) = $xx xx
$02	2	Module ID = $00 01
$04	2	Header Format Identifier = $23 80
$06	4	SWMI (part # 56781234) = $03 62 69 B2
$0A	2	Alpha Code = AA ($41 41)
$0C	2	NOAR = $00 02
$0E	4	PMA#1 = $00 00 20 00
$22	4	NOB#1 = $00 00 84 00
$26	4	PMA#2 = $00 00 A5 00
$2A	4	NOB#2 = $00 01 5B 00
$2E	2	NOAM = $00 02
$30	2	NOAR_am1 = $00 01
$32	4	PMA#1_am1 = $00 02 00 00
$36	4	NOB#1_am1 = $00 00 20 00

Offset	Number of Bytes	Description
$3A	2	NOAR_am2 = $00 01
$3C	4	PMA#1_am2 = $00 02 20 00
$40	4	NOB#1_am2 = $00 00 60 00

The headers for the calibration modules in this example would contain the following information (Tables 224 and 225):

Table 224: 1st Calibration Module Header Data for Example 2

Offset	Number of Bytes	Description
$00	2	Module checksum or CRC (hex) = $xx xx
$02	2	Module ID = $00 02
$04	2	Header Format Identifier = $22 00
$06	4	SWMI (part # 12345678) = $00 BC 61 4E
$0A	2	Alpha Code = AA ($41 41)

Table 225: 2nd Calibration Module Header Data for Example 2

Offset	Number of Bytes	Description
$00	2	Module checksum or CRC (hex) = $xx xx
$02	2	Module ID = $00 03
$04	2	Header Format Identifier = $22 00
$06	4	SWMI (part # 34567812) = $02 0F 76 84
$0A	2	Alpha Code = AA ($41 41)

Note: If the header of the operational software contains address information for all of the modules in the ECU then the boot software must be capable of determining which additional module address information correlates to a given Module Identifier. This can be accomplished by having the Module Identifiers sequentially numbered for each calibration module. With this approach the address information for the first additional module (contained in the operational software header) would correspond to Module Id $00 02 (since $00 01 is the operational software), the address information for the second additional module would be for Module Id $00 03 etc. This approach also supports downloading calibrations in any order since the boot software can mathematically determine which address information in the software module header is correct for the calibration module being downloaded based on the Module Id in the calibration module header.

In this final example, the boot software is utilizing the PMA information provided in the header of the module to determine the ECU physical addresses for programming. The data within a calibration module will occupy a single contiguous memory region in the ECU. The starting address where the module will be loaded in memory is $8000 and the length is $1000 bytes. The calibration module part number is used as the SWMI and is stored in the header in 4-byte hexadecimal format. The Design Level Suffix or Alpha Code is stored in 2-byte ASCII format. The tool set used to create the headers uses only a single byte for the HFI field and the NOAR field.

For this example, the following information is given:

- The Module Id is $02.
- The calibration part number is 12345678.
- The Design Level Suffix or Alpha Code is $AA.

The header for this example would contain the following information (Table 226):

Table 226: Calibration Module Header Data for Example 3

Offset	Number of Bytes	Description
$00	2	Module checksum or CRC (hex) = $xx xx
$02	2	Module ID = $00 02
$04	1	Header Format Identifier = $21
$05	4	SWMI (part # 12345678) = $00 BC 61 4E
$09	2	Alpha Code = AA ($41 41)
$0B	1	NOAR = $01
$0C	4	Starting address of memory region = $00 00 80 00
$10	4	Length of memory region = $00 00 10 00

9.3.4 Utility File Requirements. All utility files shall meet the requirements specified in the Service Programming System (SPS) Interpreter Programmers Reference Manual.

Utility files shall be designed such that calibrations are reprogrammed any time there is a change to the operational software. If the ECU contains multiple microprocessors (each with their own operational software and calibrations), then the utility file shall be designed such that calibrations are always reprogrammed on a microprocessor any time that that same processors operational software is programmed. Utility file design requirements (for multi-processor ECUs) relative to the reprogramming of operational software and/or calibrations of one processor when a second processor is reprogrammed shall be specified in the CTS (or SSTS or supplemental diagnostic specification referenced by either of the preceeding documents).

Utility files released to the assembly plant or service shall be in binary format. The Module ID for the Programming Utility data file shall be $00.

9.3.5 ECU Supplier Requirements. Boot software shall be programmed by the ECU supplier.

If the ECU supports the SecurityAccess ($27) service for SPS programming then the supplier shall program a random seed and the corresponding key value assigned by GM Service and Parts Operations into the ECU during the ECU manufacturing process.

The utility file necessary for SPS programming shall be developed by the ECU supplier.

The CTS shall specify if operational software and/or calibrations are to be programmed by the supplier.

9.3.6 Assembly Plant Requirements. The cumulative time to program all ECUs in the vehicle assembly process shall be less than the maximum time specified in the General Assembly Programming And Test section of the GM Bill Of Process (BOP).

The ECU received at the vehicle assembly plant must be externally identifiable prior to installation into the vehicle.

All ECUs that implement the SecurityAccess service for SPS programming shall be shipped to the vehicle assembly plant with the security mechanism unlocked (via the use of the MEC). Refer to paragraph 9.3.2.6 for specifics.

Note: The MEC is not required to be supported in the boot software if the ECU is always unlocked when operating out of the boot software. Even though the boot software may not evaluate the value of the MEC, the memory address where the MEC is located shall be initialized as documented in paragraph 9.3.2.6.2.

9.4 ECU Programming Process. The overall programming procedure is comprised of the following sub-processes:

a. Read Identification Information. This is a single path consisting of all the steps required to retrieve the necessary ECU identification data for a proper selection of the data in the SPS database. This step is required for service, not necessarily for assembly plant programming.

b. Retrieve SPS data. This is a single path consisting of all steps required to retrieve the proper SPS data out of the SPS database. The selection is based on information read during step a. (service) or known in advance (assembly plant).

c. Programming Session.

   1. **Setup (Pre-Utility File).** This is a single path consisting of all the steps necessary to identify the target node(s), and prepare the GMLAN link(s) for the programming event. All steps of the pre-utility file process, except certain steps required for the verification process, are functionally addressed to all nodes on the GMLAN sub-networks.
   2. **Utility File.** These are all the steps defined within the utility file(s) for the target node(s). This may be a single path of steps (in the case of a single node being programmed or multiple nodes programmed sequentially), or multiple paths if multiple nodes are being programmed in parallel (multiple utility files running in parallel). This path also includes steps to conclude a programming event (synchronization between utility files running in parallel). All diagnostic services contained in one utility file are physically addressed to the ECU to be programmed to allow for parallel programming.

9.4.1 Read Identification Information Process. The following steps are required to read the necessary identification data out of the vehicle in order to retrieve the correct data out of the SPS data base. All steps are performed automatically they are not utility file driven. See Table 227.

Note: All steps defined are required for field service programming and are not necessarily required for assembly plant programming.

Table 227: Steps of the RequestInfo Process

Step	Action
1	Wake-up link(s) - **High voltage wake-up for SWCAN** - gateways may cascade wake-ups (where applicable) across to the other subnets on the vehicle or the tester may need to use **level $04** of diagnostic service **InitiateDiagnosticOperation ($10)** to generate the wake-up (functionally addressed to gateway ECUs only). **Note:** The tester must allow at least 500 ms after issuing a wake-up before transmitting any diagnostic request message (including tester present) to allow the ECU to process the wake-up and transition to a communication active state – see paragraph 6.1 for wake-up requirements.
2	Start sending **TesterPresent ($3E)** periodic messages to all nodes - all links as needed (using the AllNodes CANId and AllNodes extended address).
3	Use service **ReadDataByIdentifier ($1A)** service to determine diagnostic addresses and CAN Identifiers for all ECU(s) (excluding SPS_TYPE_C) on each subnet (setup of base diagnostic configuration matrix), **DID $B0.** • This determines the relationship of each ECU's diagnostic address and its physical request and response CAN Identifiers. • SPS_TYPE_C ECUs would not respond since diagnostic responses have not been enabled yet. • This step and the ReportProgrammedState ($A2) service step which occurs later in the process allows the utility file to only contain the node diagnostic address. The tester correlates the diagnostic address to the physical request and response CAN Identifiers.
4 optional step	Send an **InitiateDiagnosticOperation ($10)** request with a sub-parameter value of **$02 (disableAllDTCs)** using the AllNodes request CANId and the AllNodes extended address. • This step ensures that nodes that poll the CAN controller do not set DTCs if they do not process the mode $28 request (in the next step) before they could possibly set signal supervision failures.

Step	Action
5	Send a **DisableNormalCommunication ($28)** request (with AllNodes CANId and AllNodes extended address - on each link as needed). • This disables sending and processing of normal mode data. • An SPS_TYPE_C ECU does not respond to the $28 request but tracks receipt of the $28 request (in order to enable diagnostic responses after receiving the $A2 service later in the process). • The tester shall not enable diagnostic responses in SPS_TYPE_C ECUs (via mode $A2) until after it has received positive responses to the $28 request from all other ECUs on the link (as stored in the diagnostic configuration matrix).
6	Determine which nodes are programmable using the service **ReportProgrammedState ($A2)** with the AllNodes CANId and AllNodes extended address (finalization of the diagnostic configuration matrix - on each link as needed). • Each SPS programmable ECU shall respond. • SPS_TYPE_A and SPS_TYPE_B ECUs respond with their permanent diagnostic USDT response CANId (which falls within the reserved range of diagnostic response CAN Identifiers). • An SPS_TYPE_C ECU responds with the SPS_PrimeRsp CANId ($3xx where xx = diagnostic address). • At this point the SPS_TYPE_C ECU would activate and accept the SPS_PrimeReq ($0xx) CANId as a diagnostic PTP request CANId (where xx = diagnostic address). • MSSC ECUs shall only respond to the $A2 service on the subnet where they support programming (MSSC ECUs shall only be programmable on one subnet – see 4.3.4.
7	Retrieve necessary ECU ID and vehicle option content (e.g., engine type) information from the programmable node(s) (using **ReadDataByIdentifier ($1A)** service). • Use the SPS_PrimeReq ($0xx) diagnostic request CANId for each SPS_TYPE_C ECU targeted. • Use USDT permanent diagnostic request CANId (determined based on offset from response CANId received) for SPS_TYPE_A and SPS_TYPE_B ECUs. • Use information received via the previously executed service ReportProgrammedState ($A2) to identify all SPS programmable ECUs.
8	Retrieve seed of programmable module(s) using the **SecurityAccess ($27)** service. • Security may not be implemented in all ECUs that are SPS programmable. The SecurityAccess service is required for all SPS programmable ECUs that are Emission, Safety, or Theft related.
9 optional step	Send a **ReturnToNormalMode ($20)** request (with AllNodes CANId and AllNodes extended address - on each link as needed). • This request transitions all subnets back to normal operation while the tool is retrieving information from the SPS database. This step is optional since a $P3_C$ timeout would occur after the tester is disconnected and this shall also cause all nodes on all subnets to revert back to normal operation.
10	Disconnect tester.

9.4.2 Retrieve SPS Data Process. This process is performed to retrieve the proper SPS data out of the SPS database. See Table 228.

Note: For service, this process uses the information retrieved during the Read Identification Information process to retrieve the SPS data out of the SPS database.

Table 228: Steps of the Retrieve SPS Data Process Programming Session

Step	Action
11	Connect tester to SPS database. • Select node(s) to be programmed. • If node replacement, identify information (if necessary) to be retrieved from the old ECU to transfer to the new ECU Note 1. • Generate security key(s). • Retrieve SW/CAL Modules to download.
12	Reconnect tester.

Note 1: For remote programming, where an ECU is replaced and requires data read from the old ECU and written into the replacement ECU the technician must connect to the SPS database twice. The first time is to identify which data must be read from the old ECU, the second time is to generate the key for the replacement ECU and to retrieve the software/calibration file(s) to be downloaded.

9.4.3 Programming Session. (Figure 39) The Programming Session consists of two distinct types of diagnostic service executions:

1. **Master Execute.** All steps required to start a programming event and all steps required to be synchronized between multiple utility files running in parallel are executed only once (master execute). Those steps are the ones defined for the **Pre-Utility File Process** and the synchronization between the utility file at the point in time when the utility file reaches the point where a SW-reset of the physical ECU is required to finalize the programming of this node (conclusion of the programming event). All steps controlled by the master execute are vehicle oriented steps (functionally addressed to all nodes).
2. **Interpreter Execute.** All steps required to program an ECU are contained in a utility file. Those steps do not need to be synchronized between multiple utility files running in parallel, therefore no **master execute** is required during the execution of these steps. All steps controlled by the interpreter (interpreter execute) are ECU oriented steps (physically addressed to the ECU to be programmed).

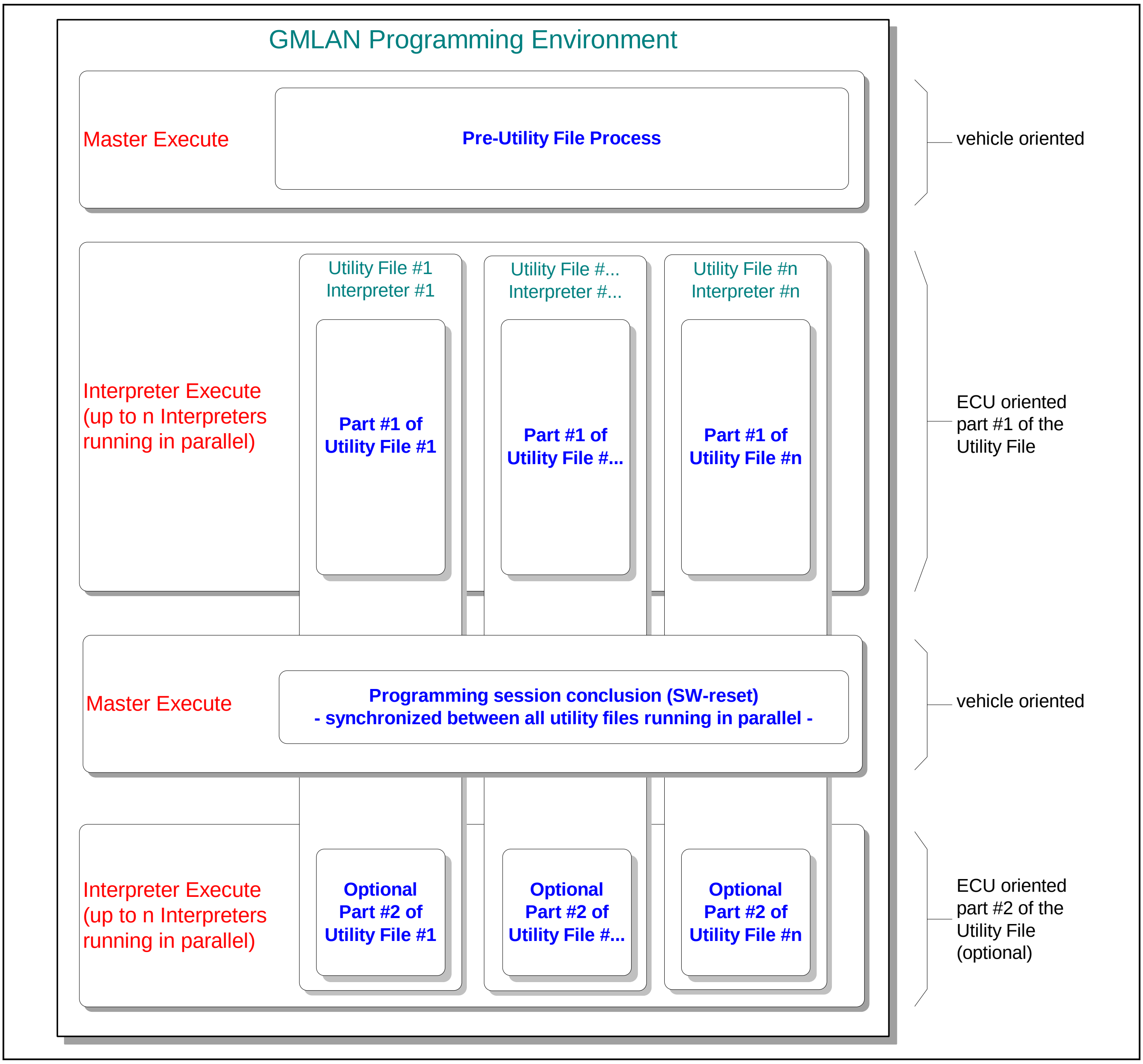


Figure 39: Programming Procedure - Overview

9.4.3.1 Programming Setup (Pre-Utility File) Process. The following steps (Table 229) are performed prior to the execution of one or multiple utility file(s) to make sure that the GMLAN network (including all GMLAN sub-networks) transitions to the programming state (vehicle oriented step). In addition the Pre-Utility File process determines the CAN communication parameters (setup of the diagnostic configuration matrix). All steps of the pre-utility file process are functionally addressed to all nodes on the GMLAN sub-networks (except certain steps included in the verification process) and performed automatically.

Table 229: Steps of the Pre-Utility File Process

Step	Action
13	Wake-up link(s) - **High voltage wake-up for SWCAN** - gateways may cascade wake-ups (where applicable) across to the other subnets on the vehicle or the tester may need to use **level $04** of diagnostic service **InitiateDiagnosticOperation ($10)** to generate the wake-up (functionally addressed to gateway ECUs only). **Note:** the tester must allow at least 500ms after issuing a wake-up before transmitting any diagnostic request message (including tester present) to allow the ECU to process the wake-up and transition to a communication active state – see section 6.1 for wake-up requirements.
14	Start sending **TesterPresent ($3E)** periodic messages to all nodes - all links as needed (using the AllNodes CANId and AllNodes extended address).
15	Use service **ReadDataByIdentifier ($1A)** to determine diagnostic addresses and CAN Identifiers for all ECU(s) (excluding SPS_TYPE_C) on each subnet (setup of base diagnostic configuration matrix, **DID $B0**). • This determines the relationship of each ECUs diagnostic address and its physical request and response CAN Identifiers. • SPS_TYPE_C ECUs would not respond since diagnostic responses have not been enabled yet. • This step and the ReportProgrammedState ($A2) service step later in the process allow the utility file to only contain the node diagnostic address. The tester correlates the diagnostic address to the physical request and response CAN Identifiers.
16 optional step	Send a **InitiateDiagnosticOperation ($10)** request with a sub-parameter value of **$02 (disableAllDTCs)** using the AllNodes request CANId and the AllNodes extended address. • This step ensures that nodes that poll the CAN controller do not set DTCs if they do not process the mode $28 request (in the next step) before they could possibly set signal supervision failures.
17	Send a **DisableNormalCommunication ($28)** request (with AllNodes CANId and AllNodes extended address - on each link as needed). • This disables sending and processing of normal mode data. • An SPS_TYPE_C ECU does not respond to a $28 request but tracks receipt of the $28 request (in order to enable diagnostic responses after receiving the $A2 service later in the process). • The tester shall not enable diagnostic responses (via mode $A2) in SPS_TYPE_C ECUs until after it has received positive responses to the $28 request from all other ECUs on the link (as stored in the diagnostic configuration matrix).

Step	Action
18	Determine which nodes are programmable using the service **ReportProgrammedState ($A2)** with the AllNodes CANId and AllNodes extended address (finalization of the diagnostic configuration matrix - on each link as needed). • Each programmable ECU shall respond. • SPS_TYPE_A and SPS_TYPE_B ECUs respond with their permanent diagnostic USDT response CANId (which falls within the reserved range of diagnostic response CAN Identifiers). • An SPS_TYPE_C ECU responds with its SPS_PrimeRsp CANId ($3xx where xx = ECU diagnostic address). • At this point the SPS_TYPE_C ECU would activate and accept the SPS_PrimeReq ($0xx) CANId as a diagnostic PTP request CANId (where xx = diagnostic address). • MSSC ECUs shall only respond to the $A2 service on the subnet where they support programming (MSSC ECUs shall only be programmable on one subnet - see 4.3.4.)
19 optional step	Verify that the tester is connected to the correct vehicle - the same vehicle from which the seed was retrieved (e.g., use the service **ReadDataByIdentifier ($1A)** with specific DID to read certain ECU Id data). • This step can only be performed after the enabling of the diagnostic response messages of SPS_TYPE_C ECUs. • This step may be used for service (e.g., required for remote programming) to make sure that the tester is connected to the same vehicle as during the Request Identification Information Process.
20	Request the appropriate level of the **ProgrammingMode ($A5)** service used to determine if it is OK to establish a programming event (send to all nodes/on all links). • On **HS-CAN** and **MS-CAN** the sub-parameter **level $01 (requestProgramming)** shall be used. • On **LS-CAN** the sub-parameter **level $01 (requestProgramming)** or **level $02 (requestProgramming_HighSpeed)** shall be used. • The tester shall verify that no negative responses are sent prior to the next step which enables the programming mode. • All ECUs shall respond.
21	Enable programming mode with service **InitiateProgramming ($A5)** and sub-parameter **level $03** (enableProgramming). • No response to this request from any ECU. • ECUs shall transition to high speed mode if high speed was specified in the previous step. • The tester shall transition its protocol device to high speed operation (if requested for the low speed subnet) and wait before communicating at high speed 100 ms longer than the time value specified in service $A5.

