DAKOTASOFT

FUNCTIONAL MANUAL FOR THE

J1939/ISO11783 PROTOCOL STACK
NMEA 2000 PROTOCOL STACK EXTENSION
ISO 15765 PROTOCOL STACK EXTENSION
ISO 14229 PROTOCOL STACK EXTENSION

VERSION: 1.47

March 25, 2021

1. Contents

2.

3. Revision History

Rev 1.00.01
Date: 12/22/06
Information:
* There is a warning that occurs with the Keil compiler where the IDENTITY_NUMBER const in the includes.h module must be cast into an U_8. This clears the warning.

Rev 1.01.00
Date: 06/05/07
Information:

* Added NMEA 2000 functionality to the stack. This functionality can be disabled by removing the #defines: FASTPACKET, REQUEST_COMMAND_ACK, TRANSMIT_PGNS.

Rev 1.01.01
Date: 11/23/07
Information:

* The function fnSetParameterGroupTx(U_32 u32PGN) was added to the TxData.c module. This function can be called to cause the immediate broadcast of a message that is in the list of messages that broadcast at a cyclic rate or upon request.

Rev 1.01.02
Date: 01/25/08
Information:

* Fixed a bug where the request parameter groups were no removed from the request message's data structure properly when a new received parameter group list was loaded for a given channel.

Rev 1.02.00
Date: 05/01/08
Information:

* The ability to handle multiple CAN ports was added to the stack. This feature impacted nearly every module in the stack.

* Fixed a bug in the includes.h file concerning the address claim NAME field. The NAME field was being built incorrectly by not placing the lower 3 bits of the manufacture code into the correct bit field. This bug affected the J1939 address claim only.

Rev 1.03.00
Date: 06/26/08
Information:

* Added documentation to a large number of modules for the new multi CAN port functionality added in the previous version of code.

* Changed the way the transmit DM2 message is handled. It is now handled in the same fashion as the DM1 message and is automatically broadcast after a request message is received for the DM2 PGN.

* Fixed a bug where the Transmit of a multipacket DM message would not always be sent to the global address.

* Increased the data size from 1 byte to 2 bytes for a loop counter in the function fnTransportProtoocIInit() so that data larger than 255 bytes can be received.

* Fixed a bug where the BAM Tx buffer's "Active" flag would not be reset after the message in the buffer was sent. This caused the stack's BAM Tx buffers to fill up and BAM transmissions to stop.

* Added a statement to the fnLoadTxMessageIntoList() function such that the destination address is concatenated to the end of the PGN if the PGN is destination specific. This allows the user to enter the same PGN into the list with multiple destination addresses.

* Fixed a bug in the fnProcessNMEATxMessage() function where the CAN port that the message is to be sent on was not copied over to the Tx structure. This would cause the message to not broadcast.

* Changed variable names BroadcastRate and Timeout to Interval and IntervalOffset. This was done to be consistent with NMEA2000 names.

* Added the function fnChangeNMEAIntervalOffset(). This function allows the complex request message to change the interval offset for a PGN.

* Fixed a bug where the fastpacket message would not wait 50ms between packet transmissions.

* Fixed a bug in the fnLoadNMEATxMesageIntoList() function where the CAN port value was not being copied from the passed data structure into the new data structure. The result is that the fastpacket messages would only be broadcast on CAN port 0.

* Fixed a bug where a complex request for a PGN that had its PGN access field set to ACCESS_DENIED would still cause a broadcast of the PGN.

* Fixed a bug where a complex request to change a PGN's broadcast rate was ignored if the complex request's "number of pairs" field was set to 0.

* Fixed a bug where the upper nibble of the byte that contains the change priority field in the complex command message was not being masked off. This caused the priority to not change for the commanded PGN.

* Added a call to the function fnChangeNMEAIntervalOffset() is the complex request message is received with a value other than 0xFFFF.

* Fixed a bug where fastpacket Tx buffer's instances field was not initialized to 0. This bug caused the stack to randomly enter an endless loop if a value greater than 65535 happened to be in the buffer at powerup.

* Fixed a bug where if a request message for an address claim message was received and the request was sent to the stacks destination address the stack would respond by sending the address claim message directly to the source address of the requesting device. It will now send the address claim message to the global address instead.

Rev 1.04.00
Date: 09/02/08
Information:

* Fixed a bug where the local variable u16Instance in function fnProcessNMEARequest() was not initialized before being used. This variable is now initialized to 0.

Rev 1.05.00
Date: 09/04/08
Information:

* Fixed a bug that caused a compiler error when the NMEA2000 address claim message was selected to be broadcast instead of the J1939 address claim message. In the function fnAddressClaimInit(*) the variable gstData.u8BroadcastRateAccess needed to be changed to .u8IntervalAccess. This change was made to be more consistent with NMEA naming convention.

* Fixed a bug in the function fnTP_DT_Tx() where the local variable 'i' was declared as a U_8. This caused the stack to enter an endless loop if a data transport session was initiated that contained more than 256 bytes. This variable has been declared as a U_16 such that it can handle an entire transport session.

Rev 1.06.00
Date: 09/20/08
Information:

* Fixed a bug where the variable u8StartByte is used to define the starting location within the body of a message, for a parameter that is to be transmitted. This variable is defined as a byte. A problem occurs when the parameter is in a multipacket message and the start byte needs to be greater than 255. The result of this bug is that the stack will only transmit packets during a transport session that contain up to 255 bytes. This problem was fixed by changing the variable u8StartByte to u16StartByte and defining the new variable as an unsigned short.

* Fixed a bug where the order of initialization for the fastpacket.c module was too low. This bug caused the fastpacket Tx message buffers to be initialized after another module had already setup its Tx data. The result of this bug was that the NMEA address claim message would not be broadcast upon being requested by the ISO request message.

* Fixed a bug where the NMEA NAME field parameters where not being entered into the NMEA transmit list buffers correctly. This bug would have caused a Fastpacket message to be broadcast with the NAME field during an address claim message. This would only happen if the request for an address claim was received. This bug was not seen because the bug mentioned above prevented the address claim message from being broadcast in the first place.

* Added the ability for the NMEA transmit messages to be able to transmit the same message directly to multiple destination addresses.

Rev 1.07.00
Date: 10/20/08
Information:

* Fixed a bug in the function fnCheckRxRequest() where the local variables i and j were not initialized at powerup. This bug will only affect the users of the stack who are using J1939 only. The variables need to be initialized to 0xFF. The result of this bug is that the Acknowledgement message is not consistently sent when a request is made for a PGN that is not supported.

* Fixed a bug where a request for address claim would cause the address claim message to be broadcast regardless of the stack state. The stack will now not accept a request message if it is not in bus active state.

* Fixed a bug where if a destination specific request message was received for a PGN that is multipacket, supported by the stack and globally broadcast the stack would respond by sending the message to the global address instead of to the address of the device issuing the request. The functions fnGetTPTxDataBuffer() and fnProcessTxMessage() were changed such that the passed variable u8DestinationAddress is used to determine what type of transport protocol to use and where to send the message.

Rev 1.08.00
Date: 06/08/09
Information:

*Fixed a bug where a parameter that is defined as less then 8 data bits in length would not be built correctly for transmitting. The function fnBuildData() was changed such that the variable *pTemp was used to access the data bits that needed to be used to build the message, instead of *pData.

Rev 1.09.00
Date: 10/07/09
Information:

*Changed the function fnTxAckMessage() such that it handles the new NMEA requirement where the Acknowledgment message is sent to the source address of the device that caused the acknowledgment message. J1939 still requires that the acknowledgment message always be sent to the global address.

*Fixed a bug were, if 2 fastpacket messages are received at the same time and each message has the same PGN but different source addresses only the first message received is accepted. The stack will now receive simultaneous fastpacket messages containing the same PGN from multiple source addresses.

Rev 1.10.00
Date: 12/01/09
Information:

* Fixed a bug where the transmission of two different sized transport protocol messages would cause both messages to be the same size after the first transmission of the larger of the two messages. Code was added to fnTP_DT_Tx() and fnCheckChannelTimeout() such that if more than one size message is broadcast during a power cycle the stack will now broadcast each message with the correct number of bytes.

Rev 1.11.00
Date: 07/02/10
Information:

*Fixed a bug in function fnBuildNMEAData() where, if the parameter that is being placed in a fast packet message spans more than 1 byte but does not perfectly span the bytes, the upper byte of the parameter will not be copied into the fastpacket message.

Rev 1.12.00
Date: 08/04/10
Information:

* Added the external function call fnRemoveTxMessageFromList() to the TxData.h file such that other modules outside the TxData.c file can have access to the function.

Rev 1.13.00
Date: 12/07/10
Information:

* Added the u8CANPort variable to each of the examples.

* Fixed a bug in Examples.c, fnDM1Init() where the wrong function was being called to load the DM1 message structure pointer into the stack.

* Fixed a bug in Examples.c, fnDM1Init() where the function that was called to start the DM1 message broadcast was not passed the CAN port to broadcast on.

* Added the channel APPLICATION_CHANNEL to the channel list in includes.h so the stack has a reference tag to associate with the messages that are to be received. The messages are loaded in the Examples.c file.

* Changed the variable name and size of u8BitLength to u16BitLength. The size was changed from a U8 to a U16. This allows for data type greater than 255 bits to be passed to the stack for Tx. Files affected are TxData.c, TxData.h, Examples.c and TestCode.c.

Rev 1.14.00
Date: 01/15/11
Information:

* Fixed a bug where if the stack sends a request for the address claim message to the global address it does not then transmit its address claim message. This has been fixed so that the stack will now Tx its address claim message when it sends a request for the address claim message to the global address.

*Added the use of multiple source addresses and multiple NAME fields. Now when multiple CAN ports are defined there is a separate source address and NAME field for each port.

Rev 1.15.00
Date: 02/11/11
Information:

* Added big endian functionality to the stack.

Rev 1.16.00
Date: 02/27/11
Information:

* Added the ability for the stack to affect only selected CAN ports. In previous versions, the stack would affect all CAN ports defined starting from 0 to n-1. For example CAN port 3 could not be selected as J1939 without the stack affecting CAN ports 1 & 2.

* Fixed a bug in the includes.h file where the const J1939_MANUFACTURER_CODE[] was defined as a U_8 instead of a U_16. This bug could cause incorrect manufacture codes values to be transmitted in the address claim message. This bug was introduced in Rev 1.14.00 of the stack with the addition of the multiple source addresses and multiple NAME fields feature.

Rev 1.17.00
Date: 03/07/11
Information:

*Changed the way NMEA messages are loaded into the stack. The old way required the 0^{th} data field to be loaded into the stack first. The new way allows the user to load the data fields in any order. This change affected functions in the fastpacket.c and .h modules.

Rev 1.18.00
Date: 05/07/11
Information:

*Fixed a bug where the response to the transport protocol, clear to send message, placed a 0 in the start packet field instead of a 1.

Rev 1.19.00
Date: 08/01/11
Information:

*Fixed a bug the stack would NAK a receive BAM message if the stack could not handle the message. The stack now does not respond to received BAM message.

Rev 1.20.00
Date: 08/20/11
Information:

* Fixed a bug in the fnCheckChannelTimeout() function where the u8CANPort was not set for a NAK on a timed out Rx connect session. This could cause an unintended memory access.

Rev 1.21.00
Date: 12/07/12
Information:

*Added the transport type to the function fnProcessTxMessage() in the function fnCheckRxRequest(). This fixes a bug where a request made to a specific address for a BAM PGN will be sent using the direct connect method of transport protocol instead of a BAM.

Rev 1.22.00
Date: 01/22/13
Information:

* Fixed a bug where a received CTS message from device with the incorrect source address or to the wrong destination address would cause the stack to send the DT packets for the message. A filter was added to the function fnTP_CM_CTS to stop this from happening.

Rev 1.23.00
Date: 02/22/13
Information:

*Added the ISO 11783 extended transport protocol to the stack.

Rev 1.24.00
Date: 04/01/13
Information:

* Fixed a bug where if a CTS message was received and it contained a value in the "clear to send number of bytes" field that was larger than the value that was in the RTS message that was sent prior, the transport protocol message would fail. An if statement was added to the function fnTP_CM_CTS where if the "clear to send number of bytes" value is larger than

the "requested number of bytes" value than the variable u8CTSNumberOfPackets is set equal to the "requested number of bytes" value instead of the "clear to send number of bytes" value.

*Fixed a bug introduced in V1.23 of the stack. This bug would cause the received data bytes for a single packet DM message to be shifted up by one byte. The extra variable u8DestinationAddress was added to the structure tTP_CM_SINGLE_PACKET so it would match the structure tTP_CM_RX on the TransportProtocol.h module. This fixed the bug.

Rev 1.25.00
Date: 02/12/14
Information:

*Fixed a bug where the stack would not respond correctly to an NMEA complex request for a piece of data that had instances. The stack was working off a zero based numbering system and NMEA uses a 1 based numbering system for identifying fields in a message. Now when a complex request message is made the field number is reduced by one to correctly index into the stack's buffer to identify a field.

Rev 1.26.00
Date: 05/23/14
Information:

*Changed the default priority of the DM1 and DM2 messages to 6.
*Fixed a bug where the EDP (extended data page) bit was not being taken into account when processing messages. This would cause messages with the EDP bit set to be acted upon when they should not.
*Fixed a bug where the stack would not respond to the request of address claim message after it claimed the NULL address.
*Added to proper reasoning for aborting a transport protocol session into the session about message.
*Fixed a bug in the connect transport protocol session where if a RTS it received during a session with the same PGN and from the same source address, the current session will be terminated and a new session will be started. The abort message will not be sent.

Rev 1.27.00
Date: 11/13/14
Information:

*Fixed a bug where the #define ETP_CM_TX_BUFFER_SIZE was set to TP_CM_TX_BUFFER_SIZE in three different places in the code. This caused memory access out of bounds errors if ETP_CM_TX_BUFFER_SIZE was set to a value greater than TP_CM_TX_BUFFER_SIZE.

Rev 1.28.00
Date: 12/12/14
Information:

*Fixed a bug during a connection mode transport session, if the receiving device only requests a subset of the entire message, when that subset is finished transmitting the enter message is erased and only 0xFFs are transmitted for the rest of the message.

Rev 1.29.00
Date: 2/9/15
Information:

*Added ISO 15765-2 transport protocol functionality to the stack.

Rev 1.30.00
Date: 3/23/15
Information:

*In the function fnRemoveTxMessageFromList() the variable gstTxMessage.u8NumberOfParameters is decremented twice for every parameter that is removed. This will cause the last parameter in the list to be removed by mistake.

Rev 1.31.00
Date: 9/14/15
Information:

*Added ISO 14229-1,-2,-3 server functionality to the stack.

*Fixed a bug where during an extended transport protocol session, if the receiving device sends a CTS message requesting more packets then what the session supports given the session's message size, the stack would send additional packets beyond the message size with zeros in the data fields. Code was added to fnETP_CM_CTS to fix this situation.

Rev 1.32.00
Date: 12/18/15
Information:

*Added ISO 15765 transport protocol to the stack transmit scheduler. This allows for messages that are loaded into the stack for cyclic transport to be able to use the ISO 15765 transport schema.

Rev 1.33.00
Date: 12/23/15
Information:

*Fixed a bug in the ISO 15765 extended transport protocol where a single packet transport would be classified as a first frame packet.

*Fixed some naming of some #defines that were labeled as ISO11765_ instead of ISO15765_.

Rev 1.34.00
Date: 01/19/16
Information:

*Changed the priority that is used by the transport protocol from a default of 6 to user defined.

*Changed the time delay between the Tx of the CM packet and the Tx of the first DT packet on the BAM transport protocol message from 100ms to 50ms.

Rev 1.35.00
Date: //
Information:

*Skip this release number

Rev 1.36.00
Date: 01/26/16
Information:

*Added the ability to set the CAN ID type to extended or standard.

Rev 1.37.00
Date: //
Information:

*Skip this release number

Rev 1.38.00
Date: 05/22/16
Information:

*Initialized the u8Active flag in the extended transport protocol when the message buffer is finished being used.

Rev 1.39.00
Date: 01/11/16
Information:

*Fixed a bug that was introduced in a previous version where the PDU specific value for a cyclically transmitted message was cleared when being loaded into the stack. This only occurred if the message PGN was above 0xFFFF.

Rev 1.40.00
Date: 04/12/17
Information:

*Fixed a bug where the index into the FMI array was using i instead of 0 as the index number. The result was that a random index would be used to check if the FMI at FMI array location 0 was 0.

Rev 1.41.00
Date: 01/23/18
Information:

*Added a u8FieldAccess variable to the NMEA 2000 Tx structure. This variable allows the programmer to set the accessibility of a given field within a message.

*Changed the way the NMEA 2000 stack handles the NMEA 2000 request message sent using the ISO transport protocol for a fastpacket message. If an NMEA 2000 request is received using the ISO transport protocol for a fastpacket message the response now uses the ISO transport protocol to respond instead of just sending the fastpacket message.

*Added a conditional statement to the Fastpacket.c module, fnProcessFastPacket() function to insure that a message could not be set a received twice in the event that a fastpacket message was received when all of the buffers were full.

Rev 1.42.00
Date: xx/xx/xx
Information:

*Skipped

Rev 1.43.00
Date: 01/23/18
Information:

*Fixed a bug in the NMEA 2000 complex request message where a request for an ISO message was ignored.

*Also fixed a bug where the time interval part of the complex request message was not being handled correctly.

Rev 1.44.00
Date: xx/xx/xx
Information:

*Skipped

Rev 1.45.00
Date: 01/23/18
Information:

*Added Product Information and Heartbeat message examples to the NMEA 2000 stack extension.

Rev 1.46.00
Date: 01/23/18
Information:

*Fixed a bug where physical CAN ports other than CAN 1 would fail to respond correctly to an NMEA 2000 complex request message.

Rev 1.47.00
Date: 03/25/21
Information:

*Added verbiage to the manual to indicate that the manual also supports ISO 15765 and ISO 14229.

4. Scope and Background Information

The purpose of this document is to provide documentation for the implementation of the DakotaSoft NMEA/J1939 Stack. This stack can be implemented on a J1939 network, NMEA 2000 network, or on a network that is using both protocols at the same time.

5. Getting Started

In order for the stack to operate properly with the CAN hardware several functions must be completed by the user of the stack. These functions, which reside in the module *Interrupt.c*, perform the initialization of the hardware and basic I/O of data between the stack and the hardware.

Once the functions in *Interrupt.c* have been completed the next step is to configure the stack for the different needed components of the J1939 and NMEA 2000 standard. Configuration of the stack takes place in the module *includes.h*. The configuration file is shipped from the factory with 100% of the capabilities of the stack enabled.

5.1 User Configurable Modules

There are two modules within the stack that will need to be configured or have code added to them by the user.

Interrupts.c – This module contains all of the low level drivers. The user will have to write the code that communicates with the physical layer of the CAN device. Within the functions of this module there is the statement **"//////////The user of this stack must complete the following code"**. Following this statement there is a description of what the user must do in order to complete the function. Within some of the functions there is code that is commented out. This code has been added to aid the user in completing the function. If the user wishes to use the code that has been commented out they will have to uncomment the code and use it as needed.

Includes.h – This module is where the user configures the functionality of the stack. It contains the **#defines** that set the size of the different queues that are used by the stack. It is also the place where the different channels for the stack are defined.

5.2 Interrupt.c Functions That Need To Be Completed

fnRxInterrupt – This is the interrupt that occurs when a new CAN message is received. The message must be copied into the structure provided and passed to the function **fnBufferRawRxData**. The structure that is provided may need to be removed from the interrupt function and made global depending on the requirements of the compiler being used.

fnSetDataForTx – This function initiates the transmission of a message on the CAN bus. The message is copied from the passed structure into the transmit buffers. The hardware is then enabled to begin transmission.

fnTxMessagePending – This function checks the transmit buffer to see if there is a message that is pending transmission. If there is a message in the transmit buffer the function returns a **TRUE** (logic 1). If there is not a message waiting in the transmit buffer then the function returns a **FALSE** (logic 0).

fnConfigureHardwareFilters – This function sets up the CAN ID hardware filters and masks.

The code provided with this function takes the passed list of 29 bit identifiers and masks for those identifiers and calculates the most intrusive filter that will still allow the list of identifiers to pass though the hardware. It is left up to the user of the stack to determine if this existing code is useful for their needs.

fnStopAllCANTx – This function stops the transmission of a message on the bus and clears the transmit queue of any pending messages. This function is mainly used for situations where the stack switches from the active state to the prestart state. If the CAN hardware does not allow a transmission to be interrupted once it has started then do nothing to this function.

fnGetCurrentTime – This function returns the current time of the stack or system clock.

A timing mechanism is needed to run the stacks timers. These timers are used to keep track of time critical event within the stack. The clock that is provided should be at least a 4 byte value and tick no slower than once every 50ms. If there

are messages that need to be transmitted at a faster rate than once every 50ms then the clock should be set to tick at a rate at least as fast as the fastest message that needs to be transmitted. Hence if there is a message that needs to be transmitted every 10ms then the tick rate should be at least 1 tick every 10ms.

5.3 CAN Stack Configuration Settings

The configuration settings consist of a series of compiler switches that allow the user to include or exclude code at compile time based on the needed operations of the stack. The stack currently has the following configuration switches located in *includes.h*:

MS_PER_CLOCK_TICK – This is the number of milliseconds between ticks for the stack or system clock. The tick value is that which is returned by a call to the function **fnGetCurrentTime** which is located in *Interrupt.c*.

TX_DM1 – includes code to transmit DM1 messages.
TX_DM2 – includes code to transmit DM2 messages.
RX_DM1 – includes code to receive DM1 messages.
RX_DM2 – includes code to receive DM2 messages.
TX_DM3 – includes code to transmit the DM3 message.
TP_CM_BAM_RX – includes code to receive a transport protocol – broadcast announce message.
TP_CM_BAM_TX – includes code to transmit a transport protocol – broadcast announce message.
TP_CM_CONNECT_RX – includes code to receive a transport protocol – direct connection message.
TP_CM_CONNECT_TX – includes code to transmit a transport protocol – direct connection message.
ETP_CM_CONNECT_RX – includes code to receive an extended transport protocol message.
ETP_CM_CONNECT_TX – includes code to transmit an extended transport protocol message.
ISO15765_FIXED_MIXED – includes code for transmitting and receiving ISO15765 Normal Fixed and Mixed transport protocol messages.
ISO15765_NORMAL_EXTENDED – includes code for transmitting and receiving ISO15765 Normal and Extended transport protocol messages.
ISO14229_BASE – includes code for the base functionality of ISO14229.
REQUEST – includes code to receive and transmit the request message.
ADDRESS_CLAIM – includes code to receive and transmit the address claim message.
ACKNOWLEDGMENT – includes code to receive and transmit the acknowledgment message.
FASTPACKET – includes code to receive and transmit NMEA messages using the fastpacket protocol.
REQUEST_COMMAND_ACK – includes code to receive and transmit the NMEA complex request, command and acknowledgment messages.
TRANSMIT_PGNS – includes code to transmit the receive and transmit PGNs list.
TEST_CODE – includes code for the system test routines and flags.

enum eCHANNELS – This is where the stack channels are defined. Adding an item to this **enum** will add another channel to the stack. There can be as many as 16 different channels defined by the **enum**.

RX_DATA_BUFFER_SIZE – This is the size of the temporary receive queue that holds any messages from the receive interrupt.

TX_DATA_BUFFER_SIZE – This is the size of the temporary transmit queue that holds any messages that are ready to be loaded into the transmit buffer.

CHANNEL_BUFFER_SIZE – This is the number of messages each channel can have waiting for it at any given time. This is simply an array of pointers which point into the receive queue messages that the channel needs.

MESSAGE_BUFFER_SIZE – This is the total number of received messages that can be waiting for channel use. Each single packet message that passes through the CAN hardware/software masks and filters and has a channel that wants this data is buffered here.

ADDRESS_CLAIM_J1939 – Including this #define at compile time will allow the J1939 NAME field #defines to be included in the compile.

ADDRESS_CLAIM_NMEA - Including this #define at compile time will allow the NMEA 2000 NAME field #defines to be included in the compile.

CAN_PORT_TYPE – A constant array that defines what protocols address claim NAME will be used with each physical CAN port.

CAN_PORT_NUMBER – A constant array that defines what physical CAN port is assigned to each logical CAN port in memory.

PREFERRED_ADDRESS – This is the first address the software will try to claim at power up if the address claim functionality of the stack is operational. If address claiming is not operational then the stack will use this address for all communications.

COMMANDED_ADDRESS – This will enable or disable the receipt of the commanded address message that is defined in J1939-81. This can only be defined if the receive multipacket protocol (TP_CM_BAM_RX) has also been defined.

ARBITRARY_ACCESS_CAPABLE – This will enable or disable dynamic source addressing. If set to 1 then the software will act following the rules under J1939-81 for dynamic addressing. If set to 0 the software will act under the rules for static addressing.

ADDRESS_CLAIM_START_ADDRESS – If **ARBITRARY_ACCESS_CAPABLE** is enable (== 1) then this is the next address that will be claimed if the **PREFERRED_ADDRESS** is not available.

ADDRESS_CLAIM_END_ADDRESS – This is the last address that an attempt will be made to claim. All addresses between **ADDRESS_CLAIM_START_ADDRESS** and **ADDRESS_CLAIM_END_ADDRESS** will be tried first. If an address cannot be claimed the stack will claim the cannot claim address (254) and change the stack state to bus off.

MAX_CAN_PORTS – This is the number of CAN ports that are connected to the stack.

The following statements are used to configure the **NAME** field of the device for J1939 address claiming. Refer to the J1939-81 section on the **NAME** field for information on how to configure each statement.
J1939_ARBITRARY_ACCESS_CAPABLE
J1939_INDUSTRY_GROUP
VEHICAL_SYSTEM_INSTANCE
VEHICAL_SYSTEM
J1939_FUNCTION
FUNCTION_INSTANCE
ECU_INSTANCE
J1939_MANUFACTURE_CODE
IDENTITY_NUMBER

The following statements are used to configure the **NAME** field of the device for NMEA 2000 address claiming. Refer to the NMEA 2000 standard, section on the address claim message for information on how to configure each statement.
NMEA_ARBITRARY_ACCESS_CAPABLE
NMEA_INDUSTRY_GROUP
SYSTEM_INSTANCE
DEVICE_CLASS
NMEA_FUNCTION
DEVICE_INSTANCE_HIGH
DEVICE_INSTANCE_LOW
NMEA_MANUFACTURE_CODE
UNIQUE_NUMBER

TX_REQUEST_BUFFER_SIZE – This is the number of messages that can be waiting for the J1939 request message. This is simply an array of indexes into the cyclic transmit list of message.

RX_REQUEST_TIME – This is the amount of time the stack will wait between its requests for a particular message.

eREQUEST_TYPES – This enum defines the different types of requests that an Rx message can be setup for.
Not Requested – No request message is generated for this Rx message.
J1939 Request – A J1939 request message is generated for this Rx message.
NMEA Request – An NMEA 2000 request message is generated for this Rx message.

RX_ACK_BUFFER_SIZE – This is the number of ACK messages that can be received within **RX_ACK_TIME_OUT** seconds. Also known as the size of the ACK receive queue.

RX_ACK_TIME_OUT – This is the timeout time for a given ACK message that has been stored in the ACK receive queue. After the ACK message has timed out the queue location will be used for storing another received ACK message.

RX_BAM_BUFFER_SIZE – this is the number of simultaneous receive BAM sessions that can be open at one time.

RX_CONNECT_BUFFER_SIZE – This is the number of simultaneous receive multipacket direct connect sessions that can be open at one time.

TX_BAM_BUFFER_SIZE – This is the number to transmit BAM messages that can be queued for broadcast. Note: Only one message can be sent at a time.

TX_CONNECT_BUFFER_SIZE – This is the number of simultaneous transmit multipacket direct connect sessions that can be open at one time.

TP_CM_RX_BUFFER_SIZE – This is the number of bytes that can be buffered for each Rx multipacket session. This number must be at least 8.

TP_CM_TX_BUFFER_SIZE – This is the number of bytes that can be buffered for each Tx multipacket session. This number must be at least 8.

ERX_CONNECT_BUFFER_SIZE – This is the number of simultaneous extended connection Rx multipacket sessions that can be open at one time.

ETX_CONNECT_BUFFER_SIZE – This is the number of simultaneous extended connection Tx multipacket sessions that can be open at one time.

ETP_CM_RX_BUFFER_SIZE – This is the number of bytes that can be buffered for each extended Rx multipacket session. This number should never be less than 1786.

ETP_CM_TX_BUFFER_SIZE – This is the number of bytes that can be buffered for each extended Tx multipacket session. This number should never be less than 1786.

eTRANSPORT_TYPE – This is an enum that contains the definitions for the u8TransportType variable.

PROTECTION_LAMP – This defines that the protection lamp variable will be compiled into the software.

WARNING_LAMP – This defines that the warning lamp variable will be compiled into the software.

STOP_LAMP – This defines that the stop lamp variable will be compiled into the software.

MALFUNCTION_LAMP – This defines that the malfunction lamp variable will be compiled into the software.

CONVERSION_METHOD – This defines what conversion method the diagnostic module will use to code and decode DM1 and DM2 fault messages that are received and transmitted over the bus. Conversion method 4 is always active if the conversion method bit in the DM1 or DM2 message is set. Otherwise the conversion method that is used is defined by this statement.