9.4.3.2 Programming Utility File Process. This part contains all steps required to perform a security access to the node to be programmed and the steps for the transfer of the data to the physical node. See Table 230.

For the physically addressed request messages contained in the utility file, the tester uses the diagnostic PTP request CANId of the individual node(s) to be programmed (as determined via the diagnostic services $1A, $A2 and stored in the diagnostic mapping matrix). The exceptions to this are the TesterPresent messages transmitted automatically by the SPS communication layer and the synchronization step where the **master execute** transmits a functionally addressed (AllNode) ReturnToNormalMode ($20) request.

Table 230 shows the general process incorporated within the utility file. The actual utility file instruction set may incorporate more steps as required by the specific node, particularly regarding the actual data transfer steps.

Table 230: Steps of the Utility File Process

Step	Action
22	**SecurityAccess ($27)** For those nodes which are theft, emission, or safety related (or contain functionality otherwise bound by legal requirements).
23	**RequestDownload ($34)** • Tells a node to allocate the necessary resources needed for programming (and forces the node to begin operating out of boot software if boot exists and the node is not already executing out of the boot). **Note:** Steps performed by the ECU on reception of this service depend on the ECU internal structure. • A node which has been previously programmed shall store its permanent PTP request and USDT diagnostic response CAN Identifiers in a way such, that the boot software can access them during the remainder of the programming event.
24	**DataTransfer ($36)** • Used to download a programming algorithm into a node or execute a node resident routine and to transfer the operational software and/or calibration data to the node. • Interleave TesterPresent messages using AllNodes CANId with AllNodes extended address. • Continue until the node is completely programmed.
25	Optionally use the **WriteDataByIdentifier ($3B)** service to write any data which needs to be performed prior to executing a software reset.

Note: It may be necessary to perform multiple service $34/$36 sequences to download the data into the ECU (e.g., one sequence for each downloaded module).

Certain steps may be required to be performed after a software reset of an ECU. When the utility file of an ECU reaches this point, the interpreter has to synchronize a software reset with all other interpreters running in parallel. At the point in time when each interpreter reaches the step in the utility file where the conclusion of the programming session is requested the tester shall conclude the programming event by transmitting a ReturnToNormalMode ($20) request message onto each GMLAN subnet (master execute). See Table 231.

Table 231: Programming Procedure Conclusion

Step	Action
26	**ReturnToNormalMode ($20)** using the AllNodes CANId with AllNodes extended address (to all links as required). If the tool has established the programming session on SWCAN then it must ensure that the service $20 is sent on this bus first so all ECUs on SWCAN can switch the baud rate before the MSSC ECUs reset and potentially cause errors (due to its transceiver being in low speed operation and all other nodes still in high speed operation). • The nodes do not respond to a mode $20 request if ProgrammingMode ($A5) service is enabled. • Receipt of a $20 request causes the nodes to exit ProgrammingMode ($A5). • Receipt of a $20 request also switches the low speed link back to low speed mode (if the programming event was performed in high speed mode). • All nodes perform a software reset to enable execution of the new operational software and/or calibration and to synchronize communications startup. • **After sending the ReturnToNormalMode ($20) request, the Tester must wait 1 s to allow sufficient time for all nodes to reset and synchronize.** • **The tester suspends transmission of Tester Present ($3E) messages at this time.**

Note: Following a ReturnToNormalMode ($20) service the determination of the diagnostic addresses and diagnostic CAN Identifiers (using the service $1A with DataIdentifier $B0) is performed automatically prior to the execution of a diagnostic service. See Table 232, Step 29.

Following the ReturnToNormalMode ($20) service additional action can be performed for each ECU (utility file driven - interpreter execute) to finalize the programming. This part contains all steps required to finally conclude a programming event of a single ECU. For some nodes it may be required to write data after the software reset, when the operational software is running. Those steps are contained in the ECU specific utility file. See Table 232.

This part is utility file driven except for the determination of the GMLAN communication parameters (CAN Identifiers).

Table 232: Optional Utility File Steps Following the Programming Conclusion Step

Step	Action
27	Wake-up link(s) - **High voltage wake-up for SWCAN** - gateways may cascade wake-ups (where applicable) across to the other subnets on the vehicle or the tester may need to use **level $04** of diagnostic service **InitiateDiagnosticOperation ($10)** to generate the wake-up (functionally addressed to gateway ECUs only). This is necessary because the gateway ECU(s) has (have) been reset and might have lost the internal information about keeping a wake-up signal line asserted on another subnet on the vehicle. **Note:** The tester must allow at least 500ms after issuing a wake-up before transmitting any diagnostic request message (including tester present) to allow the ECU to process the wake-up and transition to a communication active state. Refer to paragraph 6.1 for wake-up requirements.
28	Start sending **TesterPresent ($3E)** periodic messages to all nodes - all links as needed (using the AllNodes CANId and AllNodes extended address).
29	The tester must now issue a request for service **ReadDataByIdentifier ($1A)** with **DID $B0** in order to perform a relearn of the diagnostic address and CANIds. This is necessary because an ECU which was programmed and performed a reset after the ReturnToNormalMode ($20) service from Step 23, would now use its permanent CAN Identifiers. This step is performed automatically prior to the execution of utility file driven diagnostic service (e.g., Step 25) following the ReturnToNormalMode ($20) service.
30	The **WriteDataByIdentifier ($3B)** service may be used to program option content information into SPS programmable ECUs which were just programmed.

Note: The tester may send a service $04 request to clear DTCs at the conclusion of the programming event. This is typically done at the conclusion of module programming in the vehicle assembly plant.

9.4.4 Summary. Figure 40 contains the graphical representation of the GMLAN Programming Procedure as defined in the previous sections.

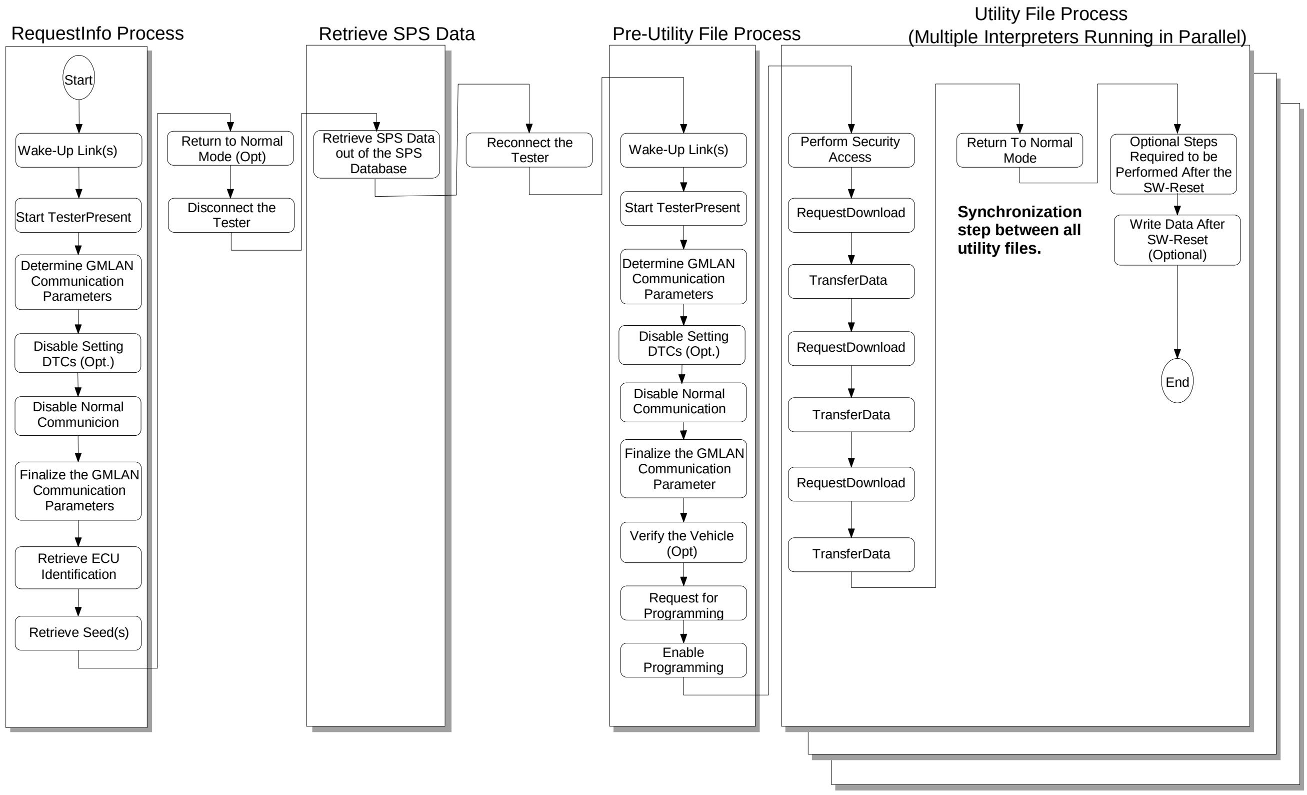

RequestInfo Process
Retrieve SPS Data
Pre-Utility File Process
Utility File Process
(Multiple Interpreters Running in Parallel)
Start
Wake-Up Link(s)
Start TesterPresent
Determine GMLAN Communication Parameters
Disable Setting DTCs (Opt.)
Disable Normal Communicion
Finalize the GMLAN Communication Parameters
Retrieve ECU Identification
Retrieve Seed(s)
Return to Normal Mode (Opt)
Disconnect the Tester
Retrieve SPS Data out of the SPS Database
Reconnect the Tester
Wake-Up Link(s)
Start TesterPresent
Determine GMLAN Communication Parameters
Disable Setting DTCs (Opt.)
Disable Normal Communication
Finalize the GMLAN Communication Parameter
Verify the Vehicle (Opt)
Request for Programming
Enable Programming
Perform Security Access
RequestDownload
TransferData
RequestDownload
TransferData
RequestDownload
TransferData
Return To Normal Mode
Synchronization step between all utility files.
Optional Steps Required to be Performed After the SW-Reset
Write Data After SW-Reset (Optional)
End

Figure 40: Programming Procedure – Summary

Note: The Programming Procedure Summary shows the general steps of the GMLAN programming procedure. Optional steps are indicated with (Opt).

9.5 ECU Programming Message Flow Example. The following example assumes:

- Programming is to be performed on a single node on the SWCAN low speed link.
- There are four nodes on the SWCAN link.
    - Node 1 (N1) is an SPS_TYPE_A ECU that has a diagnostic USDT response CANId of $641, and a diagnostic address of $28.
    - Node 2 (N2) is not programmable, has a diagnostic USDT response CANId of $64D, and a diagnostic address of $60.
    - Node 3 (N3) is an SPS_TYPE_C ECU without operational software or calibration data and a diagnostic address of $40, (so its SPS_PrimeReq and SPS_PrimeRsp programming request and response CAN Identifiers are $040 and $340 respectively). Once Node 3 is programmed, its physical diagnostic request and response CANIds are $247 and $647 respectively.
    - Node 4 (N4) is a Gateway ECU that is not programmable, has a diagnostic USDT response CANId of $651, and a diagnostic address of $42.
- Only the low speed subnet (SWCAN link) high voltage wake-up method is shown in the following examples. The High Voltage wake-up message for the SWCAN low speed link uses CANId $100.
- Node 3 (N3) is the only node which will be programmed during the programming event.
- For this example, it is assumed the ECU being programmed supports 4-byte addressing (uses address $00 $00 $23 $FF) and also is expecting a 4-byte uncompressed memory size with the $34 service.

- For this example, it is assumed that the tester pads all diagnostic USDT request messages and all ECUs pad the response messages. Pad bytes are all filled with $AA to make it easy to distinguish which bytes are pad bytes.

9.5.1 Request Identification Information Process. (Tables 233 thru 254).

Wake-up link(s).

- Perform a high voltage wake-up on LS-CAN.

Table 233: High Voltage Wake-Up on LS-CAN

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(UUDT)	$100	---	---	---	---	---	---	---	---

- Wait 500 ms after wake-up, then transmit a service $10 request message (level $04), that forces Gateway ECUs to perform a wake-up on the sub-nets nets to which they are connected.

Table 234: Wake-Up Gateway ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT)	$101	$FD	$02	$10	$04	$AA	$AA	$AA	$AA
N4(USDT)	$651	$01	$50	$AA	$AA	$AA	$AA	$AA	$AA

Determine network configuration.

- Use service $1A with DataIdentifier $B0 to read diagnostic addresses of all ECUs (excluding SPS_TYPE_C).

Table 235: Read Diagnostic Addresses of All ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$1A	$B0	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$03	$5A	$B0	$28	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$03	$5A	$B0	$60	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$03	$5A	$B0	$42	$AA	$AA	$AA	$AA

Note: No response is sent by the SPS_TYPE_C ECU at this point.

Disable normal communication.

- Use service $28 to disable the normal communication.

Table 236: Disable Normal Communication

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI

Frame Type	CAN Id	#1	#2	#3	#4	#5	#6	#7	#8
T(USDT-SF)	$101	$FE	$01	$28	$AA	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA

Note: No response is sent by the SPS_TYPE_C ECU at this point.

ReportProgrammingState service $A2 request.

- Use service $A2 to determine which ECUs are programmable and which ECUs are SPS_TYPE_C.

Table 237: Determine Programmable ECUs and SPS_Type_C ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$A2	$AA	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$02	$E2	$00	$AA	$AA	$AA	$AA	$AA
N3(USDT-SF)	$340	$02	$E2	$01	$AA	$AA	$AA	$AA	$AA

Note: No response from the non-programmable ECUs because a functional request for a service which is not supported results in no response message being sent.

At this point, the tester would request ECU ID information using mode $1A in order to determine which files are needed for the programming event. This step is not shown in this example.

Next step, read seed.

- Use service $27 to read the seed from the programmable ECUs.

Table 238: Retrieve Seed from Programmable ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$27	$01	$AA	$AA	$AA	$AA
N1(USDT_SF)	$641	$04	$67	$01	$aa	$bb	$AA	$AA	$AA
N3(USDT-SF)	$340	$04	$67	$01	$cc	$dd	$AA	$AA	$AA

At this point the technician disconnects from the vehicle and plugs the tool into the SPS database computer. The technician selects only node N3 to be programmed. The SPS database transfers the security key and the programming files to the tester. Once completed, the tester is reconnected to the vehicle.

9.5.2 Programming Session.

9.5.2.1 Pre-Utility File Process.

Wake-up link.

- Perform a high voltage wake-up on LS-CAN.

Table 239: High Voltage Wake-Up on LS-CAN

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(UUDT)	$100	---	---	---	---	---	---	---	---

- Wait 500 ms after wake-up, then transmit a service $10 request message (level $04), that forces Gateway ECUs to perform a wake-up on the sub-nets to which they are connected.

Table 240: Wake-Up Gateway ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT)	$101	$FD	$02	$10	$04	$AA	$AA	$AA	$AA
N4(USDT)	$651	$01	$50	$AA	$AA	$AA	$AA	$AA	$AA

Determine network configuration.

- Use service $1A with DataIdentifier $B0 to read diagnostic addresses from all of the ECUs (excluding SPS_TYPE_C).

Table 241: Read Diagnostic Addresses of All ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$1A	$B0	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$03	$5A	$B0	$28	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$03	$5A	$B0	$60	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$03	$5A	$B0	$42	$AA	$AA	$AA	$AA

Note: No response is sent by the SPS_TYPE_C ECU at this point

Disable normal communication.

- Use service $28 to disable the normal communication.

Table 242: Disable Normal Communication

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$28	$AA	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$01	$68	$AA	$AA	$AA	$AA	$AA	$AA

Note: No response is sent by the SPS_TYPE_C ECU at this point.

ReportProgrammingState Mode $A2 request.

- Use service $A2 to determine all programmable ECUs and all SPS_TYPE_C ECUs.

Table 243: Determine Programmable ECUs and SPS_Type_C ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$A2	$AA	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$02	$E2	$00	$AA	$AA	$AA	$AA	$AA
N3(USDT-SF)	$340	$02	$E2	$01	$AA	$AA	$AA	$AA	$AA

Note: No response from the non-programmable ECUs because a functional request for a service which is not supported results in no response message being sent.

Initiate programming mode $A5 - request programming mode in high speed.

- Use service $A5 with sub-parameters $01 for HS- and MS-CAN and $02 for LS-CAN to request, if it is OK to start a programming event.

Table 244: Request OK to Start Programming Event

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$A5	$02	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$01	$E5	$AA	$AA	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$01	$E5	$AA	$AA	$AA	$AA	$AA	$AA
N3(USDT-SF)	$340	$01	$E5	$AA	$AA	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$01	$E5	$AA	$AA	$AA	$AA	$AA	$AA

EnableProgrammingMode - Service $A5.

- Use service $A5 with sub-parameter $03 for all GMLAN subnets to start the programming event. There is no response allowed to this sub-parameter level.

Table 245: Start Programming Event

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$A5	$03	$AA	$AA	$AA	$AA

Note: No responses are allowed to the request to enable programming mode to ensure that bus errors and/or bus off conditions do not occur.

After enabling high speed mode, the tester must delay for a period of time (reference the $A5 service for minimum time interval) before sending the next diagnostic request.

9.5.2.2 Utility File Process.

SecurityAccess ($27) service (using key information generated in the SPS data retrieval step).

Request Seed.

- Retrieve the seed from the physical ECU to be programmed.

Table 246: Retrieve Seed from ECU to be Programmed

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$040	$02	$27	$01	$AA	$AA	$AA	$AA	$AA
N3(USDT-SF)	$340	$04	$67	$01	$cc	$dd	$AA	$AA	$AA

Send Key.

- Send the Key to the physical ECU.

Table 247: Send Key to the ECU

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$040	$04	$27	$02	$cc	$dd	$AA	$AA	$AA
N3(USDT-SF)	$340	$02	$67	$02	$AA	$AA	$AA	$AA	$AA

RequestDownload ($34).

- Request the download of data via service $34.

Table 248: Request Download of Data

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$040	$06	$34	$00	$00	$00	$00	$00	$AA
N3(USDT-SF)	$340	$01	$74	$AA	$AA	$AA	$AA	$AA	$AA

TransferData ($36).

- Download the data into the physical ECU.

Table 249: Download Data into ECU

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$040	$10	$FF	$36	$80	$00	$00	$23	$FF
N3(USDT-FC)	$340	$30	$00	$00	$AA	$AA	$AA	$AA	$AA
T(USDT-CF)	$040	$21	aa	bb	cc	dd	ee	ff	gg
:									
T(USDT-CF)	$040	$24	xx	yy	$AA	$AA	$AA	$AA	$AA
N3(USDT-SF)	$340	$03	$7F	$36	$78	$AA	$AA	$AA	$AA
:									
N3(USDT-SF)	$340	$02	$76	$AA	$AA	$AA	$AA	$AA	$AA

Conclusion of the programming event (Software reset, to be synchronized with optionally running utility files in parallel):

ReturntoNormalMode request ($20).

- Use service $20 to conclude the programming event. This will force the ECUs to perform a software reset and transitions the LS-CAN subnet from high speed mode to normal speed mode.

Table 250: Conclude Programming Event

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$01	$20	$AA	$AA	$AA	$AA	$AA

Note: No responses to the service $20 request from any node (because programming mode was active). This eliminates any link errors due to unsynchronized switching from high speed mode. At this point all nodes perform a software reset.

Note: After issuing the ReturnToNormalMode ($20) service in this step, the tester suspends the TesterPresent ($3E) service, and must wait 1 s before proceeding to the next step to allow sufficient time for all the nodes to reset and syncronize normal communication.