TX_DM1_FAULTS_BUFFER_SIZE – This defines how many transmit DM1 fault codes can be buffered.

DM1_BROADCAST_RATE – This is the cyclic rate at which the DM1 message will be broadcast.

MAX_DM1_SESSIONS – This is the number of simultaneous receive DM1 sessions that can be active at one time.

RX_DM1_FAULTS_BUFFER_SIZE – This defines how many receive DM1 fault codes can be buffered.

DM1_TIMEOUT – This is the amount of time that the stack will wait after receiving a DM1 message from a particular source address before recycling the buffer and using it for the receipt of a different DM1 message.

MAX_DM2_SESSIONS – This is the number of simultaneous receive DM2 sessions that can be active at one time.

RX_DM1_FAULTS_BUFFER_SIZE – This defines how many receive DM2 fault codes can be buffered.

DM2_TIMEOUT – This is the amount of time that the stack will wait after receiving a DM2 message from a particular source address before recycling the buffer and using it for the receipt of a different DM2 message.

TX_PARAMETER_BUFFER_SIZE – This is the number of parameters that can be buffered for transmission at a cyclic rate.

TX_PARAMETER_GROUP_BUFFER_SIZE – This is the number of parameter groups (PGNs) that can be buffered for cyclic transmission.

ID_BUFFER_SIZE – This is the total number of different IDs that are requested from every channel. For instance two channels are defined “DISPLAY” and “TEST”. If “DISPLAY” has 10 IDs that it needs to receive and “TEST” needs 5 IDs that are different from the “DISPLAY” channel and 5 that are the same, then ID_BUFFER_SIZE should be set to at least 15.

REQUEST_COMMAND_ACK_MODULE0
MAX_RX_FASTPACKET_SESSIONS – This is the total number of different Rx fastpacket messages that can be stored simultaneously.

RX_FASTPACKET_QUEUE_SIZE – This is the size of each fastpacket storage queue. This should be set to the size of the largest fastpacket message that will be received.

MAX_TX_FASTPACKET_PARAMETER_GROUPS – This is the total number of different parameter groups that are awaiting broadcast either cyclically or by request.

MAX_FASTPACKET_PARAMETERS – This is the maximum number of parameters that can be loaded into any given Tx parameter group. This number should represent the number of parameters that are contained within the largest parameter group.

MAX_TX_FASTPACKET_SIZE – This is the maximum number of bytes that the largest Tx fastpacket message will consume. This is used to setup the size of a buffer that is located on the user stack. Thus this memory is only used temporarily when a fastpacket message is being built and then is released.

eDATA_TYPE – This is an enum statement that contains the definitions for the u8DataType variable.

RX_ACK_PARAMETER_BUFFER_SIZE - This is the number of parameters that the acknowledgment message can relay error codes for. This should be set to the number of parameters requested or commanded by the largest request or commanded message that is sent to a specific source address (not global address).

RX_ACK_NMEA_BUFFER_SIZE – This is the number of received acknowledgment messages that can be buffered at one time.

ePGN_ERROR_CODE – This is an enum that contains the definitions for the PGN error codes that the acknowledgment message uses.

eTRANS_PRIORITY_ERROR_CODE – This is an enum that contains the definitions for the change Broadcast Rate and Priority error codes that the acknowledgment messages uses.

ePARAMETER_ERROR_CODE – This is an enum that contains the definitions for the request/commanded parameters that the acknowledgment message uses.

TRANSMIT_PGNS_BUFFER_SIZE – This is the number of Tx or Rx PGNs that can be transmitted by the Transmit PGNs message. This number should be set to the larger of the two: Number of Tx PGNs or Number of Rx PGNs.

The compiled size of the J1939 stack will change based on the #define configuration switches used.

ISO15765_TIMEOUT - This is the amount of time that transport protocol session will remain open if there is no packets received and packets are expected.

TIME_BETWEEN_CONSECUTIVE_PACKETS - This is the delay time between consecutive packets that the flow control message will relay to the transmitting node. The delay time is in milliseconds. 0 - 127 is in milliseconds; 241 - 249 = (Value - 240) * .1ms, so a value of 241 would = .1ms

FLOW_CONTROL_BLOCK_SIZE - This is the number of packets that will be sent during a transport protocol session without a flow control packet separating the sequence. 0 = All packets will be sent without a flow control packet breaking up the consecutive packets. Any other number will cause a flow control packet to be broadcast after the number of consecutive packets indicated in the FLOW_CONTROL_BLOCK_SIZE is received.

ISO15765_NUMBER_OF_WAITS - The maximum number of consecutive flow control wait messages received before aborting the transmission of a message.

eISO15765_ERRORS – This enum defines the different types of errors that are possible using the ISO15765 transport protocol.

ISO15765_NORMAL_FIXED_PHYSICAL - Compiler switch for including code for the normal fixed physical addressing method.

ISO15765_NORMAL_FIXED_FUNCTIONAL - Compiler switch for including code for the normal fixed functional addressing method.

ISO15765_MIXED_PHYSICAL - Compiler switch for including code for the mixed physical addressing method.

ISO15765_MIXED_FUNCTIONAL - Compiler switch for including code for the mixed functional addressing method.

ISO15765_TX_FIXED_MIXED_MESSAGES - This is the number of normal fixed or mixed transmit messages that can be buffered.

ISO15765_TX_FIXED_MIXED_DATA - This is the number of bytes of normal fixed or mixed data that can be buffered for each transmit message.

ISO15765_RX_FIXED_MIXED_MESSAGES - This is the number of normal fixed or mixed receive messages that can be buffered.

ISO15765_RX_FIXED_MIXED_DATA - This is the number of bytes of normal fixed or mixed data that can be buffered for each receive message.

ISO15765_NORMAL - Compiler switch for including code for the normal physical and functional addressing method.

ISO15765_EXTENDED - Compiler switch for including code for the extended physical and functional addressing method.

ISO15765_RX_NORMAL_EXTENDED_RAW_MESSAGE - This is the number of normal or extended receive messages that can be buffered.

ISO15765_RX_NORMAL_EXTENDED_RAW_DATA - This is the number of bytes of normal or extended data that can be buffered for the receive message.

ISO15765_RX_NORMAL_EXTENDED_INFO_MESSAGES - This is the number of different normal and extended CAN IDs that can be buffered and looked for on the CAN bus.

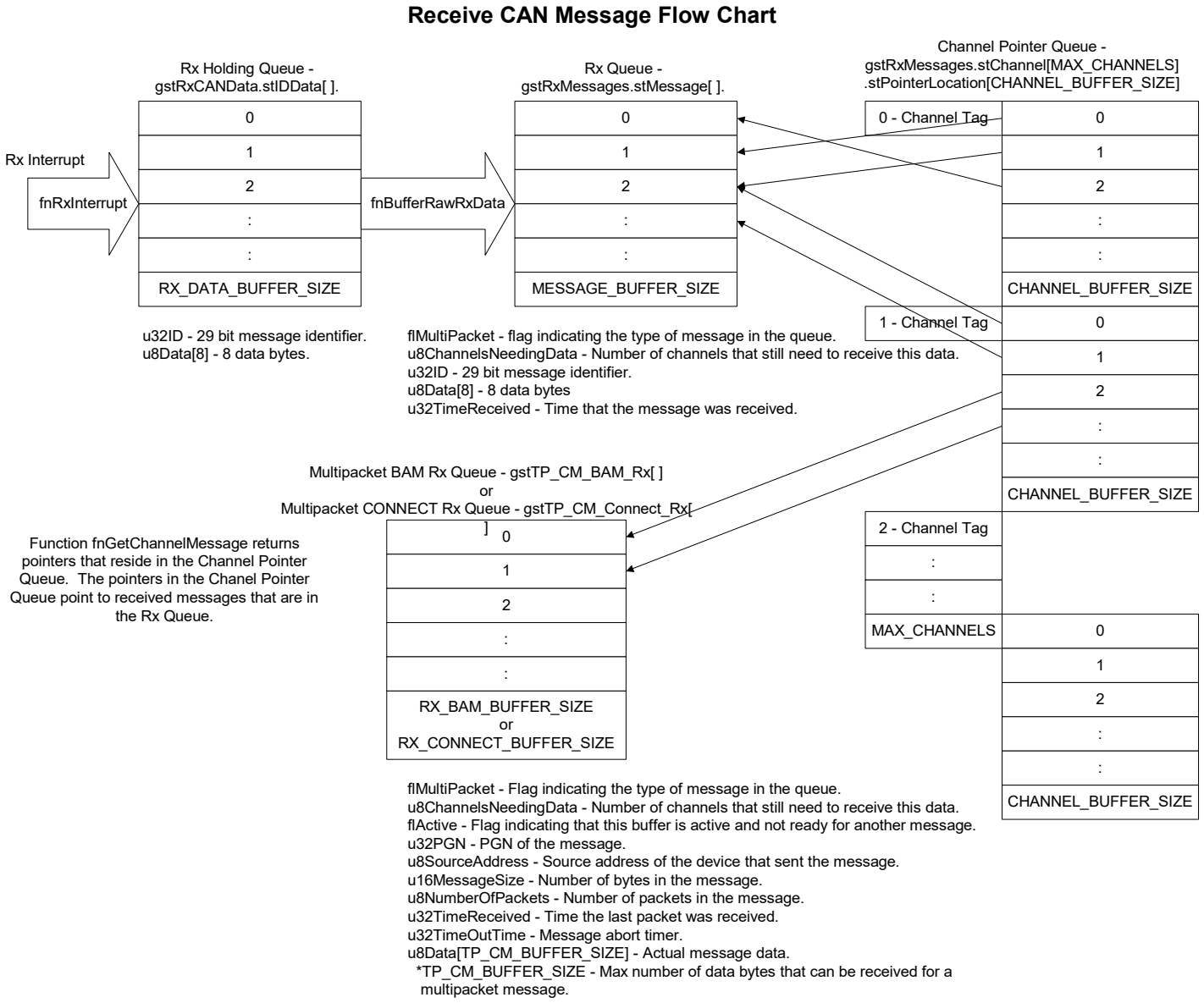
ISO15765_RX_NORMAL_EXTENDED_INFO_MESSAGES - This is the number of normal and extended messages that can be buffered for transmission.

ISO15765_TX_NORMAL_EXTENDED_INFO_DATA - This is the number of bytes that can be buffered for each normal or extended message.

ISO15765_EXTENDED_ADDRESS - This is the extended address used in the mixed addressing mode.