Optionally steps may follow the ReturnToNormalMode ($20) request message to finalize the programming of the ECU. Those steps are controlled by the utility file of the ECU. The following shows a generic example of a service $3B write performed as part 2 of the utility file process.

Wake-up link.

- Perform a high voltage wake-up on LS-CAN.

Table 251: High Voltage Wake-Up on LS-CAN

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(UUDT)	$100	---	---	---	---	---	---	---	---

- Wait 500 ms after wake-up, then transmit a service $10 request message (level $04), that forces Gateway ECUs to perform a wake-up on the sub-nets to which they are connected.

Table 252: Wake-Up Gateway ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT)	$101	$FD	$02	$10	$04	$AA	$AA	$AA	$AA
N4(USDT)	$651	$01	$50	$AA	$AA	$AA	$AA	$AA	$AA

Determine network configuration.

- Use service $1A with DataIdentifier $B0 to read diagnostic addresses from all of the ECUs (excluding SPS_TYPE_C).

Table 253: Read Diagnostic Addresses of All ECUs

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-SF)	$101	$FE	$02	$1A	$B0	$AA	$AA	$AA	$AA
N1(USDT-SF)	$641	$03	$5A	$B0	$28	$AA	$AA	$AA	$AA
N2(USDT-SF)	$64D	$03	$5A	$B0	$60	$AA	$AA	$AA	$AA
N3(USDT-SF)	$647	$03	$5A	$B0	$40	$AA	$AA	$AA	$AA
N4(USDT-SF)	$651	$03	$5A	$B0	$42	$AA	$AA	$AA	$AA

Sample $3B write such as the VIN.

- Send a service $3B request to write DID $90 (VIN).

Table 254: Write VIN

T = Frame Sent By Tester, N = Frame Sent By Node; shaded region indicates PCI									
Frame Type	**CAN Id**	**#1**	**#2**	**#3**	**#4**	**#5**	**#6**	**#7**	**#8**
T(USDT-FF)	$247	$10	$13	$3B	$90	"W"	"0"	"L"	"0"
N(USDT-FC)	$647	$30	$00	$00	---	---	---	---	---
T(USDT-CF)	$247	$21	"J"	"B"	"F"	"3"	"5"	"W"	"1"
T(USDT-CF)	$247	$22	"0"	"4"	"2"	"7"	"6"	"5"	$AA
N(USDT-SF)	$247	$02	$7B	$90	---	---	---	---	---

10 Validation

10.1 General. The node verification procedure results shall be documented in the separately available GMW3110 Test Result Template. Note that if using CANoe Option DiVa with GM Extensions for DiVa then the resulting report from CANoe may be provided in lieu of the GMW3110 template.

10.2 Validation Cross Reference Index. Not applicable.

10.3 Supporting Paragraphs. Not applicable.

11 Provisions for Shipping

Not applicable.

12 Notes

12.1 Glossary.

12.1.1 ISO Terms: This document makes use of terms defined in the ISO 15765-2 Road Vehicles - Diagnostics on Controller Area Networks (CAN) document.

ADDR: Address information.

BS: Block Size.

CAN: Controller Area Network.

CF: Consecutive Frame.

DL: Data Length.

FC: Flow Control.

FF: First Frame.

PCI: Protocol Control Information.

SF: Single Frame.

STmin: Separation Time (minimum).

USDT: Unacknowledged Segmented Data Transfer - used to signify messages which may be segmented into multiple frames due to data length of the message.

UUDT: Unacknowledged Unsegmented Data Transfer - used to signify messages which are single frame ONLY.

12.1.2 SAE Terms: This document makes use of terms defined in the SAE J1930 Electrical/Electronic Systems Diagnostic Terms, Definitions, Abbreviations and Acronyms document.

12.1.3 GM Terms

CTS: Component Technical Specification

Diagnostic Mode: An ECU is considered to be in **Diagnostic Mode** any time a diagnostic service is active which requires that a TesterPresent ($3E) message be sent periodically in order to maintain the functionality of the service. See the description of each diagnostic service to see if it places an ECU in **Diagnostic Mode**.

ECU: Electronic Control Unit. An ECU is a node on a subnet.

Gateway: This is a special case MSSC that has additional requirements to transfer data between multiple subnets and may provide the wake-up mechanism for one or more of the subnets.

HS-CAN: Generic reference for all dual wire High speed buses. Includes, but not limited to the Primary HS bus, Chassis Expansion bus, and Powertrain Expansion bus.

LS-CAN: Generic reference for all single wire Low speed buses. Includes, but not limited to the Primary LS bus.

MS-CAN: Generic reference for all dual wire Mid speed buses. Includes, but not limited to the Primary MS bus

MSSC: Multiple Subnet Signal Consumer. An ECU that is connected to multiple subnets but does not provide wake-up features and does not transfer data between subnets.

Node: An ECU on a network.

Permanent Diagnostic CAN Identifiers: These are the CAN Identifiers supported by an ECU which are used to support the transfer of diagnostic services between the tester and an ECU, or the tester and multiple ECUs.

SPS: Service Programming System

SPS_PrimeReq CANId: This is a special case CANId that is used to receive diagnostic requests during SPS programming for SPS_TYPE_C ECUs.

SPS_PrimeRsp CANId: This is a special case CANId that is used to transmit diagnostic responses during SPS programming for SPS_TYPE_C ECUs.

SPS_TYPE_A or SPS_TYPE_B or SPS_TYPE_C ECU: These terms refer to ECUs which are programmable via the SPS. SPS programmable ECUs utilize the utility file concept for reprogramming. The differences between an SPS_TYPE_A, SPS_TYPE_B and SPS_TYPE_C ECU is what portion of the ECU is programmed, and whether or not the ECU is capable of using its permanent diagnostic CAN Identifiers during the programming event. Refer to paragraph 9.1 of this specification for further definition of these terms.

SSTS: Subsystem Technical Specification

Utility File: This is a file that contains the programming instructions for an ECU which is capable of being programmed via SPS.

12.2 Acronyms, Abbreviations, and Symbols.

ABS	Anti-Lock Brake System
A/C	Air Conditioning
ASCII	American Standard Code for Information Interchange
BCD	Binary Coded Decimal
BIN	Binary or bit encoded
bps	bits per second
CANID	CAN Identifier
ConTS	Continue To Send
CPID	Control Packet Identifier
CRC	Cyclical Redundancy Check
DID	Data Identifier
DLC	Diagnostic Link Connector
DLS	Design Level Suffix
DPID	Data Parameter Identifier
DRE	Design Release Engineer
DTC	Diagnostic Trouble Code
DVT	Dynamic Vehicle Test
EA	Extended Address
EEPROM	Electrically Eraseable Programmable Read Only Memory
EGR	Exhaust Gas Recirculation
EOBD	European On Board Diagnostics
FM	Frequency Modulation
FS	Flow Status
GMLAN	GM Local Area Network
GMW	GM Worldwide
HFI	Header Format Identifier
IAC	Idle Air Control
ISO	International Standards Organization
LEV	Level
LIN	Local Interconenct Network
LSB	Least Significant Byte
MA	Memory Address Parameter
MID	Module Identifier
MS	Memory Size Parameter
MSB	Most Significant Byte
MSG	Message Number
NOAM	Number Of Additional Modules
NOAR	Number Of Address Regions
OBD	On Board Diagnostics
OVFLW	Overflow
PCM	Powertrain Control Module
PDS	Periodic DPID Scheduler
PDS	Product Description Summary
PID	Parameter Identifier
PMA	Product Memory Address
PRNDL	Transmission Range Select (Park, Reverse, Neutral, Drive, Low)
PTP	Point to Point
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read Only Memory
rpm	revolutions per minute
SDU	Service Data Unit
SID	Service Identifier
SIDPR	Service Identifier Positive Response
SIDRQ	Service Identifier Request
SN	Sequence Number
SPI	Serial Peripheral Interface
SPO	Service Parts Operations
SPS	Service Programming System
SSLT	Subsystem Leadership Team
SWCAN	Single Wire CAN
SWMI	Software Module Identifier
USN	Unsigned Numeric
VIN	Vehicle Identification Number
VN	Virtual Network
VNMF	Virtual Network Management Frames
WFT	Wait Frame Transmission
WuP	Wake-up

13 Additional Paragraphs

13.1 All parts or systems supplied to this standard must comply with the requirements of GMW3059, **Restricted and Reportable Substances for Parts.**

14 Coding System

This standard shall be referenced in other documents, drawings, etc., as follows:

GMW3110

15 Release and Revisions

15.1 Release. This standard originated in May 1998. It was first approved by the GMLAN Diagnostic Sub-Group in April 2000. It was first published in April 2000.

15.2 Revisions.

Rev	Approval Date	Description (Organization)
D	FEB 2004	Version 1.5 (GMLAN Diagnostics Sub-Group)
E	FEB 2010	Version 1.6. (GMLAN Diagnostics Sub-Group)

Appendix A: Device Control Limits Exceeded Return Code Definition

A1 The GM Service and Parts Operations (SPO) Web Page.

This web page contains the list of standardized 2-byte values which are appended to the $7F negative response message when the return code is $E3 (Device Control Limits Exceeded).

Users should refer to this web page to obtain the complete and current list. Users must use the request form available at this web page to request new return codes.

Appendix B: Corporate Common CPID Definitions

The following section contains the list of CPID(s) and functionality which has been standardized for all nodes. A node which implements any of the corporate common CPIDs shall use the corporate common CPID number assigned and adhere to the bit and byte level positions defined. Corporate common CPIDs shall always contain a fixed number of bytes (the number may vary from one common CPID to another) even though a device may choose to not implement all of the functions defined for that CPID. Nodes **shall not** add additional device control features into a corporate common CPID.

B1 CPID $FE

Table B1: Corporate CPID $FE Definition

Control Byte #	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1	Theft Deterrent Relearn 1 = ENABLE 0 = DISABLE	Theft Deterrent EEPROM Access 1 = ENABLE 0 = DISABLE						
2								
3								
4								
5								

B1.1 Theft Deterrent Relearn. A scan tool implementation of this device control allows the user to request that the Powertrain or Engine Control Module (PCM/ECM) relearn the new password from the vehicle theft deterrent controller. Upon receipt of this device control, the PCM/ECM shall verify that none of the device control limits are exceeded and begin a ten minute timer. The scan tool is not required to remain connected for the next ten minutes. Therefore, the PCM software shall not expect a Mode $3E message to be sent every P3$_C$ ms. The PCM/ECM must remain powered (ignition must not be cycled to OFF) for the next ten minutes. When the ten minutes have expired, the PCM/ECM will be armed to learn the new password on the next powerup.

This device control will only be allowed if all the following are TRUE:

1. PCM/ECM is unsecured (unlocked) or reject with ($7F $AE $31) message.
2. Engine is not running or reject with ($7F $AE $E3 $FE $01); $FE $01 is the limit exceeded.

B1.2 Theft Deterrent EEPROM Access. Implementation of this device control allows a tool user to read/modify EEPROM variables within the vehicle theft deterrent controller. This device control requires that the theft deterrent controller be unlocked (reference security access service $27) at the time of the request for activation of this device control. Upon receipt of a valid request for this device control, the device will start a 10 minute timer. When the 10 minute timer is expired, the device can read/modify EEPROM variables within the theft deterrent controller using the diagnostic services $3B and $1A. A TesterPresent message is required to keep this device control active.

If the device is secure at the time a request for this device control is received, the device shall reject the request with a $7F $AE $31 response.

If a TesterPresent timeout occurs while this device control is active, the device shall terminate this device control but no negative response is sent. The device will however send the unsolicited mode $20 response message when the timeout occurs.

B2 CPID $FD

Table B2: Corporate CPID $FD Definition

Control Byte #	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
1	Disable All System Outputs 1 = ENABLE 0 = DISABLE	ECU Reset 1 = ENABLE 0 = DISABLE	Learned Source ID Reset 1 = ENABLE 0 = DISABLE					
2								
3								
4								
5								

B2.1 Disable All System Outputs. A scan tool implementation of this device control allows the user to deactivate all output devices of an ECU simultaneously or, if this is not possible, freeze the output(s) at its (their) current value. This prevents random behavior or periodic switching of loads during a current measurement. Upon receipt of this device control, the ECU shall verify that none of the device control limits are exceeded.

When this device control is active, all device controls assigned in other CPIDs for that ECU can be activated/deactivated independently by the other CPID (overrides CPID $FD for that output).

This device control will only be allowed if all the following are TRUE:

1. Vehicle Speed = 0.
2. Engine is not running.

A TesterPresent ($3E) message is required to keep this device control active.

B2.2 ECU Reset. Implementation of this device control allows a tool user to force the ECU to restart the application program (soft reset). Previously learned configuration data, adaptive factors, and other long-term adjustments shall not be reinitialized. The actual performed action is implementation specific.

This device control will only be allowed if all the following are TRUE:

1. Vehicle Speed = 0.
2. Engine is not running.

The ECU must respond to the request prior to performing the reset.

B2.3 Learned Source ID Reset. Implementation of this device control allows a tool user to force the ECU relearn of all previously learned Source IDs. This is accomplished by resetting the learned Source IDs as follows:

- Required messages that are always guaranteed to be present in the system shall be associated with Source ID $FF.
- Optional messages whose presence is determined by the presence of a build option shall be associated with Source ID $FE.

The module’s handler shall relearn Source IDs for any message that contains supervised signals each time the message is received. Only the most recently received Source ID shall be associated with each message that contains supervised signals. Support of this device control allows ECU swapping between vehicles. Otherwise, ECU swapping can only occur when all build options are identical between the two vehicles or false Lost Communication DTCs will occur. If this device control is used when an ECU is not communicating (valid Lost Communication DTC), the bus speed specific CAN Communication Bus Performance DTC will set rather than the ECU-specific Lost Communication DTC.

Appendix C: Corporate Standard Data Identifiers (DIDs)

This Appendix contains a list of all corporate standard DIDs. Corporate standard DID numbers shall occupy the range from $80 to $FE (application specific DIDs shall use the range of numbers from $01 to $7F).

The use of DIDs provides a mechanism for tools to access predefined data within an ECU without having to know the actual ECU memory address where the data is stored. DIDs can be read only, or have read and write access. The $1A service is used to read DID information and the $3B service is used for writing DID information into an ECU. It is recommended that a tester use physical addressing when requesting a DID which results in a multiple frame response message, unless the tester can send the flow control frame(s) to each responding ECU before a network layer timeout occurs.

The tables below are structured to provide a user the needed information to correctly implement a corporate standard DID that is applicable to a specific project. Which corporate standard DIDs apply to a given project must be agreed upon by the Release Engineer and representatives from Service Operations and Manufacturing (e.g., DVT and In Process Tester Development Engineers). The columns within the tables contain the following information:

- The hex representation of the DID value.
- A description of each data parameter for a given DID.
- Whether the DID can be read (R), written (W), or both (R/W) in the ECU.
- An * after the (R) or the (W) in the R/W field indicates that SPS security is required for reading (R*), writing (W*), or both (R*/W*). Reference the global pseudo code variable Security_Access_Unlocked set to a value of TRUE in paragraph 8.8.6.2.
- The format of the data for a given DID (ASCII = American Standard Code for Information Interchange, BCD = Binary Coded Decimal, USN = Unsigned Numeric, BIN = Binary or bit encoded).
- The byte length (Len) of the parameter data.

Table C1: Definition of Vehicle Identification Number dataIdentifier

<table>
<tr><th>DID</th><th colspan="11">Description</th><th colspan="2">R/W</th><th colspan="3">Data Type</th><th colspan="2">Len</th></tr>
<tr><td rowspan="5">$90</td><td colspan="11"><b>VehicleIdentificationNumber (VIN)</b></td><td colspan="2">R/W$_1$</td><td colspan="3">ASCII</td><td colspan="2">17</td></tr>
<tr><td colspan="18">This DID contains the Vehicle Identification Number. The data content shall be specified by the vehicle manufacturer. If the VIN has not yet been programmed into the ECU permanent memory, those locations shall be either “$00” or “$FF”. All 17 digits must be programmed in order for the ECU to use this DID. If the ECU does not store all 17 digits, then the ECU shall use an application specific DID to store the VIN digits.
<b>W$_1$ =</b> Some Nodes may restrict the ability to write to this DID under certain operating conditions (e.g., based on security status). All write restrictions placed on this DID shall be agreed upon by the DRE, GM service, and GM manufacturing representatives. The write restrictions must be documented in the CTS/SSTS or other ECU specific diagnostic document referenced by the CTS or SSTS<b>.</b></td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>14</td><td>15</td><td>16</td><td>17</td></tr>
<tr><td>VIN</td><td>W</td><td>0</td><td>L</td><td>0</td><td>0</td><td>0</td><td>0</td><td>3</td><td>6</td><td>T</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>8</td><td>5</td></tr>
<tr><td>Hex value</td><td>57</td><td>30</td><td>4C</td><td>30</td><td>30</td><td>30</td><td>30</td><td>33</td><td>36</td><td>54</td><td>31</td><td>30</td><td>30</td><td>30</td><td>30</td><td>38</td><td>35</td></tr>
</table>

Table C2: Definition of System Supplier Id dataIdentifier

DID	Description	R/W	Data Type	Len
$92	**SystemSupplierId (SYSSUPPID)**	R	ASCII	9

This DID contains SystemSupplierId information which identifies the ECU hardware supplier and the system type (System Group Identifier). The SYSSUPPID also contains a 2-character reserved field for future definition. The SYSSUPPID must be stored in the ECU in a **non-erasable memory area (e.g., boot sector of flash, or EEPROM)**. It shall not be possible to change the SYSSUPPID during a programming session or via normal diagnostic services. The SYSSUPPID consists of a structured sequence of nine ASCII characters which conform to GM-SPS (GM Service Programming System) requirements.

1. System Supplier Identifier (5 ASCII characters) (e.g., GM Powertrain = GMPT, DELCO = DELCO; SIEMENS = SIEM; BOSCH = BOSCH)
   Note: For GM Powertrain and Siemens, the 5th ASCII character in this field shall be a space ($20).
2. System Group Identifier (2 ASCII characters):
   00=Powertrain, 01 = Engine, 02 = Transmission, 03 = Chassis, 04 = Body, all others reserved by document (RBD).
3. Reserved area (Two ASCII characters)

Position	1	2	3	4	5	6	7	8	9	…
SYSSUPPID	D	E	L	C	O	0	1	0	0	…
Hex Value	44	45	4C	43	4F	30	31	30	30	…
Data Type	SystemSupplierIdentifier					System-Grouped		Reserved		…

Table C3: Definition of System Name OrEngineType dataIdentifier

DID	Description	R/W	Data Type	Len
$97	**SystemNameOrEngineType (SNOET)**	R/W	ASCII	≤ 20

This DID contains the SystemNameOrEngineType which identifies the electronic system name (e.g., Automatic Transmission with 2 liter engine: GS820 X20XEV) or engine type (X30XE) installed. The data shall always be of type ASCII. The length is variable (depends on length of string) but shall not exceed 20 characters.

Position	1	2	3	4	5	6	7	8	9	10	11	12	…
SNOET	G	S	8	2	0		X	2	0	X	E	V	…
Hex Value	47	53	38	32	30	20	58	32	30	58	45	56	…

Table C4: Definition of Repair Shop Code Or Tester Serial Number dataIdentifier

DID	Description	R/W	Data Type	Len
$98	**RepairShopCodeOrTesterSerialNumber (RSCOTSN)**	R/W	ASCII	10

This DID contains the RepairShopCodeOrTesterSerialNumber which identifies the dealer code or the tester serial number. This ASCII string is programmed into the ECU memory during the last programming session (SPS) at the dealer site.