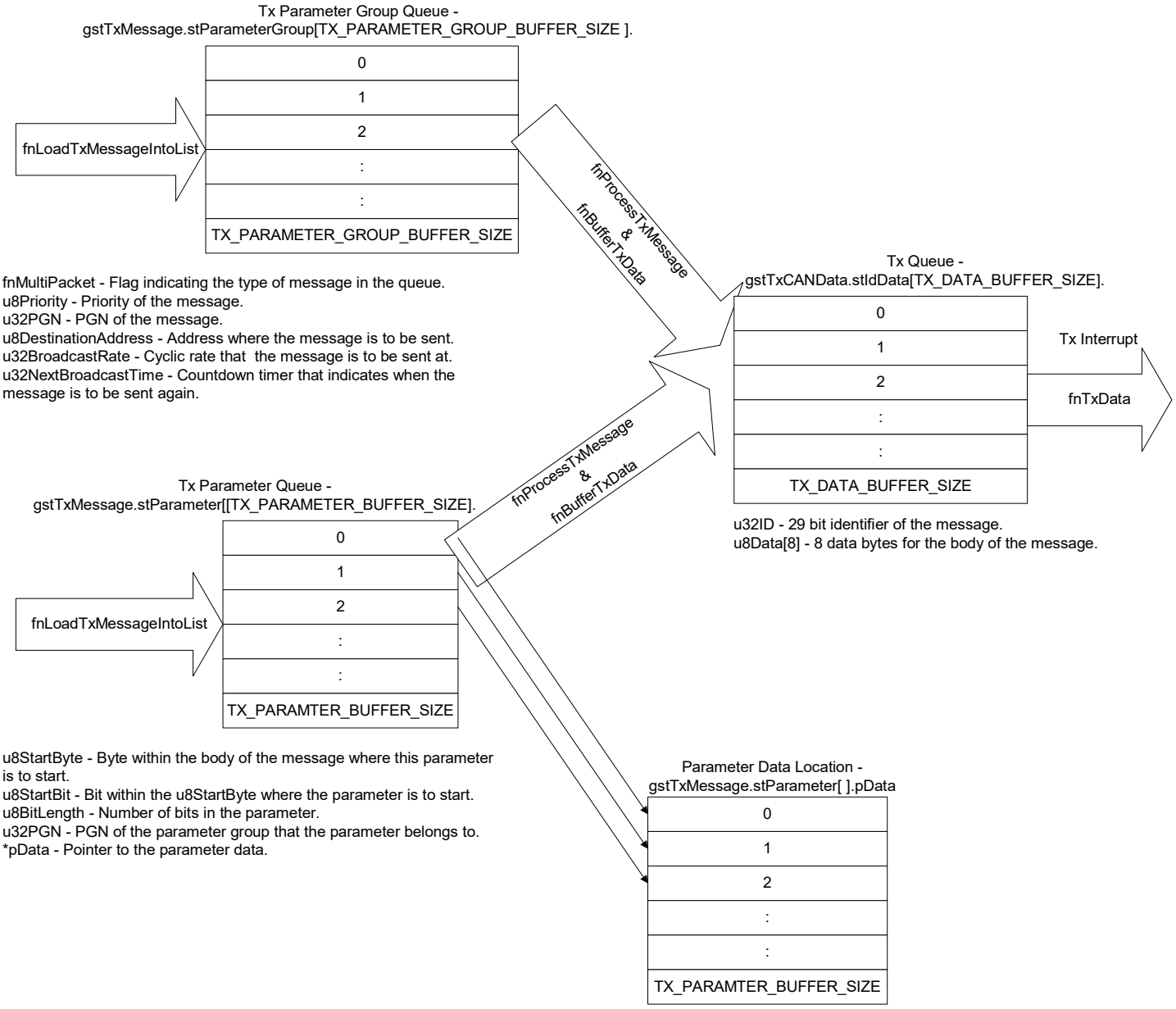
6. System Flowcharts

6.1 Message Receipt Flow Chart



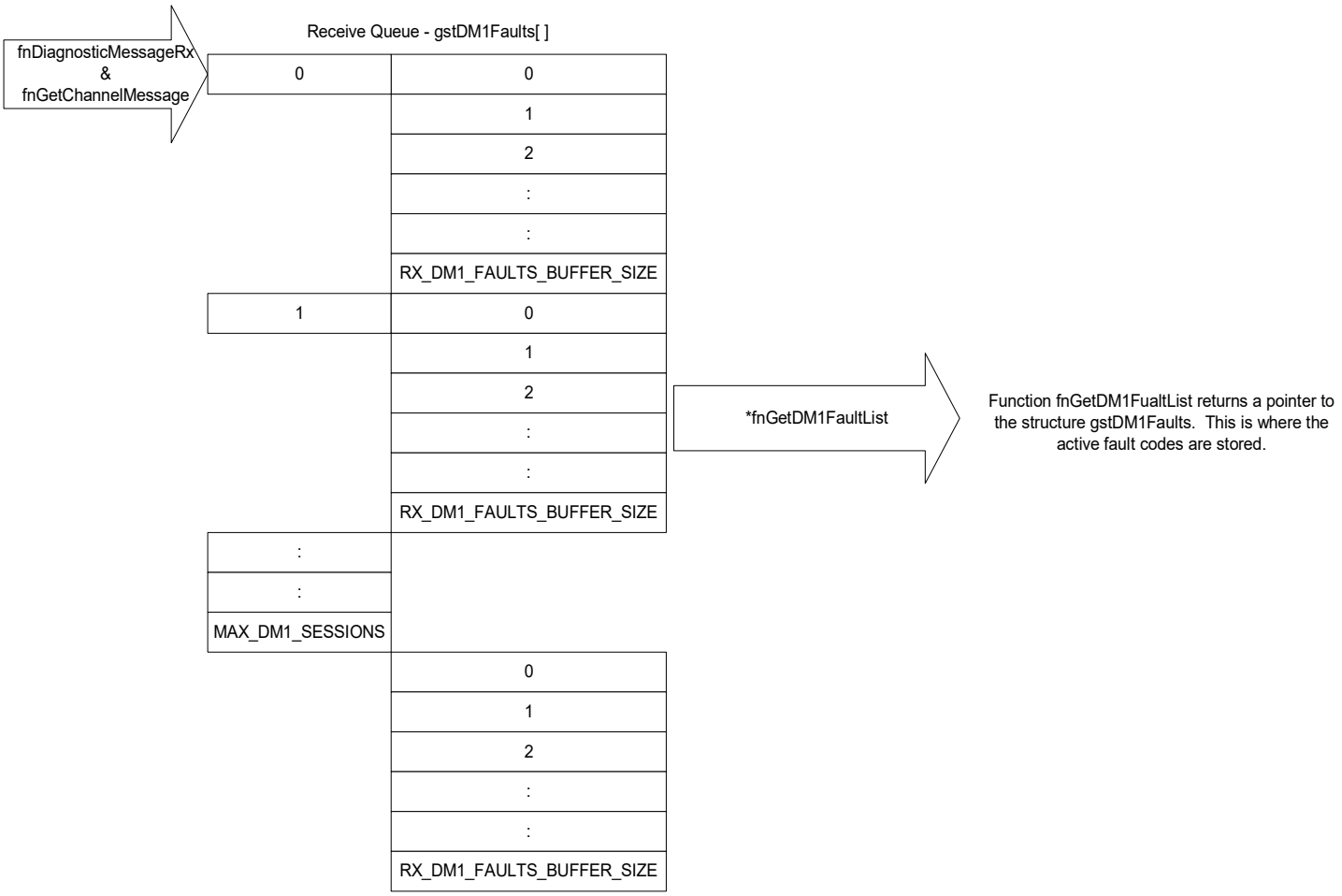
6.2 Message Transmit Flow Chart

Transmit CAN Message Flow Chart



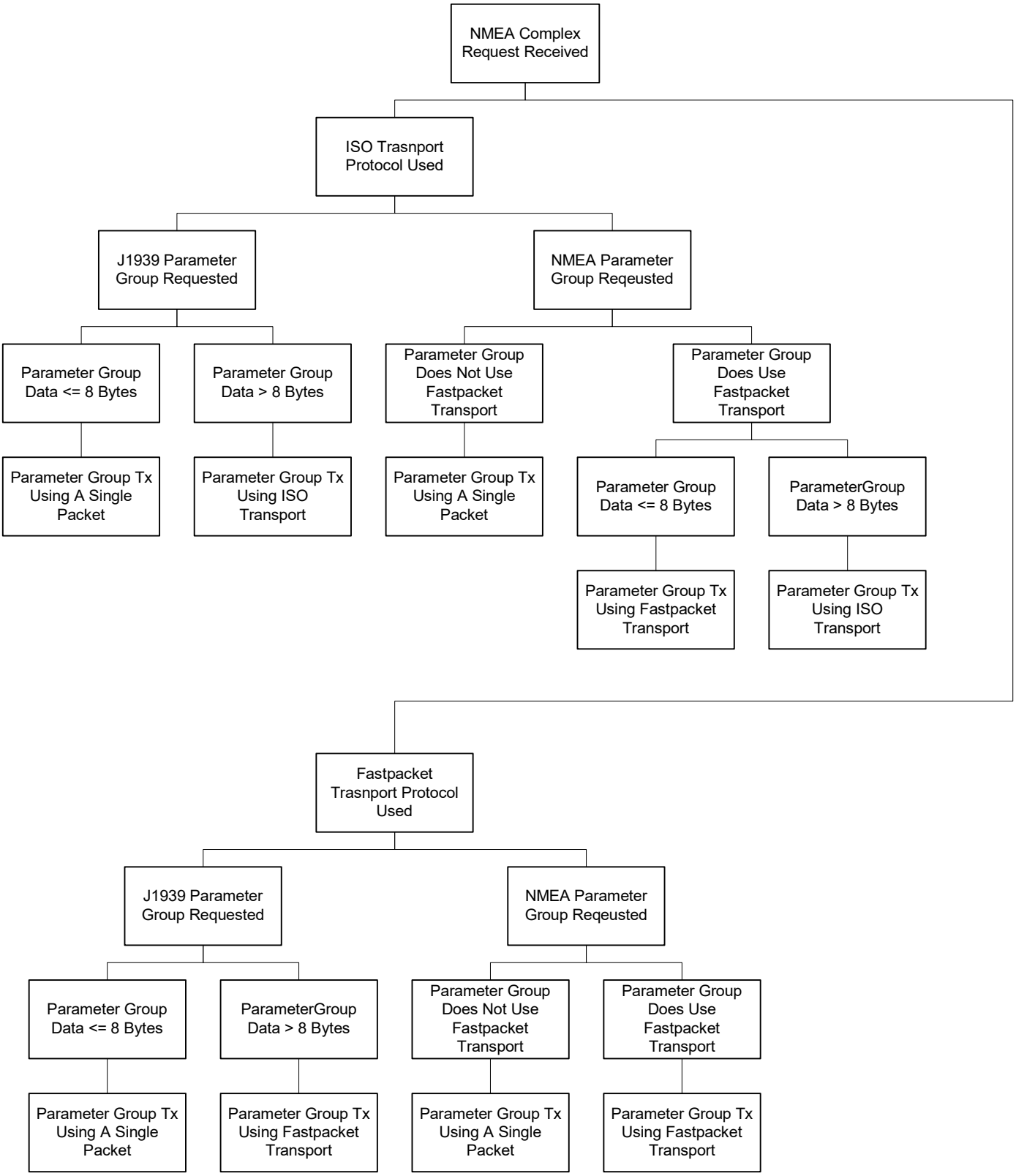
6.3 Receive Diagnostic Message 1 Flow Chart

Receive Diagnostic Message 1 Flow Chart



u8NumberOfFaults - Number of faults in the list.
u8ProtectLampStatus - The status of the protect lamp
u8WarningLampStatus - The status of the warning lamp
u8StopLampStatus - The status of the stop lamp
u8MalfunctionLampStatus - The status of the malfunction lamp
u32TimeRecieved - The time the last packet of the message was received.
u8SourceAddress - The source address of the device that sent the fault list.
stFaults[ ] - Array of the structures that contains the following list of elements:
  u32SPN - Suspect Perarameter Number
  u8FMI - Failure Mode Indicator
  u8OccurrenctCount - Occurrence Count

6.4 NMEA Complex Request Message Receipt Flow Chart



7. CAN Layer

7.1 Channels

Channels are the conduits through which receive messages are passed from the receive queue to the consumers of those messages. There are channels that are internal to the stack and channels that are external. The external channels are those defined by the user. A channel tag is assigned to every message received into the receive queue. Channel tags make it possible to associate the messages with the channels that need them.

7.1.1 Creating A Channel

Channels are created by adding an entry in the **enum tCHANNELS** which resides in the *includes.h* module. The **enum** entry **MAX_CHANNELS** must always be the last channel in this **enum** although it is not considered a channel. The J1939 stack is currently capable to having 16 different channels defined (i.e. there can be as many as 16 different entries in **enum tCHANNELS** beside the **MAX_CHANNELS** entry). All of the current entries are for internal stack use. They are encapsulated in **#if**s and are included in the compilation of the code if the configuration switch for that channel is defined. **The user must enter any additional channels that are needed for the application software here.**

7.2 Stack States

There are 3 states the CAN stack can be in. At power up the stack is in the prestart state. In this state the stack can only send the address claim message and receive messages on the CAN bus. It will remain in this state until the stack has claimed an address via the Address Claim message. After claiming an address the stack will enter the active state. In this state the stack can send and receive any message. If during the address claim procedure the stack claims the bus off address then the stack will enter the bus off state. In this state the stack cannot send any messages but it can receive them. If the stack is not setup to support the address claim message, then it will enter the active state at power up and will not change states.

7.3 Connecting Multiple CAN Ports

The stack is capable of simultaneously handling connections from multiple CAN ports. The **#define MAX_CAN_PORTS** which resides in *includes.h* defines the number of CAN ports that the stack is setup to handle. Almost every data handling structure in the stack has a field called **.u8CANPortIndex** that is used to route received message data to the correct channel and route transmitted messages to the correct CAN port.

7.4 Rx Interrupt Architecture

The function **fnBufferRawRxData()** should be called from the Rx interrupt. This function places the CAN data received by the interrupt into a temporary queue. This queue is circular in fashion and is configurable in size. The **#define RX_DATA_BUFFER_SIZE** which resides in *includes.h* can be adjusted to increase or decrease the size of the queue. The queue should be set to a value at least as large as the maximum number of CAN messages that are expected to be received between calls to the function **fnProcessRxQueue()**. Calling function **fnProcessRxQueue()** will process all of the messages in the queue.

7.5 Tx Interrupt Architecture

The function **fnBufferTxData()** should be called to place a new message into the transmit queue. This queue is circular in fashion and is configurable in size. The **#define TX_DATA_BUFFER_SIZE** which resides in *includes.h* can be adjusted to increase or decrease the size of this queue. This queue should be set to a value at least large enough to hold

the number of messages passed to it between calls to the function **fnTxData()**. Calling function **fnTxData()** will load the next message in the transmit queue into the transmit buffer and initiate the transmit interrupt. Function **fnTxData()** is automatically called at the end of function **fnBufferTxData ()** if the CAN stack is in an active state.

Function **fnTxData()** will not load a new message into the transmit buffer if there is a message pending transmission. It may be a good idea to setup an interrupt that is triggered by the completion of the transmission of the current message in the transmit buffer. Within this interrupt call **fnTxData()** to start the transmission of the next message in the transmit queue if there are any.

7.6 Rx Queue

The receive queue is a circular storage buffer where the received CAN messages are sorted and assigned channel tags for the channels that are expecting the messages. If a received message cannot be assigned to a channel then it is discarded. The size of this queue can be adjusted up or down by changing the **#define MESSAGE_BUFFER_SIZE** which resides in *includes.h*. An overflow of this queue will cause the overflowed messages to be discarded.

7.6.1 Retrieving Messages From The Rx Queue

Messages can be retrieved from the receive queue by calling the function **fnGetChannelMessage()** and passing it the tag for the channel that needs the message. This function returns a **void** pointer to the first message in the queue that is assigned to the channel who's tag was passed. Once the message has been processed by the calling function the queue location containing the message needs to be released. This is done by calling function **fnReleaseChannelMessage()** and passing it the channel tag for the channel that received the message. If there are no other channel tags assigned to this message then the queue location will be made ready to store another incoming message. If **fnGetChannelMessage()** is called twice and passed the same channel tag without calling **fnReleaseChannelMessage()** in between then the pointer that is returned for both calls will be to the same message in the receive queue. If there are no messages remaining in the receive queue for the passed channel tag then a pointer to the **NULL** address is returned.

7.7 Address Claim Message

The address claim message is meant to resolve address conflicts between different devices on the bus. It is broadcast at the power up of the device and claims an address for the device by following standard J1939-81 and NMEA2000. Setting **#define ADDRESS_CLAIM** in the *includes.h* file to 1 will include the code at compile time for implementing the address claim message.

7.7.1 Multiple CAN port setup

The stack can be configured to handle multiple CAN ports. Use the following **const** arrays to setup the order and functionality of each CAN port.

extern const U_8 CAN_PORT_NUMBER[] = { 0, 1 }; // Physical CAN port assignment
extern const U_8 CAN_PORT_TYPE[] = { J1939_PORT, NMEA_PORT }; // CAN port type

The **CAN_PORT NUMBER[] const** allows for defining the physical location of each CAN port.

The **CAN_PORT_TYPE[] const** allows for the setting of how the **NAME** field will be built for each CAN port. Setting the type to **J1939_PORT** will cause the stack to build a J1939 **NAME** field for the CAN port. Setting the type to **NMEA_PORT** will cause the stack to build an NMEA **NAME** field for the CAN port.

If it is desired for the stack to be applied to CAN ports 1 and 3 but not 0 and 2 and for port 1 to use the J1939 **NAME** field and port 3 to use the NMEA **NAME** field, then set **CAN_PORT NUMBER[]** and **CAN_PORT_TYPE[]** as follows:

extern const U_8 CAN_PORT_NUMBER[] = { 1, 3 }; // Physical CAN port assignment
extern const U_8 CAN_PORT_TYPE[] = { J1939_PORT, NMEA_PORT }; // CAN port type

7.7.2 J1939 NAME Field

The **NAME** field is used by the stack to resolve source address conflicts. Two devices that claim the same source address are in conflict and one of the devices will have to change its address. The device with the lower priority **NAME** field is the device what will have to change (according to J1939-81).

The elements of the **NAME** field for the stack are defined in the file *includes.h*. The **const** that are associated with this field are as follows:

```
const U_8 J1939_ARBITRARY_ACCESS_CAPABLE[] = { a , b };
const U_8 J1939_INDUSTRY_GROUP[] = { a , b };
 const U_8 VEHICAL_SYSTEM_INSTANCE[] = { a , b };
 const U_8 VEHICAL_SYSTEM[] = { a , b };
 const U_8 J1939_RESERVED[] = { a , b };
 const U_8 J1939_FUNCTION[] = { a , b };
 const U_8 FUNCTION INSTANCE[] = { a , b };
 const U_8 ECU_INSTANCE[] = { a , b };
 const U_16 J1939_MANUFACTURE_CODE[] = { a , b };
 const U_32 IDENTITY_NUMBER[] = { a , b };
```

Notice that the **const NAME** field values are defined as arrays. This is so you can setup a separate NAME field for each CAN port that is defined in **CAN_PORT_NUMBER[]**. Each NAME field array must contain as may values as that are values in **CAN_PORT_NUMBER[],** even if the **CAN_PORT_TYPE[]** is defines as **NMEA_PORT**. For example if the first value in **CAN_PORT_TYPE[]** is **NMEA_PORT** and the second is **J1939_PORT** then set the **'a'** value in each of the **NAME** field arrays to 0 as a place holder.