Position	1	2	3	4	5	6	7	8	9	10	…
RSCOSN	S	N	1	2	3	4	5	6	7	8	…
Hex Value	53	4E	31	32	33	34	35	36	37	38	…

Table C5: Definition of Programming Date dataIdentifier

<table>
<tr><th>DID</th><th colspan="6">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$99</td><td colspan="6"><b>ProgrammingDate (PD)</b></td><td>R/W</td><td>BCD</td><td>4</td></tr>
<tr><td colspan="9">This DID contains the ProgrammingDate of the last programming event (SPS) at the dealer site. This BCD value was programmed into the ECU memory during the last programming session (SPS).</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td colspan="4">Description</td></tr>
<tr><td>PD (BCD)</td><td>20</td><td>00</td><td>01</td><td>22</td><td colspan="4">2000 = Year; 01 = Month; 22 = Day</td></tr>
<tr><td>Hex Value</td><td>20</td><td>00</td><td>01</td><td>22</td><td colspan="4"></td></tr>
</table>

Table C6: Definition of Diagnostic Data Identifier dataIdentifier

<table>
<tr><th>DID</th><th colspan="4">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$9A</td><td colspan="4"><b>DiagnosticDataIdentifier (DDI)</b></td><td>R/W*</td><td>USN</td><td>2</td></tr>
<tr><td colspan="7">This DID contains the DiagnosticDataIdentifier which identifies a supplier and system specific diagnostic data stream and shall be used by diagnostic testers to interpret a diagnostic data stream. The data format of the DDI is defined as two bytes, where the most significant byte of the DDI identifies the system by a system code and the least significant byte identifies the system data stream version number.
System Code (DDI high byte) shall be used to distinguish between the following cases:
• Different ECU systems, where the ECU uses the same Diagnostic Address (see DID $B0 Table for definition).
• Different system suppliers of the same component.
• Non-compatible ECU hardware and/or non-compatible software.
Diagnostic Data Stream Version Number (DDI low byte) shall be incremented when:
• The diagnostic implementation is changed and the changes affect Service/Aftersales and/or Manufacturing diagnostic test tools.
For an SPS_TYPE_B or SPS_TYPE_C ECU, the value of the least significant byte of the DDI shall always be $00 (provided that the DID is supported in a node which is not completely programmed).</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td colspan="4">Description</td></tr>
<tr><td>DDI</td><td>02</td><td>03</td><td colspan="4">System Code, e.g., $02 = Simtec MS56.5</td></tr>
<tr><td>Hex Value</td><td>02</td><td>03</td><td colspan="4">Diagnostic Data Stream Version Code, e.g., $03 = version 03</td></tr>
</table>

Table C7: Definition of XML Configuration Compatibility Identifier dataIdentifier

<table>
<tr><th>DID</th><th colspan="4">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$9B</td><td colspan="4"><b>XmlConfigurationCompatibilityIdentifier (XMLCCID)</b></td><td>R</td><td>USN</td><td>2</td></tr>
<tr><td colspan="7">This DID contains the XmlConfigurationCompatibilityIdentifier. This DID shall be supported if an ECU performs option configuration via a released XML file. The tool must read this DID and ensure that the data read matches the value of the XMLCCID embedded in the XML file before performing any DID writes dictated by the XML configuration file. This ensures that the configuration data within the XML file is compatible with the software/hardware within the ECU.</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td colspan="4">Description</td></tr>
<tr><td>XMLCCID</td><td colspan="2">44166</td><td colspan="4"></td></tr>
<tr><td>Hex Value</td><td>AC</td><td>86</td><td colspan="4"></td></tr>
</table>

Table C8: Definition of XML Data File Part Number dataIdentifier

<table>
<tr><th>DID</th><th colspan="5">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td>$9C</td><td colspan="5"><b>XmlDataFilePartNumber (XMLDFPN)</b></td><td>R/W</td><td>USN</td><td>4</td></tr>
<tr><td></td><td colspan="8">This DID is used to identify the part number of the released XML data file that is used to configure an ECU. Converting the 4-byte USN value to decimal provides the 8-digit part number assigned by the division. If an Alpha Code is used, then the Alpha Code associated with this part number shall be stored in dataIdentifier $9D.</td></tr>
<tr><td></td><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td colspan="3">…</td></tr>
<tr><td></td><td>XMLDFPN</td><td colspan="4">16265965</td><td colspan="3">…</td></tr>
<tr><td></td><td>Hex value</td><td>00</td><td>F8</td><td>32</td><td>ED</td><td colspan="3">…</td></tr>
</table>

Table C9: Definition of XML Data File Alpha Code dataIdentifier

<table>
<tr><th>DID</th><th colspan="4">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td>$9D</td><td colspan="4"><b>XmlDataFileAlphaCode (XMLDFAC)</b></td><td>R/W</td><td>ASCII</td><td>2</td></tr>
<tr><td></td><td colspan="7">This DID contains the 2-character representation of the Alpha Code (or Design Level suffix) associated with the XMLDataFilePartNumber (stored in dataIdentifier $9C).</td></tr>
<tr><td></td><td>Position</td><td>1</td><td>2</td><td colspan="4">---</td></tr>
<tr><td></td><td>XMLDFAC</td><td colspan="2">RS</td><td colspan="4">---</td></tr>
<tr><td></td><td>Hex value</td><td>52</td><td>53</td><td colspan="4">---</td></tr>
</table>

Table C10: Definition of PreviousStoredRepairShopCodeOrTesterSerialNumbers dataIdentifier

<table>
<tr><th>DID</th><th colspan="11">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td>$9F</td><td colspan="11"><b>PreviousStoredRepairShopCodeOrTesterSerialNumbers (PSRSCOTSN)</b></td><td>R</td><td>ASCII</td><td>20</td></tr>
<tr><td></td><td colspan="14">This DID contains the PreviousStoredRepairShopCodeOrTesterSerialNumbers. This data provides a mechanism for a tool user to identify up to the last two dealer codes or tester serial numbers programmed into DID $98. All bytes in this DID shall be initialized by the ECU supplier to $20 (ASCII = space).
<b>Last Stored Repair Shop Codes or Tester Serial Number (Bytes #1 thru #10)</b>
When a tool writes a dealer code or tester serial number into DID $98, the data previously stored in DID $98 is copied (without additional tool intervention) into the module memory housing byte positions #1 through #10 of this DID.
<b>Second from Last Stored Repair Shop Codes or Tester Serial Number (Bytes #11 thru #20)</b>
When a tool writes a dealer code or tester serial number into DID $98, the data previously in byte positions #1 through #10 of this DID are moved to positions #11 through #20. The data previously stored in DID $98 are then copied into this DID in byte positions #1 through #10.</td></tr>
<tr><td></td><td>1 Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td colspan="3">…</td></tr>
<tr><td></td><td>LSRSCOSN</td><td>S</td><td>N</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td colspan="3">…</td></tr>
<tr><td></td><td>Hex Value</td><td>58</td><td>31</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td><td>38</td><td colspan="3">…</td></tr>
<tr><td></td><td>2 Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td colspan="3"></td></tr>
<tr><td></td><td>SLSRSCOSN</td><td>S</td><td>N</td><td>3</td><td>3</td><td>3</td><td>8</td><td>5</td><td>9</td><td>9</td><td>7</td><td colspan="3"></td></tr>
<tr><td></td><td>Hex Value</td><td>58</td><td>33</td><td>33</td><td>33</td><td>33</td><td>38</td><td>37</td><td>39</td><td>39</td><td>37</td><td colspan="3"></td></tr>
</table>

Table C11: Definition of Manufacturers Enable Counter dataIdentifier

<table>
<tr><th>DID</th><th>Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="2">$A0</td><td><b>manufacturers_enable_counter (MEC)</b></td><td>R/W$_2$</td><td>USN</td><td>1</td></tr>
<tr><td colspan="4">This DID contains the MEC, which is used when determining the current status of ECU security. Refer to paragraph 9.3.2.6.2 for a description of the manufacturers_enable_counter and for additional requirements surrounding its implementation.
$00: ECU is locked (security system armed)
$FE $01: ECU is unlocked (security system not armed)
$FF: For use during a device's development cycle. Usage of this value shall be specified in the CTS/SSTS or other ECU specific diagnostic document referenced by the CTS or SSTS
<b>W$_2$</b>= A node shall not allow the value of the MEC to change once it becomes $00 unless SecurityAccess ($27) is successfully initiated (Security_Access_Unlocked is set to TRUE). Whether or not an ECU is allowed to write to this DID after it becomes $00 shall be agreed upon by the DRE, GM service, and GM manufacturing representatives. Agreement to support modifying this DID after it becomes $00 must be documented in the CTS/SSTS or other ECU specific diagnostic document referenced by the CTS or SSTS<b>.</b> The Security_Access_Unlocked flag is not set to FALSE when the MEC value is written to $00. The Security Status changes upon evaluation of the MEC and Vulnerability Flag. The Security_Access_Unlocked flag is evaluated upon receipt of a SecurityAccess ($27) request. The Security_Access_Unlocked flag is set to FALSE upon exiting the SecurityAccess mode via a ReturnToNormalMode ($20) request or a P3$_C$ timeout.
<table>
<tr><td>Position</td><td>1</td><td>Description</td></tr>
<tr><td>MEC</td><td>127</td><td></td></tr>
<tr><td>Hex value</td><td>7F</td><td></td></tr>
</table>
</td></tr>
</table>

Table C12: Definition of ECU Configuration or Customization Data dataIdentifier

<table>
<tr><th>DID</th><th>Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="2">$A1
thru
$A8</td><td><b>ECUConfigurationOrCustomizationData (ECUCOCGD)</b></td><td>R/W$_3$</td><td>User optional</td><td>≥ 1</td></tr>
<tr><td colspan="4">DIDs ranging from $A1 thru $A8 contain data specific to the configuration of an ECU (e.g., option content data) or operating parameters (e.g., Engine Oil Life Remaining, Long Term Fuel Trim values, Transmission Adaptive Learn values, etc.) which would need to be read from an original ECU and transferred to a replacement ECU. It is the responsibility of system supplier(s) and vehicle manufacturer to define the data that can be read and written via these DIDs. The main usage of these DIDs is to provide a common set of identifiers to be used to retrieve configuration data from an ECU
DID numbers in this range shall be assigned to an ECU in ascending order. This allows a tool to retrieve all ECU configuration data by first requesting DID $A1, and continuing until either DID $A8 is retrieved, or a negative response is sent to the tool with response code $31 (requestOutOfRange).
The overall size of the ECU Configuration Data (maximum of 8 DIDs) for a single ECU shall not exceed 512 Bytes in total.
<b>W$_3$</b> = Depending on the information being written, security may be required.
<table>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>…</td></tr>
<tr><td>OPTCODE</td><td colspan="3">4293845</td><td>…</td></tr>
<tr><td>Hex Value</td><td>41</td><td>84</td><td>D5</td><td>…</td></tr>
</table>
</td></tr>
</table>

Table C13: Definition of ECU Diagnostic Address dataIdentifier

<table>
<tr><th>DID</th><th colspan="3">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B0</td><td colspan="3"><b>ECUDiagnosticAddress (ECUADDR</b>)</td><td>R/W$_4$</td><td>USN</td><td>1</td></tr>
<tr><td colspan="6">This DID contains the diagnostic address of an ECU. This DID shall be supported by all ECUs when executing operational software and by SPS_TYPE_B ECUs when executing boot software.
<b>W$_4$</b> = This DID is usually read only. An ECU may choose to allow this DID to be written when more than one of a given hardware device can exist on a single vehicle. If this value is allowed to be written, then both the operational software and the boot shall be capable of updating the value used.</td></tr>
<tr><td>Position</td><td>1</td><td colspan="4">…</td></tr>
<tr><td>ECUADDR</td><td>17</td><td colspan="4">ECM ECUDiagnosticAddress</td></tr>
<tr><td>Hex Value</td><td>11</td><td colspan="4">…</td></tr>
</table>

Table C14: Definition of ECU Functional Systems and Virtual Devices dataIdentifier

<table>
<tr><th>DID</th><th colspan="5">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B1</td><td colspan="5"><b>ECUFunctionalSystemsAndVirtualDevices (ECUFSAVD)</b></td><td>R</td><td>USN</td><td>≥ 2</td></tr>
<tr><td colspan="8">This DID contains the list of functional systems and virtual devices supported by the node (ECU). The two lists are separated by a $FE separation character (AllNode functional system address). If an ECU does not support any VD (because it does not support any VN) then it shall report a $00 following the $FE to indicate that no VD is supported. The length of the DID depends on the number of supported functional systems and virtual devices.</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td colspan="4">…</td></tr>
<tr><td>ECUFSAVD</td><td>253</td><td>254</td><td>0</td><td colspan="4">…</td></tr>
<tr><td>Hex Value</td><td>FD</td><td>FE</td><td>00</td><td colspan="4">The $FD functional system corresponds to gateway devices</td></tr>
</table>

Table C15: Definition of Manufacturing Data dataIdentifier

<table>
<tr><th>DID</th><th colspan="7">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B2</td><td colspan="7"><b>GM ManufacturingData (GMMD)</b></td><td>R/W</td><td>USN</td><td>5</td></tr>
<tr><td colspan="10">This DID contains five bytes of ManufacturingData which is both written and read by tooling in the vehicle assembly plant during the production process. This area in product memory is used as a scratch pad for assembly tooling to pass test results back and forth from the various testers used in the vehicle assembly process. The ECU memory used for this DID shall be permanent memory (e.g., flash or EEPROM).</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td colspan="4">…</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="4">…</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="4">…</td></tr>
</table>

Table C16: Definition of Data Universal Numbering System Identification

<table>
<tr><th>DID</th><th colspan="11">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B3</td><td colspan="11"><b>Data Universal Numbering System Identification (DUNS)</b></td><td>R</td><td>ASCII</td><td>9</td></tr>
<tr><td colspan="14">This DID contains the data necessary to meet the component traceability requirements as specified in GMW4710 and GMW15862. This DID contains the DUNS Id assigned to the component supplier’s manufacturing site.</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td colspan="4">---</td></tr>
<tr><td>DUNS</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td colspan="4">---</td></tr>
<tr><td>Hex value</td><td>31</td><td>32</td><td>33</td><td>34</td><td>35</td><td>36</td><td>37</td><td>38</td><td>39</td><td colspan="4">---</td></tr>
</table>

Table C17: Definition of Manufacturing Traceability Characters dataIdentifier

DID	Description	R/W	Data Type	Len
$B4	**Manufacturing Traceability Characters (MTC)**	R	ASCII	16

This DID contains the data necessary to meet the component traceability requirements as specified in GMW4710 and GMW15862. The length of this DID is variable but shall not exceed 16 characters. These fields are populated at the time of component manufacturing by the manufacturer. The ECU specific CTS or supplemental diagnostic specification referenced by the CTS or SSTS must document how the fields are to be populated.

Legacy Format (To be phased out per GMW15862) A description of each field follows:

Component Identifier (Comp Id)

The component Identifier is a 2 byte field (starting at byte 1 of the DID) that is used to indicate the component type.

Part Number / Broadcast Code (PN/BC)

This is a 4-byte field (starting at byte 3 of the DID) and contains either the ECU Broadcast Code or the last 4 digits of the ECU part number.

Supplier Code (sup)

This is a one byte field occupying the 7th byte of the DID that identifies the supplier and manufacturing location of the ECU.

Traceability Number

This field starts at byte 8 of the DID and is 9 bytes in length. The content is assigned by the module manufacturer. This field contains information about the ECU at the time of its manufacture. Examples of the type of data included in this field are Julian Date of production, shift build data, production assembly line information and ECU serial number. If less than 9 characters are required to uniquely identify an ECU, then the leading characters shall be filled with zeroes (0x30).

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	---
MTC	A	S	7	0	8	1	K	1	5	0	6	1	0	0	0	F	---
Hex value	41	53	37	30	38	31	4B	31	35	30	36	31	30	30	30	46	---
Field Desc.	Comp Id		PN/BC				sup	Traceability Number									---

GMW15862 Format (To be phased in per GMW15862) A description of each field follows:

Line Identification (Line Id)

The Line Identifier is a 1 byte field (starting at byte 1 of the DID) that is used to indicate the component assembly line.

Shift Identification 1, 2 or 3 (Shift Id)

This is a one byte field (starting at byte 2 of the DID) that identifies the shift that assembled the component.

Last Two Digits of Year (Year)

This is a two byte field (starting at byte 3 of the DID) that contains the last two digits of the year that the component was assembled.

Day Of the Year (Day)

This is a three byte field (starting at byte 5 of the DID) that contains the Julian day of the Year that the component was assembled.

Traceability Number

This value is a supplier assigned serial, lot or batch number. This field starts at byte 8 of the DID and is up to 9 bytes in length. The content is assigned by the module manufacturer. If less than 9 characters are required to uniquely identify an ECU, then the leading characters shall be filled with zeroes (0x30).

Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	---
MTC	A	1	0	8	2	8	2	1	5	0	6	1	0	0	0	F	---
Hex value	41	31	30	38	32	38	32	31	35	30	36	31	30	30	30	46	---
Field Desc.	Line	Shift	Year		Day			Traceability Number									---

Table C18: Definition of Broadcast Code dataIdentifier

<table>
<tr><th>DID</th><th colspan="5">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B5</td><td colspan="5"><b>GM BroadcastCode (GMBC)</b></td><td>R/W</td><td>ASCII</td><td>4</td></tr>
<tr><td colspan="8">This DID contains four alpha characters which represent the ECU broadcast code. The broadcast code is unique for each combination of hardware, software, and calibrations for an ECU.</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td colspan="3">…</td></tr>
<tr><td>Broadcast Code</td><td>A</td><td>B</td><td>C</td><td>D</td><td colspan="3">…</td></tr>
<tr><td>Hex value</td><td>41</td><td>42</td><td>43</td><td>44</td><td colspan="3">…</td></tr>
</table>

Table C19: Definition of GM Target Vehicle Data dataIdentifier

<table>
<tr><th>DID</th><th colspan="7">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="5">$B6</td><td colspan="7"><b>GM Target Vehicle (GMTV)</b></td><td>R/W$_5$</td><td>USN/ASCII</td><td>5</td></tr>
<tr><td colspan="10">This DID contains five bytes of data used to identify the vehicle that the ECU is targeted for. The data is broken down as follows:
<b>Byte 1 (Model Year)</b>
This byte is an USN value that represents the target model year using the year 1900 as the base year. To represent the 2000 model year, the value of byte 1 would be 100 or ($64). This is derived from 1900 + 100 = 2000. If the module supports multiple model years, then this byte is filled in with $23 (ASCII equivalent to “#”).
<b>Byte 2 (Vehicle Make)</b>
This byte is the ASCII representation of the Vehicle Make as Identified in the Vehicle Identification Number (VIN). For example, Cadillac = 6 or $36. If the part is used in multiple vehicle makes, then a value of $23 (ASCII “#”) is used.
<b>Byte 3 (Vehicle Car Line)</b>
This byte is an ASCII character representing the vehicle car line (e.g., <b>J</b> for Chevy Cavalier). If the module can be used in multiple car lines, then a value of $23 (ASCII “#”) is used.
<b>Bytes 4 and 5 (Engine RPO)</b>
These two bytes contain the ASCII representation of the last two characters of the engine Regular Production Option (RPO). The first character of an engine RPO is constant and is therefore not stored. For the L67 engine, the value for these two bytes would be “6” = $36, and “7” = $37. If the module is to be used with multiple engines, then a $23 (ASCII “#”) is put in both bytes. If the module can be used with any engine type, then $2A (ASCII “*”) is put in both bytes.
<b>W$_5$ =</b> This DID may be written with service $3B at the discretion of the platform.</td></tr>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td colspan="4">---</td></tr>
<tr><td>GMTV</td><td>2002</td><td>1</td><td>J</td><td>D</td><td>9</td><td colspan="4">For 2002 Chevy Cavalier with LD9 engine</td></tr>
<tr><td>Hex value</td><td>66</td><td>31</td><td>4A</td><td>44</td><td>39</td><td colspan="4">---</td></tr>
</table>

Table C20: Definition of GM Software Usage Description Data dataIdentifier

DID	Description	R/W	Data Type	Len
$B7	**GM Software Usage Description (GMSUD)**	R/W$_6$	USN/ASCII	5

This DID contains five bytes of data used to identify the intended usage of ECU development software. The data is broken down as follows:

Byte 1 (Program Phase)

This byte is contains an ASCII value that represents the program phase that the software is target for (e.g., B = Beta, G = Gamma, etc).

Byte 2 (Software Release Type/Calibration Release Type)

This byte is stored in USN format where the upper nibble represents the software release type and the lower nibble represents the calibration release type. A value of $D = development, and $A = approved for release by supplier.

Byte 3 (Software Release Number)

This byte is stored in USN format and represents the release number of the ECU software set. Software revision numbers shall start at $01 and increment with each release.

Bytes 4 and 5 (PROM Identifier)

These two bytes are stored in USN format and are used to provide a unique representation of a software version (e.g., 2000 NAO V6 PCM with level 2400 software = $09 $60).

W$_6$ = This DID may be written with service $3B at the discretion of the platform.

Position	1	2	3	4	5	For Beta release with approved software and calibrations. 2nd software release and software revision level = 2400
GMSUD	B	AA	2	2400		
Hex value	42	AA	02	09	60	

Table C21: Definition of GM Bench Verification Information Data dataIdentifier

DID	Description	R/W	Data Type	Len
$B8	**GM Bench Verification Information (GMBVI)**	R/W$_7$	USN/ASCII	5

This DID contains five bytes of data used to provide information about bench verification during the development process. This DID is made up of the following components:

Byte 1 and 2 (Verification Initials)

These two bytes are the ASCII representation of the first and last initials of the individual who authorized this software release by checking a sample version on the software integration bench.

Bytes 3, 4 and 5 (Verification Date)

The data stored in bytes 3, 4, and 5 are saved in USN format and represent the date that the bench verification took place. The data is arranged in the following format: YY/MM/DD, where the Year (YY) will use 1900 as a base and be stored in byte 3. Month (MM) is stored in byte 4, and the day (DD) is stored in byte 5.

W$_7$ = This DID may be written with service $3B at the discretion of the platform.

Position	1	2	3	4	5	…
GMBVI	J	D	2001	2	3	John Doe performed Bench Verification on Feb 3rd, 2001
HEX	4A	44	65	02	03	…

Table C22: Definition of Subnet Config List High Speed dataIdentifier

DID	Description	R/W	Data Type	Len
$B9	**Subnet_Config_List_HighSpeed (SCLHS)**	R/W	BIN	≥ 1
	This DID contains the SCLHS. Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the high speed subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification. **Example:** ECM Engine Control Module			

Position	1			2			
Bit location	7	6 to 1	0	7	6 to 1	0	…
Bit state	0	…	1	0	…	0	…
DA (hex) J3200	…	…	11	…	…	…	…
ECU name	…	…	ECM	…	…	…	…
Hex value	01			00			

Table C23: Definition of Subnet Config List Low Speed dataIdentifier

DID	Description	R/W	Data Type	Len
$BA	**Subnet_Config_List_LowSpeed (SCLLS)**	R/W	BIN	≥ 1
	This DID contains the SCLLS. Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the low speed subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification. **Example:** PDM Passenger Door Module CIM Column Integration Module DDM Driver Door Module			

Position	1			2			3			4		
Bit location	7	6 to 1	0	7	6 to 1	0	7	6 to 1	0	7	6 to 1	0
Bit state	1	…	1	0	…	1	0	…	0	0	…	0
DA (hex) J3200	A4	…	40	---	…	A3	---	…	---	---	…	---
ECU name	PDM	…	CIM	---	…	DDM	---	…	---	---	…	---
Hex value	81			01			00			00		

Table C24: Definition of Subnet Config List Mid Speed dataIdentifier

<table>
<tr><th>DID</th><th colspan="5">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="9">$BB</td><td colspan="5"><b>Subnet_Config_List_MidSpeed (SCLMS)</b></td><td>R/W</td><td>BIN</td><td>≥ 1</td></tr>
<tr><td colspan="8">This DID contains the SCLMS. Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the mid speed subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.
<b>Example:</b> The example assumes that the ECC occupies bit 7 of byte 0 and the TMS occupies bit 0 of byte 0.
ECC Electronic Climate Control
TMS Telematic System</td></tr>
<tr><td>Position</td><td colspan="3">1</td><td colspan="3">2</td><td>…</td></tr>
<tr><td>Bit location</td><td>7</td><td>6 to 1</td><td>0</td><td>7</td><td>6 to 1</td><td>0</td><td>…</td></tr>
<tr><td>Bit state</td><td>1</td><td>…</td><td>1</td><td>0</td><td>…</td><td>0</td><td>…</td></tr>
<tr><td>DA (hex)
J3200</td><td>A0</td><td>…</td><td>91</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>ECU name</td><td>ECC</td><td>…</td><td>TMS</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>Hex value</td><td colspan="3">81</td><td colspan="3">00</td><td>…</td></tr>
</table>

Table C25: Definition of Subnet Config Lists for Non-CAN Buses 1 - 2 dataIdentifier

<table>
<tr><th>DID</th><th colspan="5">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td>$BC</td><td colspan="5"><b>Subnet_Config_List_NonCan 1 - 2 (SCLNC 1/2)</b></td><td>R/W</td><td>BIN</td><td>≥ 1</td></tr>
<tr><td rowspan="8">$BD</td><td colspan="8">These 2 DIDs are allocated to contain subnet config lists for data links not directly connected to the DLC but contain a member node which is diagnosed via a GMLAN subnet. The tool can write the configuration information of the non GMLAN network into the GMLAN node which is also present on the non GMLAN network, for the purpose of network management of the non GMLAN network. Examples of this may be a GMLAN node that is also an entertainment ECU that is part of a MOST network.
Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.
<b>Example:</b>
EHU Entertainment Head Unit (MOST)</td></tr>
<tr><td>Position</td><td colspan="3">1</td><td colspan="3">2</td><td></td></tr>
<tr><td>Bit location</td><td>7</td><td>6 to 1</td><td>0</td><td>7</td><td>6 to 1</td><td>0</td><td>…</td></tr>
<tr><td>Bit state</td><td>0</td><td>…</td><td>1</td><td>0</td><td>…</td><td>0</td><td>…</td></tr>
<tr><td>DA (hex)</td><td>…</td><td>…</td><td>81</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>ECU name</td><td>…</td><td>…</td><td>EHU</td><td>…</td><td>…</td><td>…</td><td>…</td></tr>
<tr><td>Hex value</td><td colspan="3">01</td><td colspan="3">00</td><td>…</td></tr>
</table>

Table C26: Definition of Subnet Config List for LIN Busses dataIdentifier

DID	Description	R/W	Data Type	Len
$BE	**Subnet_Config_List_LIN (SCLLIN)**	R/W_8	BIN	≥ 2

This DID contains the SCLLIN for all Local Interconnect Network (LIN) buses supported by the LIN bus Master. The DID shall include all LIN subnets that are connected to the Master. Two bytes shall be used per LIN network as described in the table below. The length of the DID will be determined by the number of connected LIN networks, i.e., length is equal to 2*n, where n is the number of connected LIN networks.

The tool can write the configuration information of the LIN network(s) into the GMLAN node which is also the master of the LIN network, for the purpose of network management.

The most significant bit of the two bytes for each LIN subnet shall indicate the LIN master configuration status (CS) for this subnet. A logic “1” indicates that the LIN master has been configured for this subnet.

The remaining bits each represent one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.

Example:

LIN bus 1 is configured.

DDM Driver Door Module

DDS Driver Door Switch

W_8 = The writing of this DID may not be allowed in certain implementations. The CTS/SSTS or other ECU specific diagnostic document referenced by the CTS or SSTS shall indicate if this DID can be written.

Position	1 LIN bus 1			2 LIN bus 1			3 LIN bus 2			4 LIN bus 2		
Bit location	7	6 to 1	0	7 to 2	1	0	7	6 to 1	0	7	6 to 1	0
Bit state	1	…	0	0	1	1	0	…	0	0	…	0
DA (hex)	N/A	…	---	---	A4	A0	N/A	…	---	---	…	---
ECU name	CS	…	---	---	DDS	DDM	CS	…	---	---	…	---
Hex value	80			03			00			00		
Position	2n -3 LIN bus n-1			2n -2 LIN bus n-1			2n - 1 LIN bus n			2n LIN bus n		
Bit location	7	6 to 1	0	7	6 to 1	0	7	6 to 1	0	7	6 to 1	0
Bit state	0	…	0	0	…	0	0	…	0	0	…	0
DA (hex)	N/A	…	---	---	…	---	N/A	…	---	---	…	---
ECU name	CS	…	---	---	…	---	CS	…	---	---	…	---
Hex value	00			00			00			00		

Table C27: Definition of Subnet Config Lists for GMLAN Expansion Buses dataIdentifier

DID	Description	R/W	Data Type	Len
$BF	**Subnet_Config_List_GMLANChassisExpansionBus (SCLGCEB)**	R/W	BIN	≥ 2
	This DID contains the SCLGCEB. Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the Chassis expansion bus subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification. **Example:** SAS Steering Angle Sensor			

Position	1			2			
Bit location	7	6 to 1	0	7	6 to 1	0	---
Bit state	0	…	0	0	…	1	---
DA (hex) J3200	---	…	---	---	…	11	---
ECU name	---	…	---	---	…	SAS	---
Hex value	00			01			---

Table C28: Definition of Boot Software Part Number dataIdentifier

DID	Description	R/W	Data Type	Len
$C0	**BootSoftwarePartNumber (BSPN)**	R	USN	4
	This DID contains BootSoftwarePartNumber information which is a 4-byte, unsigned numeric representation of the 8-digit GM part number assigned to the boot software of the ECU. If a Boot Software Alpha Code is used, then the Alpha Code associated with this part number shall be stored in dataIdentifier $D0.			

Position	1	2	3	4	---
BootSoftwarePartNumber	16265965				---
Hex value	00	F8	32	ED	---

Table C29: Definition of Software Module Identifier dataIdentifier

DID	Description	R/W	Data Type	Len
$C1 to $CA	**SoftwareModuleIdentifier (SWMI)**	R	USN or ASCII	4 or 5 to 16

DIDs ranging from $C1 to $CA contain Software Module Identifier (SWMI) information used to uniquely identify a given software or calibration module. The SWMI shall be included in each software module that can be separately programmed into the ECU permanent memory via the SPS. Individual software modules can contain Boot software, Operational software, or Calibration data.

The SWMI information shall be stored using one of two allowed formats. The first format consists of a four byte USN representation of the eight digit GM part number assigned to the software or calibration module. The second allowed format consists of an ASCII representation of either an assigned GM part number or the base name of the file as released by the module supplier. If the ASCII format is used, then the part number/file name shall contain a minimum of five characters and a maximum of 16.

The length of the data reported in the response message for a given SWMI is used by the tool to determine which of the two formats is being used. A given ECU shall use the same format for all supported SWMI.

If an Alpha Code (or Design Level Suffix) is supported, then this information shall be stored in ASCII format in a dataIdentifier with the upper nibble set to $D and the lower nibble the same as the lower nibble of the dataIdentifier that contains the corresponding SWMI. In other words, if a SWMI is stored in dataIdentifier $C1, then the corresponding Alpha Code (or Design Level Suffix) shall be stored in dataIdentifier $D1.

Each SWMI has a corresponding dataIdentifier associated with it. ECUs are only required to support the DIDs which correspond to SWMIs valid for that ECU. The relationship between the SWMI and DID number is as follows:

- $C1: SoftwareModule_01_Identifier
- $C2: SoftwareModule_02_Identifier
- $C3: SoftwareModule_03_Identifier
- $C4: SoftwareModule_04_Identifier
- $C5: SoftwareModule_05_Identifier
- $C6: SoftwareModule_06_Identifier
- $C7: SoftwareModule_07_Identifier
- $C8: SoftwareModule_08_Identifier
- $C9: SoftwareModule_09_Identifier
- $CA: SoftwareModule_10_Identifier

There are a maximum of ten (10) software modules possible with DIDs $C1 thru $CA. ECUs that have a combined number of software and calibration modules that exceed 10 shall also support DID $DD. DID $DD provides a way for the test tool to determine which DIDs contain the SWMI and SWMIAC for the 11th through the nth software/calibration module(s).

Position	1	2	3	4	
SWMI	16265965				
Hex value	00	F8	32	ED	

Table C30: Definition of End Model Part Number dataIdentifier

<table>
<tr><th>DID</th><th>Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="3">$CB</td><td><b>EndModelPartNumber</b></td><td>R/W$_9$</td><td>USN</td><td>4</td></tr>
<tr><td colspan="4">This DID is used to identify the part number that represents the combination of hardware/software/calibrations present in the ECU as it is received in the vehicle assembly plant. This part number is also used in service to uniquely identify the combination of hardware/software/calibrations programmed into the ECU at the time the part is ordered. Converting the four byte USN value to decimal provides the 8-digit part number assigned by the division. If an Alpha Code is used, then the Alpha Code associated with this part number shall be stored in dataIdentifier $DB.
<b>W$_9$</b> = This DID may or may not be updated at the conclusion of a programming event based on divisional practices. This DID is not required to be writeable if divisional practices do not require this part number to be updated after module programming.</td></tr>
<tr><td colspan="4"><table>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>---</td></tr>
<tr><td>EndModelPartNumber</td><td colspan="4">16265965</td><td>---</td></tr>
<tr><td>Hex value</td><td>00</td><td>F8</td><td>32</td><td>ED</td><td>---</td></tr>
</table></td></tr>
</table>

Table C31: Definition of Base Model Part Number dataIdentifier

<table>
<tr><th>DID</th><th>Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="3">$CC</td><td><b>BaseModelPartNumber (BMPN)</b></td><td>R</td><td>USN</td><td>4</td></tr>
<tr><td colspan="4">This DID contains BaseModelPartNumber information which is used during SPS programming to identify a unique combination of ECU hardware and all non-reprogrammable software (e.g., Boot software). The BMPN must be stored in the ECU in a non-erasable memory area (e.g., boot sector of flash, or EEPROM). It shall not be possible to change the BMPN as part of a programming session or through diagnostic services when a programming session is not active. The BMPN of a service replacement ECU shall be identical to that of a Production ECU if both ECUs have the same hardware and non-reprogrammable software. If an Alpha Code is used, then the Alpha Code associated with this part number shall be stored in dataIdentifier $DC.</td></tr>
<tr><td colspan="4"><table>
<tr><td>Position</td><td>1</td><td>2</td><td>3</td><td>4</td><td>---</td></tr>
<tr><td>BaseModelPartNumber</td><td colspan="4">16265965</td><td>---</td></tr>
<tr><td>Hex value</td><td>00</td><td>F8</td><td>32</td><td>ED</td><td>---</td></tr>
</table></td></tr>
</table>

Table C32: Definition of Boot Software Part Number Alpha Code dataIdentifier

<table>
<tr><th>DID</th><th>Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="3">$D0</td><td><b>BootSoftwarePartNumberAlphaCode</b></td><td>R</td><td>ASCII</td><td>2</td></tr>
<tr><td colspan="4">This DID contains the 2-character representation of the Alpha Code (or Design Level suffix) associated with the BootSoftwarePartNumber (stored in dataIdentifier $C0).</td></tr>
<tr><td colspan="4"><table>
<tr><td>Position</td><td>1</td><td>2</td><td>---</td></tr>
<tr><td>BootSoftwarePartNumberAlphaCode</td><td colspan="2">RS</td><td>---</td></tr>
<tr><td>Hex value</td><td>52</td><td>53</td><td>---</td></tr>
</table></td></tr>
</table>

Table C33: Definition of Software Module Identifier Alpha Code dataIdentifier

DID	Description	R/W	Data Type	Len
$D1 to $DA	**SoftwareModuleIdentifierAlphaCode (SWMIAC)**	R	ASCII	2
	DIDs ranging from $D1 to $DA contain a 2-character representation of the Alpha Code (or Design Level suffix) for the corresponding Software Module Identifier (SWMI). The SWMIs occupy the dataIdentifier range $C1 thru $CA. Reference the SWMI DID description for the numerical relationship between the DIDs used for the SWMI and the corresponding SWMIAC. • $D1: SoftwareModule_01_AlphaCode • $D2: SoftwareModule_02_AlphaCode • $D3: SoftwareModule_03_AlphaCode • $D4: SoftwareModule_04_AlphaCode • $D5: SoftwareModule_05_AlphaCode • $D6: SoftwareModule_06_AlphaCode • $D7: SoftwareModule_07_AlphaCode • $D8: SoftwareModule_08_AlphaCode • $D9: SoftwareModule_09_AlphaCode • $DA: SoftwareModule_10_AlphaCode The SoftwareModuleIdentifier Alpha Code shall be specified by the vehicle manufacturer.			

Position	1	2	
Alpha Code	RS		
Hex value	52	53	

Table C34: Definition of End Mode lPart Number Alpha Code dataIdentifier

DID	Description	R/W	Data Type	Len
$DB	**EndModelPartNumberAlphaCode**	R/W$_{10}$	ASCII	2
	This DID contains the 2-character representation of the Alpha Code (or Design Level suffix) associated with the EndModelPartNumber (stored in dataIdentifier $CB). **W$_{10}$** = This DID may or may not be updated at the conclusion of a programming event based on divisional practices. This DID is not required to be writeable if divisional practices do not require this part number to be updated after module programming.			

Position	1	2	---
EndModelPartNumberAlphaCode	RS		---
Hex value	52	53	---

Table C35: Definition of Base Model Part Number Alpha Code dataIdentifier

DID	Description	R/W	Data Type	Len
$DC	**BaseModelPartNumberAlphaCode**	R	ASCII	2
	This DID contains the 2-character representation of the Alpha Code (or Design Level suffix) associated with the BaseModelPartNumber (stored in dataIdentifier $CC).			

Position	1	2	---
BaseModelPartNumberAlphaCode	RS		---
Hex value	52	53	---

Table C36: Definition of Software Module Identifier dataIdentifiers

DID	Description	R/W	Data Type	Len
$DD	**SoftwareModuleIdentifierDataIdentifiers (SWMIDID)**	R	USN	2 to 40

If an ECU supports more than ten software/calibration modules, then this DID shall be used to retrieve the list of additional DIDs (in the application specific range) that the tester must retrieve in order to determine the SWMI and SWMIAC for the 11th through the nth software/calibration module(s).

The DID numbers for the additional modules shall be transmitted in pairs with the DID number for the SWMI transmitted first followed by the DID number for the SWMIAC.

The definition and the content (data type, length, format) of the data reported when a tester requests one of the DIDs provided in this list can be found in the definition of the corporate common Data Identifiers $C1thru CA (SWMIs of the first ten modules).

If an Alpha Code (or Design Level Suffix) is not supported, then the position byte for the alpha code shall contain a value of $00.

There are a maximum of twenty (20) additional software/calibration modules possible (30 total modules for an ECU including those provided with DIDs $C1 thru $CA). The relationship between the additional SWMI/SWMIAC and the application specific DID number is as follows:

- Position #1: SoftwareModule_11_DID
- Position #2: SoftwareModule_11_AlphaCode_DID
- Position #3: SoftwareModule_12_DID
- Position #4: SoftwareModule_12_AlphaCode_DID
- Position #5: SoftwareModule_13_DID
- Position #6: SoftwareModule_13_AlphaCode_DID
- Position #7: SoftwareModule_14_DID
- Position #8: SoftwareModule_14_AlphaCode_DID
- Position #9: SoftwareModule_15_DID
- Position #10: SoftwareModule_15_AlphaCode_DID

.
.
.

- Position #31: SoftwareModule_26_DID
- Position #32: SoftwareModule_26_AlphaCode_DID
- Position #33: SoftwareModule_27_DID
- Position #34: SoftwareModule_27_AlphaCode_DID
- Position #35: SoftwareModule_28_DID
- Position #36: SoftwareModule_28_AlphaCode_DID
- Position #37: SoftwareModule_29_DID
- Position #38: SoftwareModule_29_AlphaCode_DID
- Position #39: SoftwareModule_30_DID
- Position #40: SoftwareModule_30_AlphaCode_DID

Example: An ECU is comprised of 12 total software and calibrations modules. DIDs $C1 thru $CA are used to provide the SWMI and DIDs $D1 thru $DA the SWMIAC for the first 10 modules. The ECU stores the SWMI for the 11th module in DID $70, the SWMIAC for the 11 module in DID $71, the SWMI for the 12th module in DID $72, and the SWMIAC for the 12th module in DID $73. In this case, the data reported upon request for this DID would contain the values shown below:

Position	1	2	3	4	
SWMI/SWMIAC	112	113	114	115	
Hex value	70	71	72	73	

Table C37: Definition of GMLAN Identification Data

DID	Description	R/W	Data Type	Len
$DE	**GMLANIdentificationData (GMLANID)**	R	USN	≥ 6
	This DID contains a 1-byte Bus Type, a 2-byte GMLAN Kernel, and a 3-byte value that represents the revision level of the GM serial data database file implemented on the specified network. This information shall be repeated for each GMLAN network that the controller is connected to. The length of the DID will be determined by the number of connected GMLAN networks, i.e., length is equal to 6*n, where n is the number of connected GMLAN networks. **Bus Type:** This value indicates which GMLAN network the next five bytes refer to. The Bus Type is a one byte enumerated value. The Bus Type definition must be documented in a CTS, SSTS, or supplemental diagnostic specification. **GMLAN Kernel Version:** This is the GMLAN Handler version implemented by the ECU on the network specified by the preceding Bus Type. **GM Database Revision Level:** This is the GM Database Revision Level implemented by the ECU on the network specified by the preceding Bus Type.			