7.7.3 NMEA NAME Field

The **NAME** field is used by the stack to resolve source address conflicts. Two devices that claim the same source address are in conflict and one of the devices will have to change its address. The device with the lower priority **NAME** field is the device what will have to change (NMEA 2000).

The elements of the **NAME** field for the stack are defined in the file *includes.h*. The **const** that are associated with this field are as follows:

```
const U_8  NMEA_ARBITRARY_ACCESS_CAPABLE[] = { a, b };
const U_8  NMEA_INDUSTRY_GROUP[] = { a, b };
const U_8  SYSTEM_INSTANCE[] = { a, b };
const U_8  DEVICE_CLASS[] = { a, b };
const U_8  DOMINANT_BIT[] = { a, b };
const U_8  NMEA_FUNCTION[] = { a, b };
const U_8  DEVICE_INSTANCE_HIGH[] = { a, b };
const U_8  DEVICE_INSTANCE_LOW[] = { a, b };
const U_16 NMEA_MANUFACTURE_CODE[] = { a, b };
const U_32 UNIQUE_NUMBER[] = { a, b };
```

Notice that the **const NAME** field values are defined as arrays. This is so you can setup a separate NAME field for each CAN port that is defined in **CAN_PORT_NUMBER[]**. Each NAME field array must contain as may values as that are values in **CAN_PORT_NUMBER[],** even if the **CAN_PORT_TYPE[]** is defines as **J1939_PORT**. For example if the first value in **CAN_PORT_TYPE[]** is **J1939_PORT** and the second is **NMEA_PORT** then set the **'a'** value in each of the **NAME** field arrays to 0 as a place holder.

7.7.4 Static And Dynamic Source Addressing

The stack is capable of being set up for either static or dynamic addressing. The **#define ARBITRARY_ACCESS_CAPABLE** which is located in *includes.h*, is used to determine which type of addressing is used. A setting of 1 will cause dynamic source addressing to be used. A setting of 0 will cause static source addressing to be used.

The following **#defines** which are located in *includes.h* are used by the stack to determine the order of the addresses that are to be claimed.

#define PREFERRED_ADDRESS
#define ADDRESS_CLAIM_START_ADDRESS
#define ADDRESS_CLAIM_END_ADDRESS

The stack will broadcast the address claim message by first using the **#define PREFERRED_ADDRESS**. If that address has been claimed by a different device and the **NAME** field of the other device has a higher priority, then the stack will rebroadcast the address claim message by claiming the **#define ADDRESS_CLAIM_START_ADDRESS**. If this address has also been claimed the stack will increment to the next address. This process will continue until the stack has claimed an address between **#define ADDRESS_CLAIM_START_ADDRESS** and **#define ADDRESS_CLAIM_END_ADDRESS.**

If the **#define ARBITRARY_ACCESS_CAPABLE** is set to 0 then the stack will start by claiming the address **#define PREFERRED_ADDRESS**. If this address has been claimed and the stack has the lower priority, then the J1939-81 cannot claim address will be claimed (address 254).

If it is wished that the claimed address of the device be saved in non-volatile memory than this should be done after the address has been successfully claimed. Function **fnAddressClaim** in *AddressClaim.c* is a good place to do this.

Loading this saved address as the first address that the stack claims at power up should be done in the function **fnAddressClaimInit** which is located in *AddressClaim.c*.

7.8 J1939 Commanded Address Message

If the **#define COMMANED_ADDRESS** which is located in *includes.h* is set to 1 at compile time, then the stack will look for and react to the receipt of the commanded address message. After receiving the commanded address message and changing its source address the stack will then change its state to the prestart state and broadcast the new address using the address claim message.

7.9 J1939 Request Message

If the **#define REQUEST** which is located in *includes.h* is set to 1 at compile time then the stack will include the code to broadcast and receive the requested message. The request message asks another device on the bus to broadcast a particular message or responds to other devices requests by broadcasting the message that is requested.

7.10 J1939 Acknowledgment Message

If the **#define ACKNOWLEDGMENT** which is located in *includes.h*, is set to 1 at compile time then the stack will include the code to broadcast and receive the acknowledgment message.

7.11 J1939 Transport Protocol

The transport protocol is a means of transferring messages over the bus where the body of the message contains more then 8 bytes.

If the **#define TP_CM_BAM_RX** in *includes.h* is set to 1 at compile time then the stack will include the code to receive a broadcast announce message.

If the **#define TP_CM_BAM_TX** in *includes.h* is set to 1 at compile time then the stack will include the code to transmit a broadcast announce message.

If the **#define TP_CM_CONNECT_RX** in *includes.h* is set to 1 at compile time then the stack will include the code to receive a direct connect message.

If the **#define TP_CM_CONNECT_TX** in *includes.h* is set to 1 at compile time then the stack will include the code to transmit a direct connect message.

7.12 Diagnostic Messaging

Currently the J1939 stack supports the following diagnostic messages:

DM1 – Active fault codes
DM2 – Previously active fault codes
DM3 – Clearing of previously active fault codes

If the **#define TX_DM1** in *includes.h* is set to 1 at compile time then the stack will include the code to transmit active fault codes.

If the **#define TX_DM2** in *includes.h* is set to 1 at compile time then the stack will include the code to transmit previously active fault codes.

Note: The **#define REQUEST** must also be set to 1 at compile time to include the code that receives the request for the broadcast of previously active fault codes.

If the **#define RX_DM1** in *includes.h* is set to 1 at compile time then the stack will include the code to receive active fault codes.

If the **#define RX_DM2** in *includes.h* is set to 1 at compile time then the stack will include the code to receive previously active fault codes.

Note: The **#define REQUEST** must also be set to 1 at compile time to include the code that transmits the request for the broadcast of previously active fault codes.

If the **#define TX_DM3** in *includes.h* is set to 1 at compile time then the stack will include the code to transmit a request to clear previously active fault codes.

Note: The **#define REQUEST** must also be set to 1 at compile time to include the code that builds the request message.

8. CAN Modules

8.1 StackMainLoop.c

This module contains the stack's main initialization and processing functions.

8.1.1 void fnStackInit()

This function initializes the stack's buffers and operations and must be called at power-up before any other stack related function is called.

Calling this function during operation will cause the stack to erase all application setting that have been loaded into the stack.

8.1.2 void fnStackMainLoop()

This function handles all of the run time operations of the stack. It needs to be called cyclically at a rate at least as fast as the fastest cyclically broadcast message or at a rate of no less than 50ms if the transport protocol is used.

The faster this function is called the more efficient the stack is and the smaller the received raw message buffer will need to be.

8.2 Interrupt.c

This module contains the function that the user must physically setup for the stack to operate properly. These functions deal primarily with hardware setting that have to be configured for the particular CAN device being used.

8.2.1 void fnInterruptInit()

This function contains the initialization of any variables that are global to this module. It is also where the Tx and Rx buffers and interrupts are initialized. Basically any hardware initialization of the stack should be done here.

8.2.2 void fnRxInterrupt()

This function is the interrupt that is called when a CAN message is received. **The user must write the code that copies the 29 bit identifier, time received and 8 data bytes of the received CAN message into the provided structure.** The existing code in this function has been provided to help in this requirement. The structure is then passes to another function for storage. **The user may have to change the name of the function to suite the requirements of the CAN device that the stack is implemented for.**

8.2.3 void fnSetDataForTx(tTX_ID_DATA *)

This function places the passed message into the Tx interrupt buffer and starts the transmission. **The user must write the code that writes the data from the passed structure into the proper Tx buffer and then initiate the transmission.** The existing code in this function was provided to aid in this requirement.

8.2.4 BOOLEAN fnTxMessagePending(u8CANPortIndex)

This function checks the Tx buffer for a message pending transmission. **The user must write the code that checks the Tx buffer and returns a TRUE (logic 1) if there is a message pending or FALSE (logic 0) if there is no message pending.**

8.2.5 void fnConfigureHardwareFilters(u8BufferNumber, tID_LIST1 *)

This function sets up the hardware filters for the CAN Rx buffers. The exiting code is provided to calculate the most intrusive filter and mask schema that will allow all of the messages that are contained in the passed list (tID_LIST) into a single Rx buffer. The variable u8BufferNumber that is passed to the function is a representation of the Rx buffer the filter is to be applied to. By passing u8BufferNumber this function can be used to setup multiple Rx buffers each with a different list of messages that will be allowed to pass.

The user must write the code that applies the calculated hardware filter and mask to the correct buffer. The user must also call this function during initialization or at the appropriate time to set the Rx buffer filters and masks.

8.2.6 void fnStopAllCANTx(u8CANPortIndex)

This function clears all of the pending messages in the Tx queue. **The user must write the code that stops the current transmission of any message in the Tx buffer.**

This function is used by the AddressClaim.c module for handling arbitration of address claim messages that have identical NAME fields. Although the J1939 standard states that no two devices on the bus shall have the same NAME. It is recognized that this condition may arise when using multiple like products in which there is no means of manually changing the NAME field or source address.

Some hardware platforms my not allow a message that is in the Tx buffer to be aborted once the transmission is in progress. In this case there is nothing to be done but allow the transmission to complete.

If the NAME fields of all the devices on the bus are different, then there is no need to add any software to stop a transmission of a message in the Tx buffer.

8.2.7 BOOLEAN fnPortStatus(u8CANPortIndex)

This function returns a TRUE (logic 1) if there is a problem with the part number that is passed to the function. A FALSE (logic 0) is returned if there are no error status bits set for the passed port number.

8.2.8 U_32 fnGetCurrentTime()

This function returns the current stack clock value. The stack clock is an interrupt driven timer that increments an unsigned long variable every interrupt cycle. This clock should tick no slower than once every 50ms and at least as fast as the fastest message that is to be transmitted.

8.3 TxRxDrivers.c

This module contains the functions that store the received CAN messages and the list of CAN messages that are pending transmission on the CAN bus.

8.3.1 Globals

tRX_CAN_DATA gstRxCANData – This is a temporary storage queue that holds any received CAN message. The messages in this queue are the latest received from the CAN bus and have not yet been assigned to a channel.

tTX_ID_DATA gstTxCANData – This is a temporary storage queue for the messages that are waiting to be sent on the CAN bus.

8.3.2 void fnTxRxDriverInit()

This function does the initialization of any variables that are global to this module.

8.3.3 void fnBufferRawRxData(tID_DATA *)

This function is used to store the raw CAN messages received over the CAN bus in a temporary queue. The passed structure contains the raw message and is stored in the structure **gstRxCANData**. This structures size can be adjusted by changing the value of the **#define RX_DATA_BUFFER_SIZE** which is located in the module *includes.h*.

8.3.4 tRX_CAN_DATA *fnGetRawCANData()

This function returns a pointer to the raw messages queue. This function is used by any function external to this module to access the raw messages located in the structure **gstRxCANData**.

8.3.5 BOOLEAN fnBufferTxData(tID_DATA *, u8DLC, u32Timeout)

This function is used to load the temporary Tx queue with a message. Messages reside in this queue until the Tx buffer is free to send another message on the CAN bus. The message that is passed to this function is saved in the structure **gstTxCANData**. This structures size can be adjusted by changing the value of the **#define TX_DATA_BUFFER_SIZE** which is located in the module *includes.h*. If the state of the CAN bus is not bus active then the message will not be saved to the temporary Tx queue. If for any reason a message cannot be saved to the temporary queue the function will return a FALSE (logic 0) else it will return a TRUE (logic 1).

The variable **u8DLC** is the data length code for the message that is to be transmitted. This variable is used to adjust the number of data bytes that are transmitted in the body of the message.

The variable **u32Timeout** is the system time that must be reached before the message will be transmitted.

8.3.6 BOOLEAN fnTxData()

This function is used to send the next message in the temporary Tx queue to the function that will load it into the Tx buffer. This function is called every time the function **fnBufferTxData** is called. It should also be called at a cyclic rate to ensure that any messages in the temporary Tx queue are sent. If there is a message that is already in the Tx buffer pending transmission or there are no messages in the temporary Tx queue this function will return a FALSE (logic 0) indicating that no message was sent. If a message was sent to the Tx buffer the function adjusts the queue pointers and returns a TRUE (logic 1).

8.3.7 tTX_ID_DATA *fnGetTxData

This function returns a pointer to the list of messages that are pending broadcast.

8.3.8 void fnClearTxBuffer(u8CANPortIndex);

This function clears the transmit buffer of any messages that are waiting to be broadcast on the passed CAN port.

8.4 LoadId.c

8.4.1 Globals

tID_LIST1 gstIdList – This is the storage buffer for the ID list. The ID list is made up of the information about every receive CAN message that the different channels need.

8.4.2 void fnLoadIdInit()

This function does the initialization of any variables that are global to this module.

8.4.3 BOOLEAN fnLoadIdList(tID_LIST *)

Each channel that requires messages from the CAN bus must call this function and pass to it a list of needed messages. When a message is received on the CAN bus this list is consulted. The channels that want the received message have their channel tags attached to the message. When a channel calls for received messages, it is only passed messages that have its channel tag attached to them.

If the storage buffer containing the message list overflows then this function will return a **FALSE** (logic 0). If the list of messages is successfully stored the function will return a **TRUE** (logic 1).

Messages are only stored once in the list. If more than one channel requests the same message, channel tags for each channel are attached to the needed message. There can be up to 16 different channel tags attached to a given message.

The size of the storage buffer is adjusted by changing the size of the **#define ID_BUFFER_SIZE** located in *includes.h*.

8.4.4 BOOLEAN fnSetJ1939Request(u32Id, u8Request)

This function sets the request flag for the message with the passed ID. The request flag tells the stack that the message is needed by a channel and that that message needs to be requested. The request variable is the type of request being made (J1939, NMEA2000).

Note: Even though the message is a J1939 defined message it can use the NMEA2000 complex request message for requesting.

8.4.5 BOOLEAN fnCheckMessageAgainstIdList(u32Id,u8CANPortIndex)

This function uses the passed ID to check the ID list that was created by **fnLoadIdList** for the IDs presence. If the ID list contains the passed ID then the function returns a **TRUE** (logic 1) else it returns a **FALSE** (logic 0).

8.4.6 tID_LIST *fnGetIdList()

This function returns a pointer to the ID list created by **fnLoadIdList**. This function is used by functions external to the module to gain access to the ID list.

8.5 CANEngine.c

This module contains the function that attaches channel tags to any received CAN messages.

8.5.1 Globals

tRX_MESSAGE gstRxMessages – This is the storage queue for any received messages after the channel tags have been assigned to them.

8.5.2 void fnCANEngineInit()

This function does the initialization of any variables that are global to this module.

8.5.3 void fnProcessRxQueue()

This function processes any messages that are stored in the temporary receive queue. Messages from the temporary receive queue are copied to the receive queue, structure **gstRxMessages**, and assigned channel tags. Once this function is called all of the messages in the temporary receive queue will be processed. If the receive queue overflows, the function will stop retrieving messages from the temporary receive queue and set a warning flag by calling function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. The receive queue size can be adjusted by changing the **#define MESSAGE_BUFFER_SIZE** which is located in the module *includes.h*.

Each channel has a limit to the number of messages that can be waiting for it. This number can be adjusted by changing the **#define CHANNEL_BUFFER_SIZE** located in *includes.h*. If the channel pointer buffer overflows, a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 2.

8.5.4 void *fnGetChannelMessage(u8Channel)

This function returns a pointer too the requesting channel for the next message in the receive queue. If there are no messages in the receive queue, a **NULL** pointer is returned. This function is meant to be used in conjunction with function **fnReleaseChannelMessage**. If this function is called twice without calling function **fnReleaseChannelMessage** in between, the pointer returned will be the same as it was for the previous call.

8.5.5 BOOLEAN fnReleaseChannelMessage(u8Channel)

This function uses the passed channel number to erases the channel tag connected to the next message in the receive queue that is for the passed channel. This function is meant to be used in conjunction with function **fnGetChannelMessage**. If this function is not called between calls to the function **fnGetChannelMessage**, then the pointer returned to the calling function for **fnGetChannelMessage** will be the same for both calls.

If after erasing the channel tag connected to the message, there are no other tags attached to the message, the receive queue location will be available to receive a new message.

If the channel number passed to the function has not been defined or there is no tag attached to a message for the passed channel tag, the function will return a **FALSE** (logic 0). If a tag was found for the passed channel then the function will return a **TRUE** (logic 1).

If the channel tag passed to the function has not been defined a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 5.

If the channel passed to the function does not have any messages pending in the receive queue a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 3.

Note: It is very important that each channel attends to its received messages in a timely fashion or the buffer will overflow and messages will be lost.

8.5.6 void fnSetMessageAsReceived(void *)

This function is used to place a pointer to a received multipacket message into the channel pointer buffer. The calling function passes a pointer to the data concerning the multipacket message. **fnSetMessageAsReceived** then looks to see if this message is needed by any of the channels. If it is needed, the pointer is saved in the channel pointer buffer and the tag/s of the channel/s that need the message is attached to the queue location.

If there are no channels that need the message that is passed, then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 4.

If there is a channel that requires this message but no space is available in the channel pointer buffer, then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 0.

9. J1939 Modules

9.1 AddressClaim.c

This module contains the function that handle the J1939 address claim and commanded address messages.

9.1.1 Globals

J1939 globals

U_8 gu8SourceAddres – This is the source address that this device is using.
U_8 gu8StackState – This is the current state of the stack. See the section on stack states for more detail.
U_32 gu32AddressClaimTimeout – This variable keeps track of the time since the address claim message was last sent.

NMEA2000 globals
U_8 u8ArbitraryAccessCapable – NMEA2000 NAME Field
U_8 u8IndustryGroup – NMEA2000 NAME Field
U_8 u8SystemInstance – NMEA2000 NAME Field
U_8 u8DeviceClass – NMEA2000 NAME Field
U_8 u8Function – NMEA2000 NAME Field
U_8 u8DeviceInstance – NMEA2000 NAME Field
U_16 u16ManufatureCode – NMEA2000 NAME Field
U_32 u32UniqueNumber – NMEA2000 NAME Field

9.1.2 void fnAddressClaimInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

9.1.3 void fnAddressClaim()

This function receives and directs any messages that deal with the source address of the stack to their proper destination.

The state of the stack is also determined from within this function. **When the stack state transitions to the bus active state it would be a good time to store the source address, if it has changed, to non-volatile memory**. See the section on stack states for more detail.

9.1.4 BOOLEAN fnCheckAddressClaim(tMESSAGE *)

This function checks any received address claim message source addresses and **NAME** fields to determine if an address conflict is present. If there is a address conflict and the stacks **NAME** field has higher priority, the stack will rebroadcast the address claim message with its current source address.

If the stacks **NAME** field has a lower priority then the stack will do the following:

- If the **#define ARBITRARY_ACCESS_CAPABLE** located in *includes.h* is set to 1 then the source address of the stack will change.
- If the **#define ARBITRARY_ACCESS_CAPABLE** located in *includes.h* is set to 0 then the stack will send out the cannot claim address message.

If the source address changes then the stack will be returned to prestart and the function will return a **TRUE** (logic 1) indicating that the source address has changed. If the source address did not change then the function returns a **FALSE** (logic 0) indicating that the source address has not changed.

9.1.5 void fnGetIdName(tID_DATA *, u8CANPort)

This function fills the data structure who's pointer is passed to it with the address claim ID and **NAME** field for the stack.

9.1.6 void fnSetAddressClaimTimeout(u32Timeout, u8CANPort)

This function sets the address claim timeout time to the time passed to the function.

9.1.7 U_8 *fnGetNAME(u8CANPortIndex, *pau8NAME)

This function returns a pointer to the **NAME** field that is being used by the stack.

9.1.8 BOOLEAN fnCheckCommandedAddress(TP_CM_RX *)

This function handles the receipt of the commanded address message. If the **NAME** field within the commanded address messages and the one the stack is using are the same then the stack will use the commanded source address to rebroadcast the address claim message. The state of the stack will be returned to prestart and the function will return a **TRUE** (logic 1) indicating that the source address has changed. If the **NAME** field does not match that of the stack then the function returns a **FALSE** (logic 0) indicating that the source address has not changed.

9.1.9 U_8 fnGetSourceAddress(u8CANPortIndex)

This function returns the source address of the stack to the calling function. A 0xFF is returned if a request is made for a CAN port that has not been defined in the stack.

9.1.10 U_8 fnGetStackState(u8CANPortIndex)

This function returns the current state of the stack to the calling function. A 0xFF is returned if a request is made for a CAN port that has not been defined in the stack.

9.1.11 void fnRandomDelay(u8CANPortIndex)

This function starts a random delay between 0 and 153ms. The stack timer must be incremented by an interrupt since the program counter will not exit this function until the delay time has elapsed.

9.2 Acknowledgment.c

This module contains the functions that handle the J1939 acknowledgment message.

9.2.1 Globals

tACK_MESSAGE gstAckMessage – Storage buffer for the received acknowledgment message.

9.2.2 void fnAcknowledgmentInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

9.2.3 void fnRxACKMessage()

This function saves any received acknowledgment messages and releases the messages in the receive queue. If the receive acknowledgment message queue overflows a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 0. The receive acknowledgment message queue size can be adjusted by changing the **#define RX_ACK_BUFFER_SIZE** which is located in *includes.h*.

9.2.4 tACK_MESSAGE *fnGetAckMessage(u32PGN, u8CANPortIndex)

This function checks the current queue of received acknowledgment messages for a match to the passed PGN. If a match is found a pointer to the received acknowledgment message is returned to the calling function. If a match is not found then a **NULL** is returned.

Any acknowledgment messages that is received will remain an active message until released by the **fnReleaseACKMessage** function or until it has timed out. The timeout time period is configured by changing the value of the **#define RX_ACK_TIME_OUT** which is located in *includes.h*. The **#define** is a multiplier of the stack clock frequency. Setting the **#define** to a large number (0xFFFFFFFF) will effectively disable the timeout.

9.2.5 void fnTxACKMessage(u8MessageType, u32PGN, u8CANPortIndex)

This function configures and transmits the ACK message. The passed **u8MessageType** defines what type of acknowledgment message is to be sent. The variable u32PGN is the PGN for the message that is to be acknowledged. If the transmit queue is full a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1.

9.3 Request.c

This module contains the functions that handle the J1939 request message.

9.3.1 Globals

tRX_REQUEST_LIST gstRxRequestList – Storage buffer for the received request messages.

9.3.2 void fnRequestInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

9.3.3 void fnCheckRxRequest()

This function processes any received messages for the request channel.

9.3.4 void fnCheckTxRequest()

This function checks to see if any of the messages in the receive list, that need to have a request sent in order for that message to be broadcast, have timed out and are in need of being requested again. The **#define RX_REQUEST_TIME** is the timing factor that determines how often a request is made. **RX_REQUEST_TIME** is multiplied by the time per tick of the stack clock to calculate the duration. **RX_REQUEST_TIME** can be found in *includes.h*.

If the time between calls to this function is greater than the **RX_REQUEST_TIME x (stack clock time per tick)** then some of the messages being requested during one loop though this function may be duplicated. In this case the duplicate requests are thrown out and a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 0. The best way to remedy this situation is by calling the function **fnCheckTxRequest** more often. Increasing the request time multiplier **RX_REQUEST_TIME** will also slow the rate of transmission of the cyclic request messages.

9.3.5 BOOLEAN fnTxRequest(u32PGN, u8DestinationAddress, u8CANPortIndex)

This function uses the passed PGN and destination address to build the request message. It then places the request message in the transmit queue. If the transmit queue overflows a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. Increasing the size of the **#define TX_DATA_BUFFER_SIZE**, which is located in *includes.h*, will increase the size of the transmit queue.

9.3.6 tRX_REQUEST_LIST *fnGetRxRequestList()

This function returns a pointer to the structure **gstRxRequestList**. This structure contains a list of the messages that need to be requested at a cyclic rate. This list is built by the function **fnLoadIdList**, which is located in *LoadId.c*. **fnLoadIdList** is called by the channels who pass to it, a list of the messages that they need from the CAN bus.

9.4 TransportProtocol.c

This module contains the functions that handle the J1939 transport protocol message.

9.4.1 Globals

tTP_CM_Rx gstTP_CM_BAM_Rx – Storage buffer for any received broadcast announce messages. This buffer can be adjusted by changing the size of the **#define RX_BAM_BUFFER_SIZE** which is located in *includes.h*.

tTP_CM_Rx gstTP_CM_Conncet_Rx – Storage buffer for any received direct connection messages. This buffer can be adjusted by changing the size of the **#define RX_CONNECT_BUFFER_SIZE** which is located in *includes.h*.

tTP_CM_Tx gstTP_CM_BAM_Rx – Storage buffer for any transmit broadcast announce messages. This buffer can be adjusted by changing the size of the **#define TX_BAM_BUFFER_SIZE** which is located in *includes.h*.

tTP_CM_Tx gstTP_CM_CONNCET_Tx – Storage buffer for any received broadcast announce messages. This buffer can be adjusted by changing the size of the **#define TX_CONNECT_BUFFER_SIZE** which is located in *includes.h*.

9.4.2 void fnTP_CMInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

9.4.3 void fnTransportProtocol()

This function receives the transport protocol messages and directs them to the correct functions.

9.4.4 void fnCheckChannelTimeout()

This function handles the timing for the different types of transport protocol sessions. The four types of sessions are as follows:

Broadcast announce Rx
Broadcast announce Tx
Direct connect Rx
Direct connect Tx

9.4.5 void fnTP_CM_ACK_Comm_Abort(tMESSAGE *)

This function processes the receipt of the End of Message ACK and Communication Abort messages. The direct connect Tx session that corresponds to the passed receive message is aborted and its buffer is made ready for the next direct connect Tx session.

9.4.6 void fnTP_CM_CTS(tMESSAGE *)

This function processes the receipt of a Clear to Send message. If the message is not a “hold message”, the starting packet number and number of packets to send are saved and the session abort timer is set.

9.4.7 void fnTP_CM_BAM_RTS_Rx(tMESSAGE *)

This function processes the receipt of the Broadcast Announce and Request to Send messages.

The broadcast announce message causes the stack to initialize and activate a broadcast announce receive session for the source address of the sending device. If all of the BAM receive buffers are currently being used a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 0. Changing the size of the **#define RX_BAM_BUFFER_SIZE**, which is located in *includes.h*, will allow you to change the number of simultaneous receive broadcast announce sessions that can be active.

The request to send message causes the stack to initialize and activate a direct connect receive session for the source address of the sending device. If all of the direct connect receive buffers are currently being used a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. Changing the size of the **#define RX_CONNECT_BUFFER_SIZE**, which is located in *includes.h*, will allow you to change the number of simultaneous receive direct connect sessions that can be active.

9.4.8 void fnTP_CM_Tx(u8TP_CM_Type, tMESSAGE *)

This function sets up the Request to Send, Broadcast Announce, Clear to Send, and Communication Abort messages for transmission. The variable **u8TP_CM_Type** indicates what type of message is to be built. The pointer passed to this function points to the rest of the data that is needed to complete the building of the various types of messages.

Message type **TP_CM_RTS** builds a direct connect session request that is sent to a specific source address.

Message type **TP_CM_BAM** builds a broadcast announce packet that indicates what the contents of the forth coming broadcast message is.

Message type **TP_CM_CTS** builds a clear to send message that tells the device that has initiated a direct connect session how many packets to send and the packet sequence number to start with. This stack always starts at packet 0 and clears the sending device to deliver all of the packets in the message at once.

Message type **TP_CM_Comm_Abort** builds a communications abort message that tells the device that currently has a direct connect session open with the stack to abort the session.

9.4.9 void fnTP_DT_Tx(tMESSAGE *)

This function handles the transmission of the data transport portion of the multipacket broadcast announce and direct connect messages. Once the final packet has been loaded into the transmit queue the transport session is reinitialized for the next session. If an overflow of the transmit queue occurs a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 2. To increase the size of this transmit queue, change the value of the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.c*.

9.4.10 void fnTP_DT_Rx(tMESSAGE *)

This function handle the receipt of the data transport portion of the multipacket broadcast announce and direct connect messages. Once the final packet has been received the session is marked as complete and the function **fnSetMessageAsReceived** is called to place the message in the receive queue. If the received message is longer than the receive session buffer a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 3. If only part of the last packet will fit in the receive session buffer a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 4. This does not mean that the message was not received. It only means that part of the last packet was not saved as there was not enough buffer space. The part of the last packet that was discarded should be the part that contains 0xFFs which indicates that the data locations within the packet are not used. Changing the size of the receive and transmit session buffers can be accomplished by changing the value of the **#define TP_CM_BUFFER_SIZE** which is located in *includes.h*.

9.4.11 BOOLEAN fnRemoveFromTxDataList(u32PGN, u8CANPortIndex)

This function removes the message containing the passed PGN from the direct connect or broadcast announce transmit queue. If the queue contains the message with the PGN that is passed a **TRUE** (logic 1) is returned. If the message with the passed PGN cannot be found then a **FALSE** (logic 0) is returned.