Position	1	2	3	4	5	6	
Description	Bus Type	GMLAN Kernel Version		GM Database Revision Level			
Value	HS	3.0		3.2.2			
Hex value	02	03	00	03	02	02	

Table C38: Definition of ECU Odometer Value

DID	Description	R/W	Data Type	Len
$DF	**ECUOdometerValue (ECUODO)**	R*/W*	USN	4
	This DID contains the Odometer value as stored by the ECU. The odometer value shall be stored with a scaling of E = N/64 km.			

Position	1	2	3	4	---
ECUOdometerValue	16265965				---
Hex value	00	F8	32	ED	---

Table C39: Definition of Vehicle Level Data Record

DID	Description	R/W	Data Type	Len
$E0 thru $E7	**VehicleLevelDataRecord (VLDR)**	R	Note 1	Note 1
	This range of DIDs is reserved for housing vehicle operating conditional data stored at an instant in time based on a triggering mechanism, with each DID representing a separate data record. These DIDs are intended to be held in a single ECU on the vehicle which would capture data transmitted on one or more of the vehicle serial data links based on a trigger message.			

Note 1: The contents and scaling of the data, as well as the triggering mechanism, are beyond the scope of this specification and if implemented, must be defined in the appropriate ECU CTS, SSTS, or other diagnostic specification.

Table C40: Definition of Subnet Config Lists for GMLAN Expansion Busses dataIdentifier

DID	Description	R/W	Data Type	Len
$E8	**Subnet_Config_List_GMLANPowertrainExpansionBus (SCLGPEB)**	R/W	BIN	≥ 2

This DID contains the SCLGPEB.

Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the Powertrain expansion bus subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.

Example:

HCP	Hybrid Control Processor
MCP1	Motor Control Processor 1
MCP2	Motor Control Processor 2

Position	1			2			
Bit location	7	6 to 1	0	7	6 to 1	0	---
Bit state	0	…	1	1	…	1	---
DA (hex) J3200	---	…	CE	CF	…	17	---
ECU name	---	…	MCP1	MCP2	…	HCP	---
Hex value	01			81			---

Table C41: Definition of Subnet Config Lists for GMLAN Expansion Busses dataIdentifier

DID	Description	R/W	Data Type	Len
$E9	**Subnet_Config_List_GMLANFrontObjectExpansionBus (SCLGFOEB)**	R/W	BIN	≥ 2

This DID contains the SCLGFOEB.

Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the expansion bus subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.

Example:

EOCM-R	External Object Control Module - Front

Position	1			2			
Bit location	7	6 to 1	0	7	6 to 1	0	---
Bit state	0	…	0	0	…	1	---
DA (hex) J3200	---	…	---	---	…	6C	---
ECU name	---	…	---	---	…	EOCM-F	---
Hex value	00			01			---

Table C42: Definition of Subnet Config Lists for GMLAN Expansion Busses dataIdentifier

<table>
<tr><th>DID</th><th colspan="8">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="8">$EA</td><td colspan="8">Subnet_Config_List_GMLANRearObjectExpansionBus (SCLGROEB)</td><td>R/W</td><td>BIN</td><td>≥ 2</td></tr>
<tr><td colspan="11">This DID contains the SCLGROEB.
Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the expansion bus subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.
Example:
EOCM-R External Object Control Module - Rear</td></tr>
<tr><td>Position</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="4"></td></tr>
<tr><td>Bit location</td><td>7</td><td>6 to 1</td><td>0</td><td>7</td><td>6 to 1</td><td>0</td><td colspan="4">---</td></tr>
<tr><td>Bit state</td><td>0</td><td>…</td><td>0</td><td>0</td><td>…</td><td>1</td><td colspan="4">---</td></tr>
<tr><td>DA (hex)
J3200</td><td>---</td><td>…</td><td>---</td><td>---</td><td>…</td><td>64</td><td colspan="4">---</td></tr>
<tr><td>ECU name</td><td>---</td><td>…</td><td>---</td><td>---</td><td>…</td><td>EOCM-R</td><td colspan="4">---</td></tr>
<tr><td>Hex value</td><td colspan="3">00</td><td colspan="3">01</td><td colspan="4">---</td></tr>
</table>

Table C43: Definition of Subnet Config Lists for GMLAN Expansion Busses dataIdentifier

<table>
<tr><th>DID</th><th colspan="8">Description</th><th>R/W</th><th>Data Type</th><th>Len</th></tr>
<tr><td rowspan="8">$EB
thru
$EF</td><td colspan="8">Subnet_Config_List_GMLANExpansionBus1-5 (SCLGEB1/2/3/4/5)</td><td>R/W</td><td>BIN</td><td>≥ 2</td></tr>
<tr><td colspan="11">These DIDs contain the SCLGEB1/2/3/4/5.
Each bit represents one ECU (with unique Diagnostic Address, see Appendix D) that can exist on the expansion bus subnet. A logic “1” indicates that a given ECU is present on the vehicle. The bit mapping of ECUs must be documented in a CTS, SSTS, or supplemental diagnostic specification.
Example:
SAS Steering Angle Sensor</td></tr>
<tr><td>Position</td><td colspan="3">1</td><td colspan="3">2</td><td colspan="4"></td></tr>
<tr><td>Bit location</td><td>7</td><td>6 to 1</td><td>0</td><td>7</td><td>6 to 1</td><td>0</td><td colspan="4">---</td></tr>
<tr><td>Bit state</td><td>0</td><td>…</td><td>0</td><td>0</td><td>…</td><td>1</td><td colspan="4">---</td></tr>
<tr><td>DA (hex)
J3200</td><td>---</td><td>…</td><td>---</td><td>---</td><td>…</td><td>11</td><td colspan="4">---</td></tr>
<tr><td>ECU name</td><td>---</td><td>…</td><td>---</td><td>---</td><td>…</td><td>SAS</td><td colspan="4">---</td></tr>
<tr><td>Hex value</td><td colspan="3">00</td><td colspan="3">01</td><td colspan="4">---</td></tr>
</table>

Appendix D: Diagnostic Addresses

This Appendix contains Table D1 which shows the acceptable ranges of diagnostic addresses which can be applied to a given type of ECU. The diagnostic address provides a means for a test tool to correlate diagnostic CANIds to physical ECUs. The diagnostic address is also used to determine the diagnostic CANIds used for programming certain SPS Programmable ECUs. All ECUs on a given vehicle shall have a unique diagnostic address. The $1A ReadDataByIdentifier service is used to read the diagnostic address information out of an ECU.

SAE J2178/1 was used as the basis for the ranges assigned to each controller type.

Table D1: Diagnostic Address Assignment Ranges

Controller Type/Category		**Address Range (hex)**
Powertrain Controllers		00 to 1F
	Integration/Manufacturer Expansion	00 to 0F
	Engine Controllers	10 to 17
	Transmission Controllers	18 to 1F
Chassis Controllers		20 to 3F
	Integration/Manufacturer Expansion	20 to 27
	Brake Controllers	28 to 2F
	Steering Controllers	30 to 37
	Suspension Controllers	38 to 3F
Body Controllers		40 to C7
	Integration/Manufacturer Expansion	40 to 57
	Restraints	58 to 5F
	Driver Information/ Displays	60 to 6F
	Lighting	70 to 7F
	Entertainment/ Audio	80 to 8F
	Personal Communication	90 to 97
	Climate Control(HVAC)	98 to 9F
	Convenience (Doors, Seats, Windows, etc.)	A0 to BF
	Security	C0 to C7
Electric Vehicle Energy Transfer System		C8 to CB
	Utility Connection Services	C8
	AC to AC Conversion	C9
	AC to DC Conversion	CA
	Energy Storage Management	CB
Future Expansion		CC to FD
Reserved By Document		FE to FF

Appendix E: DTC Status Byte and Failure Type Byte Definitions

This Appendix contains the bit definition of the DTC status byte, and also the definition of the DTC Failure Type byte. A tester can retrieve this information (along with the base DTC number) from an ECU by requesting the $A9 service.

E1 DTC Status Byte Bit Definitions

Every node shall adhere to the convention for storing bit-packed DTC status information as defined in Table E1. Actual usage of the bit-fields may vary from node to node. The table breaks down the usage convention by the type of node. The categories are Emission related nodes, Safety related nodes, and other types of nodes which do not fall into either of the previous categories.

Table E1: DTC Status Bit Assignments

Bit	Description	Cvt (By Node Type)			Mnemonic
		Emiss	Safety	Other	
7	**warningIndicatorRequestedState** This bit shall report the status of any warning indicators associated with a particular DTC. Warning outputs may consist of indicator lamp(s), displayed text information, etc. If no warning indicators exist for a given system or particular DTC, this bit shall be set to a value of 0. If the warning output is on for a given DTC, the history status flag shall also be true. 1. Bit set to 1 = Warning indicator requested to be ON. 2. Bit set to 0 = Warning indicator not requested to be ON. 3. Bit will be set to a value of 0 on a successful code clear.	M	M	M_1 **Note 1**	WIRS
6	**currentDTCSincePowerUp** This bit shall indicate that a DTC became current during the current ignition/power cycle. This bit shall be reset to a value of 0 with each ignition cycle. 1. Bit set to 1 = DTC became current this power up. 2. Bit set to 0 = DTC did not become current this power up. 3. Bit will be set to a value of 0 on a successful code clear.	M	U	U	CDTCSPU
5	**testNotPassedSinceCurrentPowerUp** This bit shall indicate a value of 0 once the test indicates a passed result. A bit value of 1 indicates that the test has not run, or that the test has run and failed. This bit shall be reset to a value of 1 with each ignition cycle. 1. Bit set to 1 = Test not passed since current power up. 2. Bit set to 0 = Test passed since current power up. 3. Bit will be set to a value of 1 on a successful code clear.	M	M_1 **Note 1**	M_1 **Note 1**	TNPSCPU

Bit	Description	Cvt (By Node Type)			Mnemonic
4	**historyDTC** A history DTC indicates that a current DTC status has met sufficient criteria for storing a code into long term memory. Sufficient criteria may consist of a current status detected over a span of time or over multiple test cycles. If DTC criteria to activate a warning indicator are satisfied, then the DTC history bit shall be set to a value of 1 and the DTC status must be stored to long term memory. 1. Bit set to 1 = DTC is history. 2. Bit set to 0 = DTC is not history. 3. Bit will be set to a value of 0 on a successful code clear.	M	M	M	HDTC
3	**testFailedSinceDTCCleared** This bit shall indicate that a current DTC is or has been set sometime since the last time the DTC status was reset. 1. Bit set to 1 = Test failed since DTCs have been cleared. 2. Bit set to 0 = Test not failed since DTCs have been cleared. 3. Bit will be set to a value of 0 on a successful code clear.	M	U	U	TFSDTCC
2	**testNotPassedSinceDTCCleared** This bit shall indicate a value of 0 once the test indicates a passed result. A value of 1 indicates that the test has not run, or that the test has run and failed. 1. Bit set to 1 = Test not passed since DTC cleared. 2. Bit set to 0 = Test passed since DTC cleared. 3. Bit will be set to a value of 1 on a successful code clear.	M	M_1 **Note 1**	M_1 **Note 1**	TNPSDTCC
1	**currentDTC** A current DTC shall indicate that test conditions have been met and the test results show that a fault is currently present. This bit shall indicate the results of the last diagnostic test performed. A current code status may span multiple power cycles since it reflects the result of the last test performed. 1. Bit set to 1 = DTC is current. 2. Bit set to 0 = DTC is not current. 3. Bit will be set to a value of 0 on a successful code clear.	M	M	M	CDTC
0	**DTCSupportedByCalibration** This bit shall indicate that a node will trigger the test associated with a particular DTC as soon as the criteria for performing the test have been satisfied. 1. Bit set to 1 = DTC supported. 2. Bit set to 0 = DTC not supported. 3. Bit is NOT altered by a code clear attempt.	U	U	U	DTCSBC

Note 1: M_1 – The intent of these bits is to provide build integrity and repair verification. Deviation from supporting these bits (e.g., Hardware limitations) shall be negotiated by service, assembly, and the DRE, and documented in the CTS.

Note: If either bit 6 or bit 5 is supported, then the other should also be supported. Also, if either bit 3 or bit 2 is supported, then the other should also be supported. In other words, bits 6 and 5 should be supported as a pair, and bits 3 and 2 should be supported as a pair.

E1.1 DTC Status Byte Bit Operation. The following figures illustrate the expected values of the status byte for various types of DTCs under various operating conditions. In general, DTCs may be divided into two major categories, Powertrain and Non-Powertrain DTCs.

For Non-Powertrain DTCs, the DTCs may be categorized based upon how the DTC is latched. Latching a DTC refers to retaining the present state of the current bit in the status byte. This retention may be permanent or of limited duration until an event occurs (e.g., ECU power up reset).

Latching a DTC alters the operation of the ECU with respect to the execution of and/or reaction to the results of a supported diagnostic test. A diagnostic test consists of actions performed from a given moment when all conditions to start a diagnostic fault detection algorithm are fulfilled until the moment a diagnostic fault detection algorithm makes a pass/fail decision. A failed diagnostic test indicates that the detected failure is severe enough to set a DTC with current status.

Certain diagnostic tests are only run once per ignition cycle (e.g., EEPROM checksum test where the checksum is calculated at power down and recalculated at power up with a mismatch indicating a fault). Once the Current status is set to true, it remains set to true for the duration of the ignition cycle because the diagnostic test does not run until the next ignition cycle. This is not considered a latching DTC.

All Non-Powertrain DTCs may be classified as belonging to one of three types:

- **Non-Latching DTCs (Figures E1 and E2):** (Also called Condition Latching DTCs) Once the Current status has been set to “True”, it remains set to true until the diagnostic test makes a pass decision. Most currently implemented DTCs are of the non-latching type.
- **Limited Duration Latching DTCs (Figures E3 and E4):** Once the Current status has been set to true, it remains set to “True” regardless of the results of the diagnostic test until a specified event (e.g., power up reset) occurs.

  Note: The DTC is cleared when a clear diagnostic information command is received. Since the DTC is latched, the diagnostic algorithm does not run. Therefore, the repair cannot be verified unless the specified event occurs (e.g., ignition is cycled).
- **Permanently Latched DTCs (Figures E5 and E6):** Once the Current status has been set to true, it remains set to true regardless of the results of the diagnostic test. This type of DTC cannot be permanently cleared and requires ECU replacement. The DTC can be temporarily cleared using a scan tool, but the DTC returns at the next power up of the ECU.

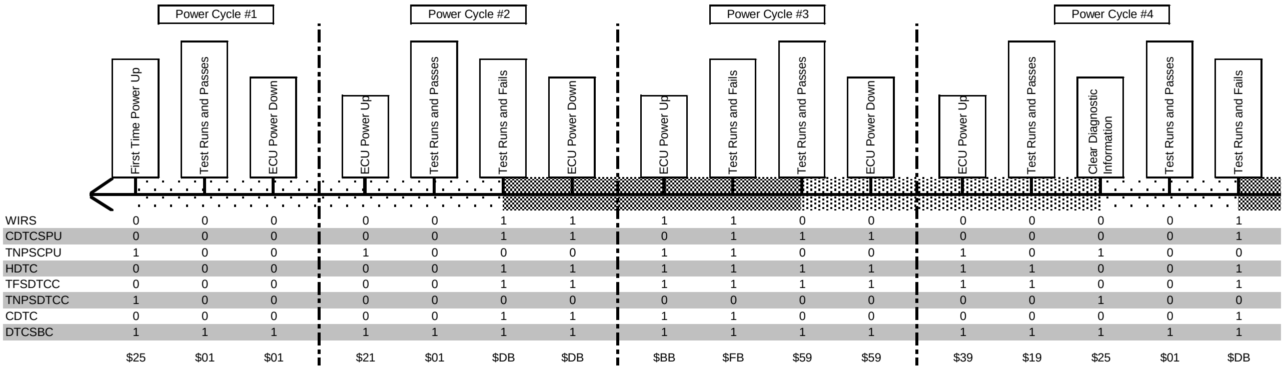
Powertrain DTCs are also categorized into three main categories; Type A, Type B, and Type C (there is a fourth category for DTC not supported that is not illustrated here, as these DTCs are in the ECU software but are not run) based upon illumination of the Malfunction Indicator Lamp (MIL).

Note: Due to varying regional requirements, the following tables cannot illustrate all possible cases. These tables are provided as examples and, in case of a conflict, do not supersede any documented diagnostic requirements.

- **Type A DTCs (Figures E7, E8, and E9):** The ECU sets the status current, history, and requests illumination of the malfunction indicator lamp (MIL) when the diagnostic runs and fails. The current status is cleared when a Test Pass is reported but the MIL remains illuminated for 3 consecutive ignition cycles, depending on the DTC, when the diagnostic runs and does not fail.
- **Type B DTCs (Figures E10, E11, and E12):** The ECU illuminates the malfunction indicator lamp (MIL) when the diagnostic runs and fails during 2 or 3 consecutive trips, depending on the regional requirements. The first failure will cause the current status bit to be set, but the history and warning indicator requested status bits do not set unless the diagnostic runs and fails during 2 or 3 consecutive trips.
- **Type C DTCs (Figures E13 and E14):** The ECU stores the DTC information into memory when the diagnostic runs and fails. Type C DTCs do not illuminate the malfunction indicator lamp (MIL) but may cause another indicator to illuminate.

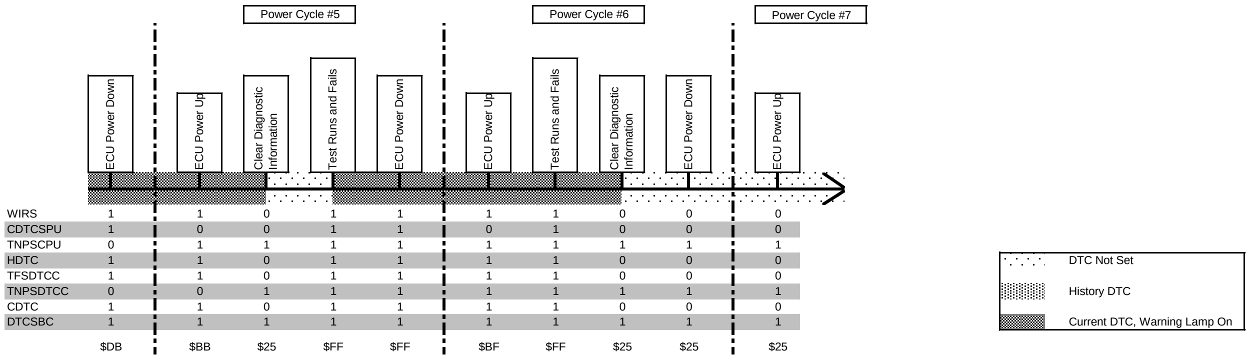
DTC information includes the base DTC (two bytes), failure type byte, status byte, freeze frame data (emission related ECUs only), failure record information, and other data such as flags, counters, timers, etc., specific to the DTC. This information has limited duration usefulness. An automatic clearing of DTC information function is implemented in order to avoid reporting out of date information. For Non-Powertrain ECUs, all DTCs are cleared at the same time. This means that the timing of the automatic clearing of DTCs is based upon the most recently detected DTC.

- **Automatic Clearing of DTCs (Figure E15):** Automatic clearing of DTC Information is clearing all DTCs in the ECU memory after a predetermined number of consecutive ignition on/off cycles with no test fail result. The typical number of fault free ignition cycles is 40.