9.4.12 tTP_CM_Tx *fnGetTPTxDataBuffer(u32PGN, u8DestinationAddress)

This function finds an empty broadcast announce or direct connect buffer, determined by the passed PGN and destination address, and returns a pointer to that buffer. If an empty buffer cannot be found a **NULL** is returned.

9.5 TxData.c

This module contains the functions that handle messages that are to be broadcast on the CAN bus either once or at a cyclic rate.

9.5.1 Globals

tTX_MESSAGE – Storage buffer for messages that are to be broadcast on the CAN bus.

U_8 gu8TxDataTestFlags – This is 1 byte of data flags that concerns the operation of this module. This variable is only included in the compilation of the software if the complier switch **TEST_CODE** is defined. **TEST_CODE** is located in the module *includes.h*.

9.5.2 void fnTxDataInit()

This function does the initialization of any variables that are global to this module.

9.5.3 BOOLEAN fnLoadTxMessageIntoList(tMESSAGE_INFO *)

This function loads the cyclic transmit queue with the messages that are to be broadcast at a cyclic rate and those messages that are to be broadcast upon request.

The passed message is loaded into a cyclic transmit message queue. The calling function passes a pointer to a memory location where the data to be transmitted is stored. Changing the contents of that memory location will automatically change the contents of the message that is transmitted.

If a PGN is passed to this function that has a broadcast rate of 0 then the message will not be sent at a cyclic rate. This message will remain in the buffer and not be broadcast unless a request is received over the CAN bus for that PGN.

If the cyclic transmit buffer containing the parameter group information overflows, then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. The size of this buffer may be changed by changing the **#define TX_PARAMETER_GROUP_BUFFER_SIZE** which is located in *includes.h*.

If the cyclic transmit buffer containing the parameter information overflows then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. The size of this buffer may be changed by changing the **#define TX_PARAMETER_BUFFER_SIZE** which is located in *includes.h*.

Not all parameters for a message need to be loaded into the stack. Any parameters that are not loaded into the stack will be set to all binary 1's ("Data Not Available", as per J1939 standard).

9.5.4 BOOLEAN fnRemoveTxMessageFromList(u32PGN,u8CANPortIndex)

This function removes a parameter group from the list of messages to be transmitted. The function uses the passed PGN to find the correct message and removes it along with all of its parameters from the queue. If the user only wants to remove some of the parameters from the transmit list but leave others then they must remove all of the parameters by calling this function and then call the function **fnLoadTxMessageIntoList** to reload the needed parameters.

If PGN that is passed does not match any of the PGNs in the transmit queue the function will return a **FALSE** (logic 0). If a PGN match is found the function will return a **TRUE** (logic 1).

9.5.5 void fnCheckTxData()

This function checks the messages in the cyclic transmit queue and activates any that are ready to broadcast. This function needs to be called at a rate at least as fast as the fastest parameter group that is set to broadcast cyclically.

9.5.6 void fnProcessTxMessage(u8Index, u8DestinationAddress, u32Timeout)

This function loads a message from the cyclic transmit queue into the transmit buffer.

If the message is a multipacket message the data is loaded into the multipacket transmit queue pointed to by the return of the function **fnGetTPTxDataBuffer**. If **fnGetTPTxDataBuffer** returns a **NULL** then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 2. The size of the multipacket transmit queue can be changed by changing the **#define TX_BAM_BUFFER_SIZE** or **#define TX_CONNECT_BUFFER_SIZE** which are located in *includes.h.*

If the message is a single packet message, the data is loaded into the transmit queue. If there is an overflow of this queue a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 3. To increase the size of the transmit queue change the value of the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.c*.

If the message with the passed index is not a globally broadcast message then the message will be sent to the destination address that is passed to the function.

The variable **u32Timeout** is the system time that must be reached before the message is transmitted. A value of 0 would cause the message to transmitted immediately.

9.5.7 U_16 fnBuildData(u8Index, *pData)

This function copies the data for the parameter group that corresponds to the passed index into the byte data array that is pointed to by the passed byte pointer.

9.5.8 U_8 fnFindTxMessage(u32PGN, u8CANPortIndex)

This function checks the list of PGNs in the cyclic transmit queue for a match to the PGN passed to the function. If a match is found an index number into the queue where the parameter group is located is passed back to the calling function. If a match is not found 0xFF is passed back to the calling function.

9.5.9 U_8 fnChange TxBroadcastRate(u8Index, u32BroadcastRate)

This function changes the broadcast rate of the parameter group that corresponds to the passed index to the value of the passed broadcast rate. The returned value is an indication of success or a reason for the failure to update the broadcast rate.

9.5.10 U_32 fnGetJ1939Timeout(u8Index)

This function returns the timeout time for the parameter group index that is passed to the function.

9.5.11 U_32 fnGetJ1939BroadcastRate(u8Index)

This function returns the broadcast rate for the parameter group index that is passed to the function.

9.5.12 U_8 *fnGetSequenceNumber(u8Index)

This function returns a pointer to the sequence number for the parameter group with the passed index.

9.5.13 U_8 fnGetPriority(u8Index)

This function returns the priority for the parameter group with the passed index.

9.5.14 U_8 fnChangePriority(u8Index, u8Priority)

This function changes the priority of the parameter group with the passed index to that of the passed priority. It will return an error code indicating whether the change was successful or the access level of the priority for the parameter group if the change was not successful. The returned value is an indication of success or a reason for the failure to update the priority.

9.5.15 U_8 fnGetJ1939PGNAccess(u8Index)

This function returns the access level for the parameter group with the passed index.

9.5.16 tTX_MESSAGE *fnGetTxParameterList()

This function returns a pointer to the list of messages that are cyclically broadcast or broadcast upon request.

9.5.17 BOOLEAN fnSetParameterGroupTx(u32PGN, u8CANPortIndex)

This function initiates the immediate transmission of the a message in the list of messages that are transmitted cyclically or by request. The passed PGN is compared to the messages in the list and if a match is found the message is broadcast. A TRUE or FALSE is returned to the calling function to indicate the success or failure finding a message with the passed PGN.

9.6 DiagnosticMessage.c

This module contains the functions that handle the J1939-73 diagnostic messaging.

9.6.1 Globals

BOOLEAN gflStartTx – This flag indicates the condition of the DM1 transmitter (**TRUE/FALSE**). If **TRUE** then the DM1 transmitter is turned on and the DM1 message is cyclically being transmitted on the CAN bus. If **FALSE** then the DM1 message is not being transmitted on the CAN bus.

tTX_DM_FAULTS *gpDM1TxFaultList – Pointer to the storage location of the DM1 fault list. This list resides outside the stack so the users of the list may easily manipulate it. The faults in the list are transmitted at a cyclic rate if the flag **gflStartTx** is **TRUE**.

U_8 gu8ConversionMethod – This is the conversion method that the stack uses to assemble/disassemble the fault codes for/from the CAN bus.

tRX_DM1_FAULTS gstDM1Faults – Storage buffer for any received DM1 faults. The number of simultaneous DM1 sessions that can be stored in this buffer can be adjusted by changing the **#define MAX_DM1_SESSIONS** which is located in *includes.h*. The number of DM1 faults that can be stored for each session can be adjusted by changing the **#define RX_DM1_FAULTS_BUFFER_SIZE** which is located in *includes.h*.

tRX_DM2_FAULTS gstDM2Faults – Storage buffer for any received DM2 faults. The number of simultaneous DM2 sessions that can be stored in this buffer can be adjusted by changing the **#define MAX_DM2_SESSIONS** which is located in *includes.h*. The number of DM2 faults that can be stored for each session can be adjusted by changing the **#define RX_DM2_FAULTS_BUFFER_SIZE** which is located in *includes.h*.

tACK_LIST gstDM3Ack – Storage location for any acknowledgment messages received for a DM3 request that was send on the CAN bus.

9.6.2 void fnDiagnosticMessageInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

9.6.3 void fnDiagnosticMessageRx()

This function receives and redirects any messages for the diagnostic channel.

9.6.4 void fnEvaluateMessage(tTP_CM_Rx *)

This function evaluates the received diagnostic message and stores the data accordingly.

If the received message is a DM1 and the DM1 sessions buffer is full then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 0. The number of simultaneous DM1 sessions that can be stored in this buffer can be adjusted by changing the **#define MAX_DM1_SESSIONS** which is located in *includes.h*.

If the received message is a DM2 and the DM2 sessions buffer is full then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 1. The number of simultaneous DM2

sessions that can be stored in this buffer can be adjusted by changing the **#define MAX_DM2_SESSIONS** which is located in *includes.h*.

If the number of DM1 faults that are received is larger than the buffer size that they are to be placed in, a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 2. The number of DM1 faults that can be stored for each session can be adjusted by changing the **#define RX_DM1_FAULTS_BUFFER_SIZE** which is located in *includes.h*.

If the number of DM2 faults that are received is larger than the buffer size that they are to be placed in, a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 3. The number of DM2 faults that can be stored for each session can be adjusted by changing the **#define RX_DM2_FAULTS_BUFFER_SIZE** which is located in *includes.h*.

9.6.5 void fnSetupDMTxMessage(u8DMType, tTX_DM_FAULTS *)

This function causes the DM pointer to point to the DM buffer that contains the faults list associated with the passed DM type variable (DM1 or DM2). **The user of the stack must setup the fault location using the structure tTX_DM_FAULTS and then call this function and pass a pointer to that location.** This function need only be called once during initialization of the software unless the memory locations where the fault lists reside change.

After a DM2 message location set is passed to this function, any request messages received for a DM2 PGN will automatically be responded to with the data at the location of the passed DM pointers.

9.6.6 void fnStartDM1Tx(u8CANPortIndex)

This function starts the cyclic transmission of the DM1 faults list on the CAN bus. If **fnSetupDM1TxMessage** has not been called before calling the function **fnStartDM1Tx()** then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 4.

9.6.7 void fnStopDM1Tx(u8CANPortIndex)

This function stops the cyclic transmission of the DM1 faults list on the CAN bus.

9.6.8 void fnDiagnosticMessageTx()

This function checks to see if the DM1 transmit flag is active and if the cyclic timer has lapsed. It will initiate a transmission of the DM1 faults list if both conditions are satisfied.

9.6.9 void fnCyclicDM1TxOverride(u8CANPortIndex)

This function overrides the current timeout time for broadcasting the DM1 message and causes the message to be transmitted right away instead of at the end of the timeout cycle. This function is used if the contents of the DM1 message has changed and an immediate broadcast in needed.

9.6.10 void fnTxDM2FaultList(tTX_DM_FAULTS *)

This function sends the passed DM2 faults list to the format for transmission function.

Note: This function has be left in the code for backwards compatibility. It is no longer needed to broadcast a DM2 message as that functionality can now be achieved by passing the DM2 list to the fnSetupDmTxMessage() function.

9.6.11 void fnLoadDMTxBuffer(u32Id, tTX_DM_FAULTS *)

This function formats the passed DM faults list for transmission on the CAN bus.

If the DM fault list only requires a single packet message to broadcast its contents then the formatted list is loaded into the transmit queue. If the queue overflows a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 5. The transmit queue can be adjusted by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.

If the DM fault list requires the transport protocol to broadcast its contents then the formatted list is loaded into a mutipacket DM sessions buffer via the function **fnGetTPTxDataBuffer**. If there are no free sessions in the BAM transmit queue then a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 6. The number of simultaneous DM sessions that can be stored in a buffer can be adjusted by changing the **#define MAX_DM1_SESSIONS** for the DM1 buffer and the **#define MAX_DM2_SESSIONS** for the DM2 buffer. These #defines are located in *includes.h*.

9.6.12 void fnConvertSPN(u32TempSPN)

This function converts the passed SPN to a value that will be used for transmission on the CAN bus. This conversion is based on the conversion method that is chosen by the user. The **#define CONVERSION_METHOD** which is located in *includes.h*. can be changed to select a different method. The J1939 standard currently defines 4 methods for conversion. For transmitting the DM message, if method 4 is chosen then the conversion method bit within the fault message will be set to 0. If any of the other methods are chosen then the conversion method bit will be set to 1. Methods 3 and 4 are identical as far as converting the data. The only difference is the setting of the conversion method bit.

9.6.13 void fnTxDM2(u8CANPortIndex)

This function is used by the stack to broadcast the DM2 message list on the passed CAN port.

9.6.14 void fnSetConversionMethod(u8ConversionMethod)

This function allows the user to set the fault code conversion method during runtime.

9.6.15 tRX_DM1_FAULTS *fnGetDM1FaultList()

This function returns a pointer to the received DM1 faults list. If there are no active faults then the function returns a **NULL**.

9.6.16 tRX_DM2_FAULTS *fnGetDM2FaultList()

This function returns a pointer to the received DM2 faults list. If there are no previously active faults then the function returns a **NULL**.

9.6.17 void fnSendDM3(u8DestinationAddress, u8CANPortIndex)

This function initiates the transmission of a DM3 message to the passed destination address. If the transmit queue overflows a warning flag will be set by calling the function **fnSetWarningFlag** and passing it the module tag and the warning flag number 6. The transmit queue can be adjusted by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.

9.6.18 tACK_LIST *fnGetDM3AckType(u8CANPortIndex)

This function returns a pointer to any received acknowledgment message concerning a DM3 request. If there are no acknowledgment messages concerning a DM3 request then a **NULL** is returned.

Note: If the destination address of the requested DM3 is the global address then there will not be a acknowledgment response from any device on the bus.

10. NMEA Modules

10.1 FastPacket.c

This module contains the functions that handle the transmitting and receiving of the NMEA 2000 fastpacket message.

10.1.1 Globals

tFASTPACKET_RX gstFastPacket_Rx – This structure contains the received fastpacket messages. The number of receive fastpacket sessions that can be stored in this buffer can be adjusted by changing the **#define MAX_RX_FASTPACKET_SESSIONS** which is located in *includes.h*.

tFASTPACKET_TX gstFastPacket_Tx – This structure contains the information for the fastpacket transmit messages that have been loaded into the cyclic or broadcast on request queue.

10.1.2 void fnFastPacketInit()

This function does the initialization of any variables that are global to this module.

10.1.3 void fnProcessFastPacket(*pRawCANData)

This function is the main engine for processing received fastpacket messages. It is called at a periodic rate by an application function. Messages are copied from the raw can data buffer into a waiting buffer. Once all of the packets for a particular parameter group have been received a pointer is passed to the receive data buffer so the information can be accessed by the channels that need it.

10.1.4 BOOLEAN fnLoadNMEATxMessageIntoList(*pTxMessage)

This function loads a message passed to it by the calling function into the NMEA transmit list. The calling function passes a pointer to a memory location where the data to be transmitted is stored. Changing the contents of that memory location will automatically change the contents of the message that is transmitted. A flag is returned to the calling function that indicates whether the message was loaded successfully.

For NMEA messages, all parameters for a given message must be loaded into the stack, even if the parameters are not used. This requirement is because some NMEA fields can be variable in length. Thus in order to build the NMEA message with the correct number of bytes, all parameters must be loaded.