	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails
WIRS	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
CDTC	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$DB	$DB	$BB	$FB	$59	$59	$39	$19	$25	$01	$DB

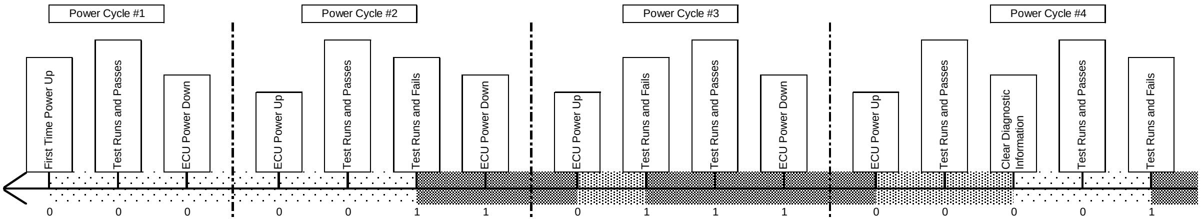
Figure E1: Non-Latching DTC (1 of 2)



	ECU Power Down	ECU Power Up	Clear Diagnostic Information	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Clear Diagnostic Information	ECU Power Down	ECU Power Up
WIRS	1	1	0	1	1	1	1	0	0	0
CDTCSPU	1	0	0	1	1	0	1	0	0	0
TNPSCPU	0	1	1	1	1	1	1	1	1	1
HDTC	1	1	0	1	1	1	1	0	0	0
TFSDTCC	1	1	0	1	1	1	1	0	0	0
TNPSDTCC	0	0	1	1	1	1	1	1	1	1
CDTC	1	1	0	1	1	1	1	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1	1
	$DB	$BB	$25	$FF	$FF	$BF	$FF	$25	$25	$25

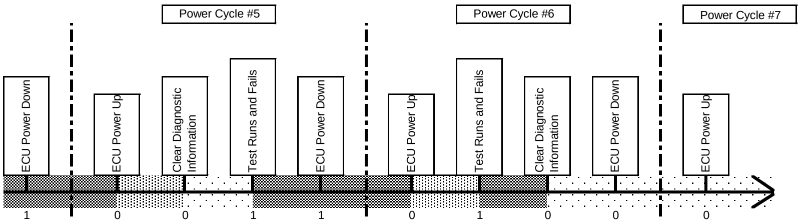
⋰	DTC Not Set
░	History DTC
▩	Current DTC, Warning Lamp On

Figure E2: Non-Latching DTC (2 of 2)



	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails
WIRS	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
CDTC	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$DB	$DB	$39	$FB	$DB	$DB	$39	$19	$25	$01	$DB

Figure E3: Limited Duration Latching DTC (1 of 2)



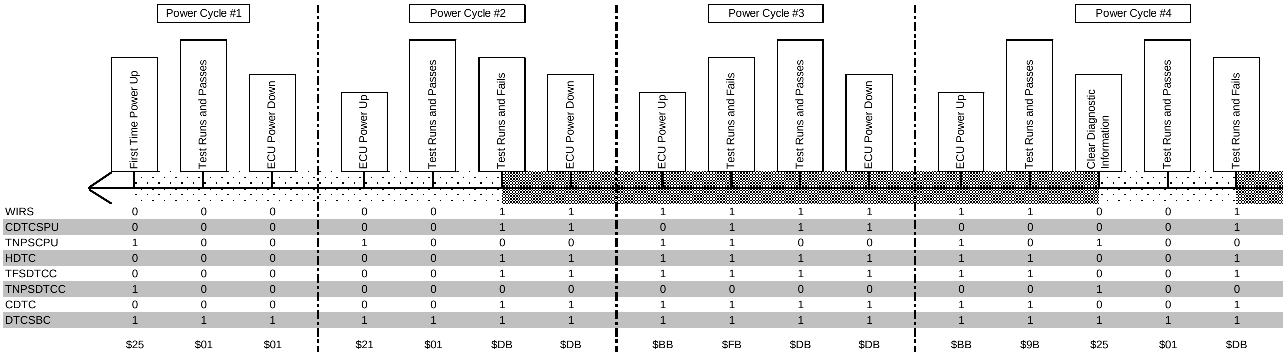
	ECU Power Down	ECU Power Up	Clear Diagnostic Information	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Clear Diagnostic Information	ECU Power Down	ECU Power Up
WIRS	1	0	0	1	1	0	1	0	0	0
CDTCSPU	1	0	0	1	1	0	1	0	0	0
TNPSCPU	0	1	1	1	1	1	1	1	1	1
HDTC	1	1	0	1	1	1	1	0	0	0
TFSDTCC	1	1	0	1	1	1	1	0	0	0
TNPSDTCC	0	0	1	1	1	1	1	1	1	1
CDTC	1	0	0	1	1	0	1	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1	1
	$DB	$39	$25	$FF	$FF	$3D	$FF	$25	$25	$25

⁚ DTC Not Set

▒ History DTC

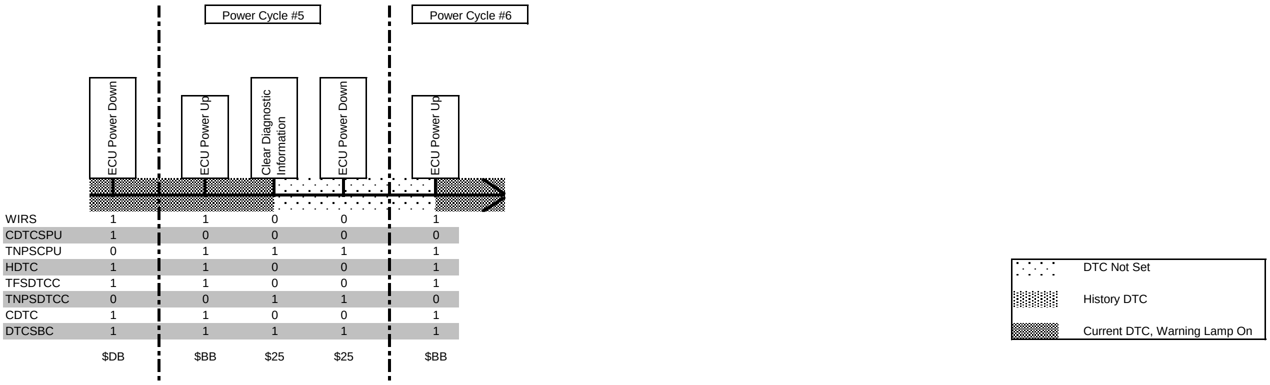
▓ Current DTC, Warning Lamp On

Figure E4: Limited Duration Latching DTC (2 of 2)



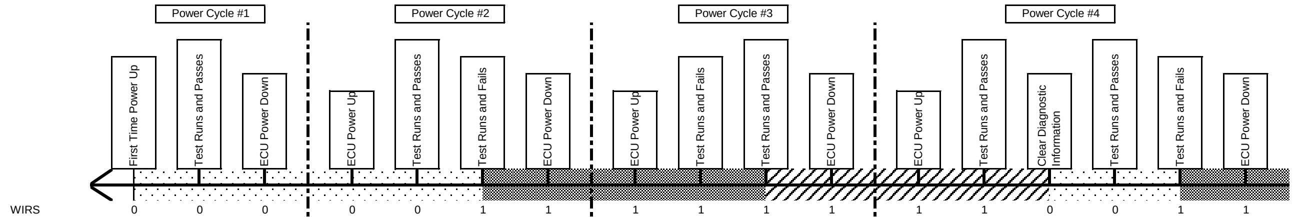
	Power Cycle #1			Power Cycle #2				Power Cycle #3				Power Cycle #4				
	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails
WIRS	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
CDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$DB	$DB	$BB	$FB	$DB	$DB	$BB	$9B	$25	$01	$DB

Figure E5: Permanently Latched DTC (1 of 2)



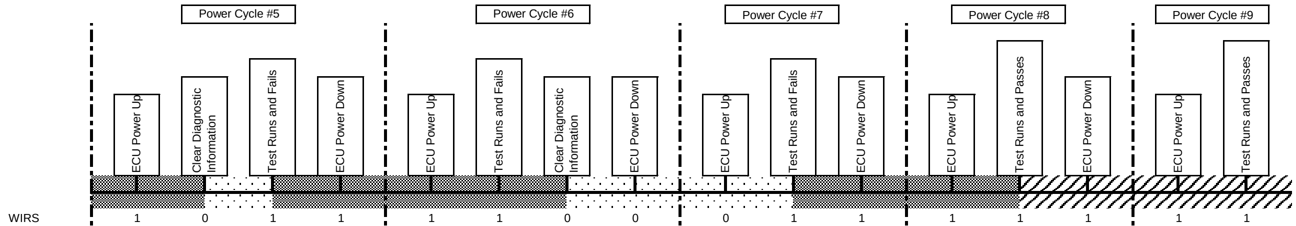
		Power Cycle #5			Power Cycle #6
	ECU Power Down	ECU Power Up	Clear Diagnostic Information	ECU Power Down	ECU Power Up
WIRS	1	1	0	0	1
CDTCSPU	1	0	0	0	0
TNPSCPU	0	1	1	1	1
HDTC	1	1	0	0	1
TFSDTCC	1	1	0	0	1
TNPSDTCC	0	0	1	1	0
CDTC	1	1	0	0	1
DTCSBC	1	1	1	1	1
	$DB	$BB	$25	$25	$BB

Figure E6: Permanently Latched DTC (2 of 2)



	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails	ECU Power Down
WIRS	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
CDTC	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$DB	$DB	$BB	$FB	$D9	$D9	$B9	$99	$25	$01	$DB	$DB

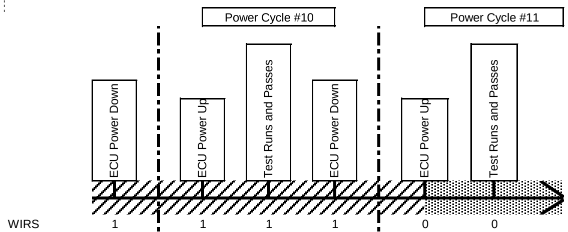
Figure E7: Powertrain Type A DTC (1 of 3)



	ECU Power Up	Clear Diagnostic Information	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Clear Diagnostic Information	ECU Power Down	ECU Power Up	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes
WIRS	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	1
CDTCSPU	0	0	1	1	0	1	0	0	0	1	1	0	0	0	0	0
TNPSCPU	1	1	1	1	1	1	1	1	1	1	1	1	0	0	1	0
HDTC	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	1
TFSDTCC	1	0	1	1	1	1	0	0	0	1	1	1	1	1	1	1
TNPSDTCC	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
CDTC	1	0	1	1	1	1	0	0	0	1	1	1	0	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$BB	$25	$FF	$FF	$BF	$FF	$25	$25	$25	$FF	$FF	$BF	$99	$99	$B9	$99

Figure E8: Powertrain Type A DTC (2 of 3)

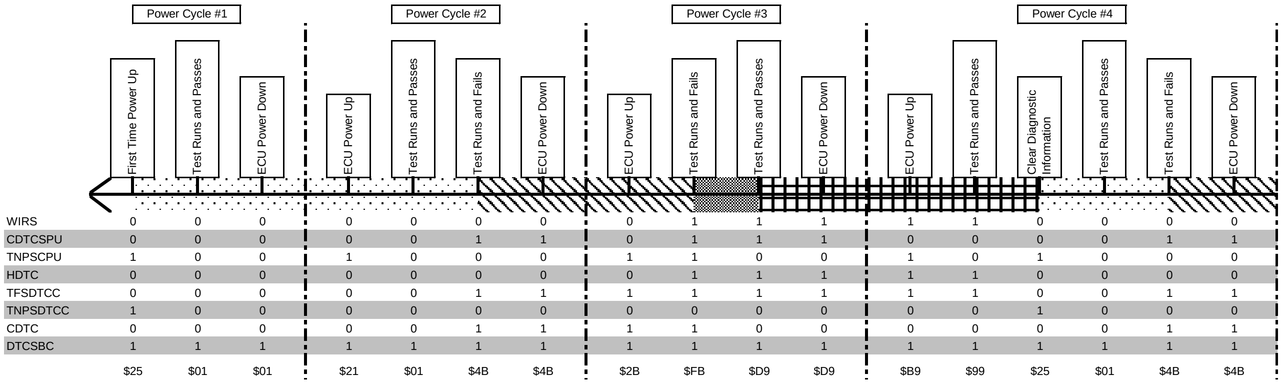
Status Byte Operation



	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes
WIRS	1	1	1	1	0	0
CDTCSPU	0	0	0	0	0	0
TNPSCPU	0	1	0	0	1	0
HDTC	1	1	1	1	1	1
TFSDTCC	1	1	1	1	1	1
TNPSDTCC	0	0	0	0	0	0
CDTC	0	0	0	0	0	0
DTCSBC	1	1	1	1	1	1
	$99	$B9	$99	$99	$39	$19

⁚	DTC Not Set
░	History DTC
▨	History DTC, Warning Lamp On
▓	Current DTC, Warning Lamp On

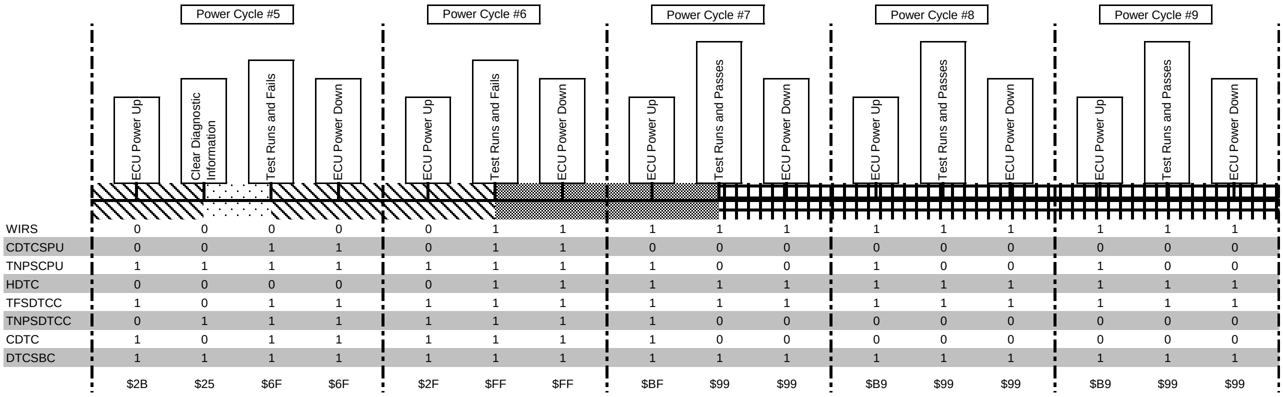
Figure E9: Powertrain Type A DTC (3 of 3)



	Power Cycle #1			Power Cycle #2				Power Cycle #3				Power Cycle #4					
	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails	ECU Power Down
WIRS	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0	0
HDTC	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
CDTC	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$4B	$4B	$2B	$FB	$D9	$D9	$B9	$99	$25	$01	$4B	$4B

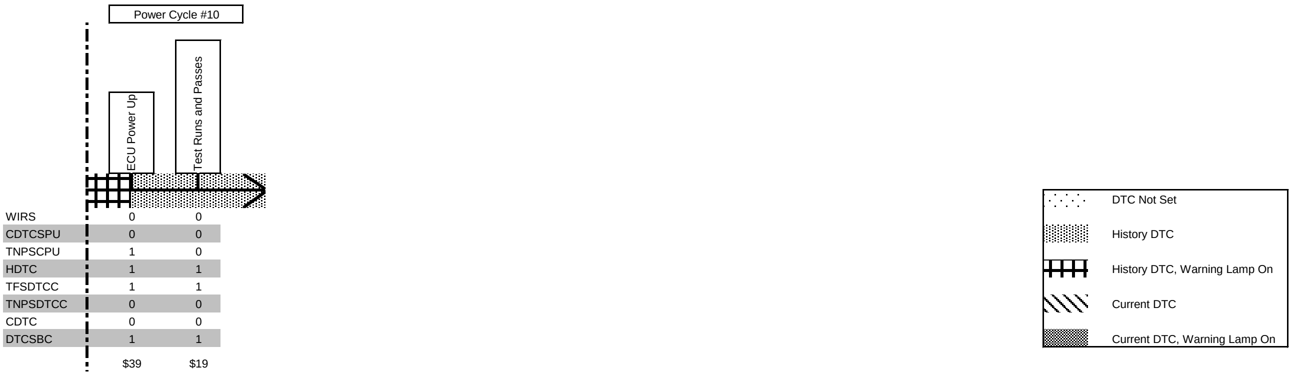
Figure E10: Powertrain Type B DTC (1 of 3)

Status Byte Operation



	Power Cycle #5				Power Cycle #6			Power Cycle #7			Power Cycle #8			Power Cycle #9		
	ECU Power Up	Clear Diagnostic Information	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down
WIRS	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
CDTCSPU	0	0	1	1	0	1	1	0	0	0	0	0	0	0	0	0
TNPSCPU	1	1	1	1	1	1	1	1	0	0	1	0	0	1	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
TFSDTCC	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
TNPSDTCC	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0
CDTC	1	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$2B	$25	$6F	$6F	$2F	$FF	$FF	$BF	$99	$99	$B9	$99	$99	$B9	$99	$99

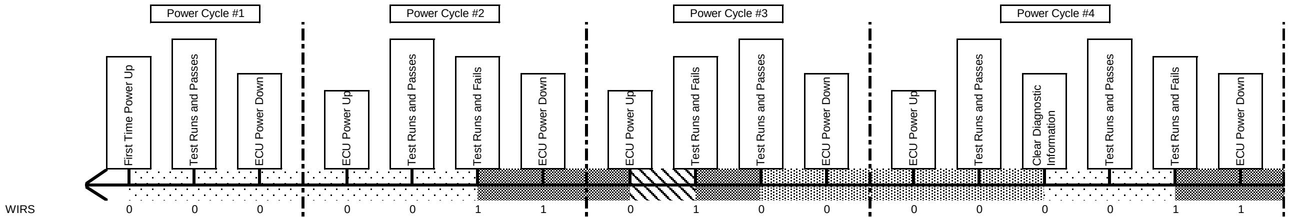
Figure E11: Powertrain Type B DTC (2 of 3)



	Power Cycle #10	
	ECU Power Up	Test Runs and Passes
WIRS	0	0
CDTCSPU	0	0
TNPSCPU	1	0
HDTC	1	1
TFSDTCC	1	1
TNPSDTCC	0	0
CDTC	0	0
DTCSBC	1	1
	$39	$19

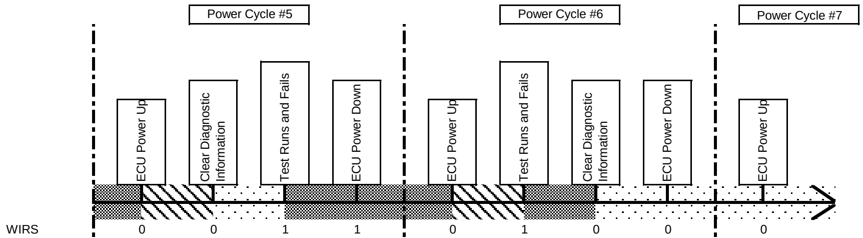
Figure E12: Powertrain Type B DTC (3 of 3)

Status Byte Operation



	Power Cycle #1			Power Cycle #2				Power Cycle #3				Power Cycle #4					
	First Time Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	Clear Diagnostic Information	Test Runs and Passes	Test Runs and Fails	ECU Power Down
WIRS	0	0	0	0	0	1	1	0	1	0	0	0	0	0	0	1	1
CDTCSPU	0	0	0	0	0	1	1	0	1	1	1	0	0	0	0	1	1
TNPSCPU	1	0	0	1	0	0	0	1	1	0	0	1	0	1	0	0	0
HDTC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
TFSDTCC	0	0	0	0	0	1	1	1	1	1	1	1	1	0	0	1	1
TNPSDTCC	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
CDTC	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1
DTCSBC	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	$25	$01	$01	$21	$01	$DB	$DB	$3B	$FB	$59	$59	$39	$19	$25	$01	$DB	$DB

Figure E13: Powertrain Type C DTC (1 of 2)

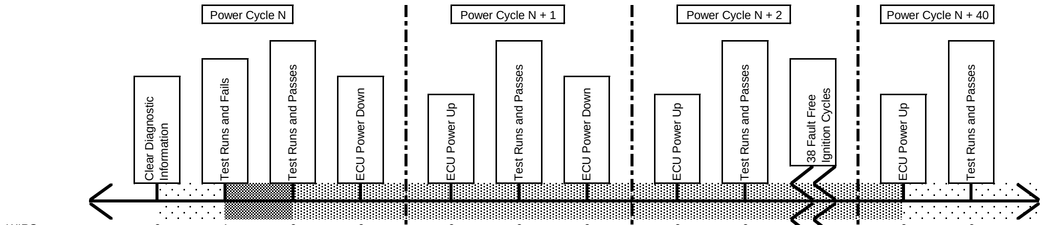


	Power Cycle #5				Power Cycle #6				Power Cycle #7
	ECU Power Up	Clear Diagnostic Information	Test Runs and Fails	ECU Power Down	ECU Power Up	Test Runs and Fails	Clear Diagnostic Information	ECU Power Down	ECU Power Up
WIRS	0	0	1	1	0	1	0	0	0
CDTCSPU	0	0	1	1	0	1	0	0	0
TNPSCPU	1	1	1	1	1	1	1	1	1
HDTC	1	0	1	1	1	1	0	0	0
TFSDTCC	1	0	1	1	1	1	0	0	0
TNPSDTCC	0	1	1	1	1	1	1	1	1
CDTC	1	0	1	1	1	1	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1
	$3B	$25	$FF	$FF	$3F	$FF	$25	$25	$25

⁘	DTC Not Set
▒	History DTC
▧	Current DTC
▓	Current DTC, Warning Lamp On

Figure E14: Powertrain Type C DTC (2 of 2)

Status Byte Operation



	Power Cycle N				Power Cycle N + 1			Power Cycle N + 2		Power Cycle N + 40	
	Clear Diagnostic Information	Test Runs and Fails	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Down	ECU Power Up	Test Runs and Passes	ECU Power Up	Test Runs and Passes
WIRS	0	1	0	0	0	0	0	0	0	0	0
CDTCSPU	0	1	1	1	0	0	0	0	0	0	0
TNPSCPU	1	1	0	0	1	0	0	1	0	1	0
HDTC	0	1	1	1	1	1	1	1	1	0	0
TFSDTCC	0	1	1	1	1	1	1	1	1	0	0
TNPSDTCC	1	1	0	0	0	0	0	0	0	1	0
CDTC	0	1	0	0	0	0	0	0	0	0	0
DTCSBC	1	1	1	1	1	1	1	1	1	1	1
	$25	$FF	$59	$59	$39	$19	$19	$39	$19	$25	$01

⁘	DTC Not Set
▒	History DTC
▓	Current DTC, Warning Lamp On

Figure E15: Automatic Clearing of DTCs

E1.2 DTC Failure Type Definition General Description. The DTC Failure Type consists of 16 different failure categories, where each category is associated with 16 Sub Type Failures (also known as symptoms). The Sub Type Failures are logically grouped in a DTC Failure Type Category. This was done to simplify the selection of the appropriate Sub Type Failure (Symptom) for a DTC.

The DTC Failure Category is decoded in the High Nibble of the **DTC Failure Type Byte** and the Failure Sub Type is decoded in the Low Nibble of the **DTC Failure Type Byte.** Refer to Table E2.

Table E3 provides the DTC Failure Type Byte category definitions.

Table E2: DTC Failure Type Byte Definition

DTC Failure Type Byte Definition	
High Nibble ($0x thru $Fx)	**Low Nibble ($x0 thru $xF)**
DTC Failure Category	DTC Failure Sub Type

Table E3: DTC Failure Category Definition

DTC Failure Type Byte Category Definitions		
High Nibble (0000b thru 1111b)	**Category #**	**Category Description**
0000	0	General Electrical Failures
0001	1	Additional General Electrical Failures
0010	2	FM/PWM (Frequency/Pulse Width Modulated) Failures
0011	3	ECU Internal Failures
0100	4	ECU Programming Failures
0101	5	Algorithm Based Failures
0110	6	Mechanical Failures
0111	7	Bus Signal/Message Failures
1000 thru 1110	8 thru E	Reserved By Document
1111	F	Unique System Specific (assignments made by service operations and coordinated with manufacturing dept.)

E1.2.1 Definition and Appropriate Uses Of DTC Failure Type Value $00. The DTC Failure Type value zero ($00) is used to indicate that no additional information is available beyond that provided by the description for the DTC two (2) byte base code. Although the value of zero ($00) falls into the General Electrical Failures category, it can also be used for other types of DTCs as long as no other fault type values exist that would provide additional information.

In cases where legislative requirements do not allow the reporting of DTCs with a fault type byte (e.g., emissions related DTCs), the ECU may choose to report its DTCs via enhanced diagnostic services with the fault type byte coded to zero ($00). This approach allows the same DTC tables to be used for reporting DTCs with both legislated and enhanced diagnostic services.

E1.3 DTC Failure Type Definition by Category. The DTC Failure Type Byte Definitions in the tables below shall be used to provide additional fault information for all DTCs which do not meet one of the exceptions outlined in section E1.2.1.

E1.3.1 DTC Failure Sub Type (Symptom) Definition of General Electrical Failures.

Table E4: DTC Failure Sub Type Definition for Failure Category "0"

Failure Type	**Sub Type**	**General Electrical Failures** This category includes standard wiring failure modes (i.e., shorts and opens), direct current (DC) quantities related by Ohm's Law and quantities related to amplitude, frequency or rate of change, and waveshape.	
Byte	Nibble	**Sub Type**	
(hex)	(binary)	**Description**	**Definition**

Failure Type	**Sub Type**	**General Electrical Failures** This category includes standard wiring failure modes (i.e., shorts and opens), direct current (DC) quantities related by Ohm's Law and quantities related to amplitude, frequency or rate of change, and waveshape.	
Byte	Nibble	**Sub Type**	
(hex)	(binary)	**Description**	**Definition**
00	0000	no additional information (see E1.2.1)	This sub type is used for failures that cannot be assigned to a specific sub type. See E1.2.1.
01	0001	short to battery	This sub type is used for failures where the ECU measures vehicle system (battery positive) potential for greater than a specified time period or when some other value is expected.
02	0010	short to ground	This sub type is used for failures where the ECU measures ground (battery negative) potential for greater than a specified time period or when some other value is expected.
03	0011	voltage below threshold	This sub type is used for failures where the ECU measures a voltage below a specified range but not necessarily a short to ground.
04	0100	open circuit	This sub type is used for failures where the ECU determines an open circuit via lack of bias voltage, low current flow, no change in state of an input in response to an output, etc.
05	0101	short to battery or open	This sub type is used for failures where the condition detected by the ECU is the same for either indicated failure mode.
06	0110	short to ground or open	This sub type is used for failures where the condition detected by the ECU is the same for either indicated failure mode.
07	0111	voltage above threshold	This sub type is used for failures where the ECU measures a voltage above a specified range but not necessarily a short to battery.
08	1000	signal invalid	This sub type is used for failures where the value of the signal is not plausible given the operating conditions.
09	1001	rate of change above threshold	This sub type is used for failures where the signal transitions more quickly than is reasonably allowed.
0A	1010	rate of change below threshold	This sub type is used for failures where the signal transitions more slowly than is reasonably allowed.
0B	1011	current above threshold	This sub type is used for failures where the ECU measures current flow above a specified range.
0C	1100	current below threshold	This sub type is used for failures where the ECU measures current flow below a specified range.
0D	1101	resistance above threshold	This sub type is used for failures where the ECU infers a circuit resistance above a specified range.
0E	1110	resistance below threshold	This sub type is used for failures where the ECU infers a circuit resistance below a specified range.
0F	1111	erratic	This sub type is used for failures where the signal is momentarily implausible (not long enough for signal invalid) or discontinuous.

E1.3.2 DTC Failure Sub Type (Symptom) Definition of Additional General Electrical Failures.

Table E5: DTC Failure Sub Type Definition for Failure Category "1"

Failure Type	**Sub Type**	**Additional General Electrical Failures** **Note:** This category includes the overflow from category 0.	
Byte (hex)	Nibble (binary)	**Sub Type Description**	**Definition**
10	0000	reserved	
11	0001	above maximum threshold	This sub type is used for failures where some circuit quantity is above a specified range.
12	0010	below minimum threshold	This sub type is used for failures where some circuit quantity is below a specified range.
13	0011	voltage low/high temperature	This sub type is used for failures where a temperature sensor with a negative temperature coefficient detects a voltage below a specified range.
14	0100	voltage high/low temperature	This sub type is used for failures where a temperature sensor with a negative temperature coefficient detects a voltage above a specified range.
15	0101	signal rising time failure	This sub type is used for failures where the signal rise time is outside a specified range.
16	0110	signal falling time failure	This sub type is used for failures where the signal fall time is outside a specified range.
17	0111	signal shape/ waveform failure	This sub type is used for failures where the shape of the signal (plot of the amplitude with respect to time) is not correct, e.g., improper circuit impedance.
18	1000	signal amplitude < minimum	This sub type is used for failures where the ECU measures a signal voltage below a specified range but not necessarily a short to ground, e.g., gain too low.
19	1001	signal amplitude > maximum	This sub type is used for failures where the ECU measures a signal voltage above a specified range but not necessarily a short to battery, e.g., gain too high.
1A	1010	bias level out of range	This sub type is used for failures where the ECU applies a bias voltage to a circuit upon which is superimposed a signal voltage, e.g., Oxygen Sensor circuit.
1B	1011	signal cross coupled	This sub type is used for failures where the ECU detects a circuit shorted to another circuit when both circuits are controlled by the ECU.
1C	1100	reserved	
1D	1101	reserved	
1E	1110	reserved	
1F	1111	intermittent	This sub type is used for failures where the ECU momentarily detects one of the conditions defined above but not long enough to set a specific sub type.

E1.3.3 DTC Failure Sub Type (Symptom) Definition Of FM/PWM Failures.

Table E6: DTC Failure Sub Type Definition for Failure Category "2"

Failure Type	**Sub Type**	**FM/PWM (Frequency/Pulse Width Modulated) Failures** **Note:** This category includes faults related to Frequency Modulated (FM) and Pulse Width Modulated (PWM) inputs and outputs of the ECU. This category also includes faults where position is determined by counts.	
Byte	Nibble	**Sub Type**	
(hex)	(binary)	**Description**	**Definition**
20	0000	reserved	
21	0001	incorrect period	This sub type is used for failures where the ECU measures an incorrect duration for one cycle of the output.
22	0010	low time < minimum	This sub type is used for failures where the ECU detects the low pulse is too narrow with respect to time.
23	0011	low time > maximum	This sub type is used for failures where the ECU detects the low pulse is too wide with respect to time.
24	0100	high time < minimum	This sub type is used for failures where the ECU detects the high pulse is too narrow with respect to time.
25	0101	high time > maximum	This sub type is used for failures where the ECU detects the high pulse is too wide with respect to time.
26	0110	frequency too low	This sub type is used for failures where the ECU detects too few cycles in a given time period.
27	0111	frequency too high	This sub type is used for failures where the ECU detects too many cycles in a given time period.
28	1000	incorrect frequency	This sub type is used for failures where the ECU measures an incorrect number of cycles in a given time period.
29	1001	too few pulses	This sub type is used for failures where the ECU measures too few pulses (e.g., position is calibrated in counts from one extreme to the other).
2A	1010	too many pulses	This sub type is used for failures where the ECU measures too many pulses (e.g., position is calibrated in counts from one extreme to the other).
2B	1011	missing reference	This sub type is used for failures where the ECU does not detect a reference for a signal circuit or a group of signal circuits.
2C	1100	reserved	Defined in previous revision of this specification as **reference compare error**. Eliminated due to redundancy with other existing fault types.
2D	1101	reserved	
2E	1110	reserved	
2F	1111	reserved	

E1.3.4 DTC Failure Sub Type (Symptom) Definition of ECU Internal Failures.

Table E7: DTC Failure Sub Type Definition for Failure Category “3”

Failure Type	**Sub Type**	**ECU Internal Failures** **Note:** This category includes faults related to memory, software, and internal electrical circuitry; requiring component replacement.	
Byte (hex)	Nibble (binary)	**Sub Type** **Description**	**Definition**
30	0000	reserved	
31	0001	general checksum failure	This sub type is used by the ECU to indicate an incorrect checksum calculation where memory type is not specified.
32	0010	general memory failure	This sub type is used by the ECU to indicate a memory failure where memory type is not specified.
33	0011	special memory failure	This sub type is used by the ECU to indicate a memory failure where the specific memory type is not defined in this category.
34	0100	RAM failure	This sub type is used by the ECU to indicate a Random Access Memory (RAM) failure.
35	0101	ROM failure	This sub type is used by the ECU to indicate a Read Only Memory (ROM) failure.
36	0110	EEPROM failure	This sub type is used by the ECU to indicate an Electrically Erasable Programmable Read Only Memory (EEPROM) failure.
37	0111	watchdog/safety µC failure	This sub type is used by the ECU to indicate a failure in the execution of operational software.
38	1000	supervision software failure	This sub type is used by the ECU to indicate a loop time error in the execution of the operational software.
39	1001	internal electronic failure	This sub type is used by the ECU to indicate the detection of an internal circuit failure.
3A	1010	incorrect component installed	This sub type is used by the ECU to indicate a mismatch between the hardware connected to the ECU and the hardware expected by the ECU.
3B	1011	Internal Self Test Failed	This sub type is used by the ECU to indicate a sensor self-test failure launched by an ECU command.
3C	1100	Internal Communications Failure	This sub type is used by the ECU to indicate loss of an internal communication line, e.g., between microprocessors in a dual microprocessor configuration.
3D	1101	reserved	
3E	1110	reserved	
3F	1111	reserved	

E1.3.5 DTC Failure Sub Type (Symptom) Definition of ECU Programming Failures.

Table E8: DTC Failure Sub Type Definition for Failure Category "4"

Failure Type	**Sub Type**	**ECU Programming Failures** **Note:** This category includes faults related to operational software, calibrations, and options; remedied by programming the ECU.	
Byte (hex)	Nibble (binary)	**Sub Type** **Description**	**Definition**
40	0000	reserved	
41	0001	operational software/calibration set not programmed	This sub type is used to indicate that only boot software is present in the ECU.
42	0010	calibration data set not programmed	This sub type is used to indicate that operational software is present but calibration data is not.
43	0011	EEPROM error	This sub type is used to indicate an EEPROM error that may be remedied by reprogramming the module.
44	0100	security access not activated	This sub type is used to indicate that programming was attempted without unlocking the ECU.
45	0101	variant not programmed	This sub type is used to indicate the need to enter (program) the sub system option content.
46	0110	vehicle configuration not programmed	This sub type is used to indicate the need to enter (program) the vehicle option content.
47	0111	VIN not programmed	This sub type is used to indicate the need to enter (program) the vehicle identification number (VIN).
48	1000	theft/security data not programmed	This sub type is used to indicate the need to enter (program) the theft/security code.
49	1001	RAM error	This sub type is used by the ECU to indicate a Random Access Memory (RAM) error remedied by reprogramming.
4A	1010	checksum error	This sub type is used by the ECU to indicate an incorrect checksum calculation where memory type is not specified.
4B	1011	calibration not learned	This sub type is used by the ECU to indicate that a password, operational range, etc. for a sensor or actuator must be learned by the ECU.
4C	1100	DTC memory full	This sub type is used by the ECU to indicate that more DTCs have been detected than can be accommodated by the memory allocated for DTC storage.
4D	1101	stack overflow	This subtype is used by the ECU to indicate that more memory has been used in a stack than is allocated to a program.
4E	1110	reserved	
4F	1111	reserved	

E1.3.6 DTC Failure Sub Type (Symptom) Definition of Algorithm Based Failures.

Table E9: DTC Failure Sub Type Definition for Failure Category "5"

Failure Type	**Sub Type**	**Algorithm Based Failures** **Note:** This category includes faults based on comparing two or more input parameters for plausibility or comparing a single parameter to itself with respect to time.	
Byte (hex)	Nibble (binary)	**Sub Type** **Description**	**Definition**
50	0000	reserved	
51	0001	reserved	Defined in previous revision of this specification as **calculation failure**. Eliminated due to redundancy with other existing fault types.
52	0010	reserved	Defined in previous revision of this specification as **compare failure**. Eliminated due to redundancy with other existing fault types.
53	0011	temperature low	This sub type is used for failures where the ECU calculates a low temperature condition based upon the duration of certain operating parameters.
54	0100	temperature high	This sub type is used for failures where the ECU calculates a high temperature condition based upon the duration of certain operating parameters.
55	0101	expected number of transitions/ events not reached	This sub type is used for failures where the ECU monitors a parameter over time within specified limits and detects fewer than the expected number of transitions.
56	0110	allowable number of transitions/ events exceeded	This sub type is used for failures where the ECU monitors a parameter over time within specified limits and detects more than the expected number of transitions.
57	0111	reserved	Defined in previous revision of this specification as **expected reaction after event did not occur**. Eliminated due to redundancy with other existing fault types.
58	1000	incorrect reaction after event	This sub type is used for failures where the ECU does not see the expected change to a parameter or group of parameters in response to a particular event.
59	1001	circuit/component protection time-out	This sub type is used for failures where the ECU detects a function is active for greater than a specified time period.
5A	1010	plausibility failure	This sub type is used for failures where the ECU compares two or more input parameters for plausibility.
5B	1011	reserved	
5C	1100	reserved	
5D	1101	reserved	
5E	1110	reserved	
5F	1111	reserved	

E1.3.7 DTC Failure Sub Type (Symptom) Definition of Mechanical Failures.

Table E10: DTC Failure Sub Type Definition for Failure Category "6"

Failure Type	**Sub Type**	**Mechanical Failures** **Note:** This category includes faults detected by inappropriate motion in response to an ECU controlled output.	
Byte	Nibble	**Sub Type**	
(hex)	(binary)	**Description**	**Definition**
60	0000	reserved	
61	0001	actuator stuck	This sub type is used for failures where the ECU does not detect any motion in response to energizing a motor, solenoid, relay, etc.
62	0010	actuator stuck open	This sub type is used for failures where the ECU does not detect any motion upon commanding the operation of a motor, solenoid, relay, etc., to close some piece of equipment.
63	0011	actuator stuck closed	This sub type is used for failures where the ECU does not detect any motion upon commanding the operation of a motor, solenoid, relay, etc., to open some piece of equipment.
64	0100	actuator slipping	This sub type is used for failures where the ECU detects excessive duration to command a motor, solenoid, relay, etc., to move a piece of equipment to a desired position.
65	0101	emergency position not reachable	This sub type is used for failures where the ECU is unable to command a motor, solenoid, relay, etc., to move a piece of equipment to the emergency position.
66	0110	wrong mounting position	This sub type is used for failures where the server detects incorrectly mounted components, e.g., acceleration sensor showing a position error of 90°.
67	0111	incorrect assembly	This sub type is used for failures where the ECU detects that the component has been incorrectly installed (e.g., circuits cross-wired) or polarity errors.
68	1000	reserved	
69	1001	reserved	
6A	1010	reserved	
6B	1011	reserved	
6C	1100	reserved	
6D	1101	reserved	
6E	1110	reserved	
6F	1111	reserved	

E1.3.8 DTC Failure Sub Type (Symptom) Definition of Bus Signal Failures.

Table E11: DTC Failure Sub Type Definition for Failure Category "7"

Failure Type	**Sub Type**	**Bus Signal/Message Failures** **Note:** This category includes faults related to bus hardware and signal integrity. This category is also used when the physical input for a signal is located in one ECU and another ECU diagnoses the circuit.	
Byte	Nibble	**Sub Type**	
(hex)	(binary)	**Description**	**Definition**
70	0000	reserved	
71	0001	invalid serial data received	This sub type is used by the ECU to indicate a signal was received with the corresponding validity bit equal to **invalid** or post processing of the signal determines it is invalid.
72	0010	alive counter incorrect/not updated	This sub type is used by the ECU to indicate a signal was received without the corresponding rolling count value being properly updated.
73	0011	parity error	This sub type is used by the ECU to indicate a message was processed with an incorrect parity calculation.
74	0100	value of signal protection calculation incorrect	This sub type is used by the ECU to indicate a message was processed with an incorrect protection (checksum) calculation.
75	0101	signal above allowable range	This sub type is used for failures where some circuit quantity, reported via serial data, is above a specified range.
76	0110	signal below allowable range	This sub type is used for failures where some circuit quantity, reported via serial data, is below a specified range.
77	0111	reserved	
78	1000	reserved	
79	1001	reserved	
7A	1010	reserved	
7B	1011	reserved	
7C	1100	reserved	
7D	1101	reserved	
7E	1110	reserved	
7F	1111	Erratic	This sub type is used for failures where the signal, reported via serial data, is momentarily implausible or discontinuous.

E1.4 Requesting New DTC Numbers and/or Failure Types. The failure types listed beginning with paragraph E1.3 are current at the time of the publication of this specification. Assignment of DTC numbers and new DTC failure types (including any failure type assignments in the “F” failure type range) are managed by the GM Service and Parts Operations organization. The GM Service and Parts Operations web site contains the latest versions of the DTC documents and the site also contains a form for requesting new DTCs and/or DTC failure types.