10.1.5 BOOLEAN fnRemoveNMEATXMessageFromList(u32PGN, *pData)

This function removes a message with the passed PGN and data address from the transmit list. A flag is returned to the calling function that indicates whether the message was successfully removed.

10.1.6 U_8 fnGetNMEAField(u8Index, u8Field)

This function returns an index value that correlates with the passed field for the parameter group with the passed index.

10.1.7 U16 fnGetNMEAInstance(u8Index, u8Field, *pData)

This function returns an index into the instances of a parameter groups that contains the pointed to data at the passed field index.

10.1.8 U_8 fnBuildNMEAData(u8Index, u16Instance, *pData)

This function fills an external array (passed pointer *pData) with the data for the parameter group with the passed index and instance. The function will return the number of bytes that were copied into the external array.

10.1.9 void fnCheckNMEATxData()

This function checks the times for each of the messages pending transmission and activates any that have timed out.

10.1.10 BOOLEAN fnProcessNMEATxMessage(u8Index, u16Instance, u8DestinationAddress, u32IntervalOffset)

This function builds the message that needs to be transmitted that contains the passed index and timeout info. If the message is not a fastpacket message it will be loaded into the transmit queue. If the message is a fastpacket message fnSetUpFastPacketTx() will be called to handle the transmission of the message. A flag is returned if the message was successfully loaded into the Tx queue.

10.1.11 BOOLEAN fnSetUpFastPacketTx(*pstTxParametergroup, *pData, u8NumberOfBytes, u32Intervaloffset, *pSequenceNumber)

This function loads a fastpacket message with the passed information into the Tx queue. A flag is return if the message was successfully loaded into the Tx queue.

10.1.12 U_8 fnFindNMEATxMessage(u32PGN, u8CANPortIndex)

This function returns an index into the transmit message list to the parameter group with the passed PGN. 0xFF is returned if the parameter group could not be found.

10.1.13 tFASTPACKET_TX *fnGetNMEATxParameterGroupList()

This function returns a pointer to the cyclically transmitted NMEA message list. This allows functions outside of the FastPacket.c module to gain access to the transmit parameters.

10.1.14 tFASTPACKET_PARAMETER_GROUPS *fnGetNMEATxParameter(U_16 u16Index)

This function returns a pointer a parameter group entry at the passed index.

10.1.15 U_32 fnGetNMEATimeout(u8Index, u16Instance)

This function returns the timeout time for the parameter group with the passed index and instance.

10.1.16 U_32 fnGetNMEAIntervalOffset(U_8 u8Index, U_16 u16Instance)

This function returns the IntervalOffset time for the passed parameter group index and instance.

10.1.17 U_8 fnChangeNMEAPriority(u8Index, u8Priority)

This function changes the priority for the parameter group with the passed index to that of the passed priority. An error code is returned indicating whether the change was successful or the access level of the priority for the parameter group.

10.1.18 U_8 fnChangeNMEAInterval(u8Index, u16Instance, u32Interval)

This function changes the interval for the parameter group with the passed index to that of the passed interval. An error code is returned indicating whether the change was successful or the access level of the interval for the parameter group.

10.1.19 U_8 fnChangeNMEAIntervalOffset(u8Index, u16Instance, u32IntervalOffset)

This function changes the Interval offset for the parameter group with the passed index to that of the passed interval offset. An error code is returned indicating whether the change was successful or the access level of the interval offset for the parameter group.

10.1.20 U_8 fnGetNMEAPGNAccess(u8Index)

This function returns the access level for the parameter group with the passed index.

10.1.21 U_8 fnGetFieldAccess(U_8 u8NMEAIndex, U_8 u8Field)

This function returns the field access level for the parameter index that is passed to the function.

10.1.22 BOOLEAN fnSetNMEAParameterGroupTx(U_32 u32PGN, U_8 u8CANPortIndex, U_8 u8DestinationAddress, U_8 u8Instance)

This function allows a channel to immediately broadcast a message that is in the Tx parameter list.

10.2 RequestCommandAck.c

This module contains the functions required to transmit and receive the NEMA complex request, command and acknowledgment messages.

10.2.1 Globals

tNMEA_ACK_MESSAGE gstNMEAAckMessage – This structure contains the received acknowledment messages. The number of receive acknowledgment messages that can be stored in this buffer can be adjusted by changing the **#define RX_ACK_NMEA_BUFFER_SIZE** which is located in *includes.h*.

10.2.2 void fnRequestCommandAckInit()

This function does the initialization of any variables that are global to this module. It also loads the messages needed by this channel into the receive messages list.

10.2.3 void fnCheckRequestCommandAck()

This function is the main engine for processing the NMEA complex receive/command/acknowledgment messages. It is called at a periodic rate by an application function.

10.2.4 void fnProcessRequestMessage(*pMessage)

This function processes the Rx of the complex request message.

10.2.5 void fnProcessCommandMessage(*pMessage)

This function processes the receipt of the NMEA command message. It will take any actions that are commanded and transmit an acknowledgment message indicting the success or failure of each command.

10.2.6 void fnProcessAckMessage(pMessage)

This function processes the receipt of the acknowledgment message. The message is copied into the structure gstNMEAAckMessage. This structure can be accessed by calling function fnGetNMEAAckMessage().

10.2.7 tNMEA_ACK_MESSAGE *fnGetNMEAAckMessage(u32PGN)

This function returns a pointer to the structure location that contains information regarding the passed PGN. If there are no acknowledgement messages it the structure that match the passed PGN then a NULL is returned.

10.2.8 void fnNMEATxAckMessage(*pMessage, *pErrorCodes)

This function processes the transmission of the NMEA acknowledgment message. Information about the message is passed to the function via the *pMessage structure pointer and *pErrorCodes pointer. The message is then built and place in the Tx queue.

10.2.9 static void fnReadFields(*pMessage)

This function receives and interprets the Read Fields request message.

10.2.10 static void fnWriteFields(*pMessage)

This function receives and interprets the Write Fields request message.

10.3 TransmitPGNs.c

This module contains the functions required to transmit the NEMA Transmit PGNs message.

10.3.1 Globals

u8TransmitPGNsSequence – This is a sequence number that the Transmit PGNs message uses when sending its PGNs using the fastpacket protocol. The sequence number is used to differentiate messages with the same ID from one another.

10.3.2 void fnTransmitPGNsInit()

This function does the initialization of any variables that are global to this module.

10.3.3 void fnProcessNMEATransmitPGNs(*pMessage, *pErrorCodes, *pData)

This function processes the receipt of a NMEA request message for the NMEA transmit PGNs message. If the message cannot be sent than an error code will be loaded into the passed error buffer and transmitted to the device who sent the request.

10.3.4 void fnProcessTransmitPGNs(u8SourceAddress, u8CANPortIndex)

This function processes the receipt of a J1939 request message for the NMEA transmit PGNs message. Since the J1939 request message does not have a data field where either the Transmit Tx PGNs or Transmit Rx PGNs can be specified,

receiving the Transmit PGNs message using this format will cause the Transmit Tx PGNs and Transmit Rx PGNs messages to be broadcast.

10.3.5 void fnTransmitPGNMessage(*pData, *pErrorCodes, u16NumberOfBytes, u8SourceAddress, u8CANPortIndex, u32TransmitTime)

This function sends the NMEA Transmit PGNs message to the Tx queue. If the request message was received using the J1939 transport protocol then the transmission of the Transmit PGNs message will also use the J1939 transport protocol. If the Transmit PGNs message is less than 8 bytes then the fastpacket protocol will be used regardless of how the request for the Transmit PGNs message was received.

10.3.6 void fnTxTransmitPGNs(*pData, u8CANPortIndex)

This function builds the data for the Transmit Tx PGNs message. This function simply places the data that will go into the body of the Transmit Tx PGNs message into a byte array. A pointer to the byte array is passed to this function.

10.3.7 void fnTxReceivePGNs(*pData, u8CANPortIndex)

This function builds the data for the Transmit Rx PGNs message. This function simply places the data that will go into the body of the Transmit Rx PGNs message into a byte array. A pointer to the byte array is passed to this function.

11. ISO 15765 Modules

11.1 ISO_15765_TP_Fixed_Mixed.c

11.1.1 Globals

tISO_15765_RX_FIXED_MIXED_MESSAGE gastISO15765Rx – This structure contains the received Normal Fixed and Mixed ISO15765 messages. The number of received messages that can be stored in this buffer can be adjusted by changing the **#define ISO15765_RX_FIXED_MIXED_MESSAGES** which is located in *includes.h*.

tISO_15765_TX_FIXED_MIXED_MESSAGE_INTERNAL gastISO15765Tx– This structure contains the transmitted Normal Fixed and Mixed ISO15765 messages. Once a message is transmitted this buffer is cleared for the next message. The number of simultaneous transmitted messages and thus the size of the buffer can be adjusted by changing the **#define ISO15765_TX_FIXED_MIXED_MESSAGES** which is located in *includes.h*.

11.1.2 void fnISO_15765FixedMixedInit ()

This function does the initialization of any variables that are global to this module. It also loads the standard Normal Fixed and Mixed, physical and functional CAN IDs into the stack.

11.1.3 void fnISO_15765FixedMixedEngine()

This function is the main engine for the module. It is called cyclically from fnStackMainLoop(). It routes all received messages and calls the functions that handle the cyclic transmission of packets.

11.1.4 static void fnSaveNormalFixed(*pMessage)

This function processes any received Normal Fixed messages. The passed structure pointer pMessage contains the raw CAN data for the received message. This message also initiates the flow control message when necessary. This function is used by other functions within the stack and cannot be called by the application.

11.1.5 static void fnSaveMixed (*pMessage)

This function processes any received Mixed messages. The passed structure pointer pMessage contains the raw CAN data for the received message. This message also initiates the flow control message when necessary. This function is used by other functions within the stack and cannot be called by the application.

11.1.6 static tISO_15765_RX_FIXED_MIXED_MESSAGE *fnGetOpenBuffer(*pMessage)

This function returns a pointer to an open receive buffer to the calling function. This function is used by other functions within the stack and cannot be called by the application.

11.1.7 static BOOLEAN fnTxFlowControlStatus(*pMessage, u8Status, u8AddressingMode)

This function causes the transmission of the flow control message. The passed pointer pMessage contains the data from the received message that fulfilled the requirements to cause a flow control message to transmit. The variable u8Status contains the status field information for the flow control message. The variable u8AddrssingMode contains information concerning what type of message is to be broadcast (Normal Fixed physical, Normal Fixed functional, Mixed physical or Mixed functional). This function is used by other functions within the stack and cannot be called by the application.

11.1.8 void tISO_15765_RX_FIXED_MIXED_MESSAGE *fnGetISO15765FixedMixedMessages()

This function is called by the application to receive a pointer to a structure containing a received message. The function fnReleaseISO15765FixedMixedMessage () must be called after calling after the data within the returned structure pointer is consumed. This will release the buffer to receive a new message. Calling the function fnGetISO15765FixedMixedMessages() twice in a row without calling fnReleaseISO15765FixedMixedMessage() will cause a pointer to the same received message to be returned twice in a row.

11.1.9 void fnReleaseISO15765FixedMixedMessage(*pBuffer)

This function release the passed receive buffer to be used for another message. The pointer that is passed by this function is the one that is returned by the calling function fnGetISO15765FixedMixedMessages().

11.1.10 static void fnCheckChannelTimeout()

This function checks to see if the stack is ready to transmit another consecutive data packet and initiates the transmission if necessary. This function is used by other functions within the stack and cannot be called by the application.

11.1.11 BOOLEAN fnLoadFixedMixedMessageToTx(*pstData)

This function is called by the application to load a message that needs to be transmitted. The passed structure pointer contains the necessary information for building the message. Once a message is loaded into the stack, the stack will handle all of the timing and packetizing of the data. The transmit buffer used my a message will automatically be released for reuse once the message is finished broadcasting. A Boolean value of TRUE/FALSE is returned to indicate the success/failure of loading the message into the stack.

11.1.12 static tISO_15765_TX_FIXED_MIXED_MESSAGE_INTERNAL *fnGetFreeISO15765TxBuffer()

This function returns a pointer to a free Tx buffer location. It is used by other functions within the stack and cannot be called by the application.

11.1.13 __weak void fnISO15765FixedMixedErrorCallback(*pstError)

This function is a callback that should be overridden by the application software. It fills the passed pointer structure with data concerning the Normal Fixed or Mixed error that has occurred.

11.2 ISO_15765_TP_Normal_Extended.c

11.2.1 Globals

U_16 gu16RxDataCount – This is the number of different received CAN IDs that have been loaded into the stack for Normal and Extended ISO15765 message.

tISO_15765_NORMAL_EXTENDED_RX_MESSAGE gstRxMessage – This is the structure that contains the data for the application who calls the fnGetISO15765NormalExtendedMessages() function.

tISO_15765_RX_NORMAL_EXTENDED_RAW_DATA gastRxRawData– This structure contains the received Normal and Extended ISO15765 raw data bytes. The number of received messages that can be stored in this buffer can be adjusted by changing the **#define ISO15765_RX_NORMAL_EXTENDED_RAW_MESSAGE** which is located in *includes.h*.

tISO_15765_RX_NORMAL_EXTENDED_DATA gastRxDataInfo – This structure contains the received Normal and Extended ISO15765 message information. The number of different received messages that can be stored in this buffer

can be adjusted by changing the **#define ISO15765_RX_NORMAL_EXTENDED_INFO_MESSAGES** which is located in *includes.h*.

tISO_15765_TX_NORMAL_EXTENDED_MESSAGE_INTERNAL gastTxDataInfo – This structure contains the transmitted Normal and Extended ISO15765 information and message data. Once a message is transmitted this buffer is cleared for the next message. The number of simultaneous transmitted messages and thus the size of the buffer can be adjusted by changing the **#define ISO15765_TX_NORMAL_EXTENDED_INFO_MESSAGES** which is located in *includes.h*.

11.2.2 void fnISO_15765NormalExtendedInit ()

This function does the initialization of any variables that are global to this module.

11.2.3 void fnISO_15765NormalExtendedEngine ()

This function is the main engine for the module. It is called cyclically from fnStackMainLoop(). It routes all received messages and calls the functions that handle the cyclic transmission of packets.

11.2.4 static void fnSaveNormal(*pMessage, *pstRxDataInfo)

This function processes any received Normal messages. The passed structure pointer pMessage contains the raw CAN data for the received message. The passed structure pointer pstRxDataInfo contains a pointer to the location where the received message is to be saved. This message also initiates the flow control message when necessary. This function is used by other functions within the stack and cannot be called by the application.

11.2.5 static void fnSaveExtended(*pMessage, *pstRxDataInfo)

This function processes any received Mixed messages. The passed structure pointer pMessage contains the raw CAN data for the received message. The passed structure pointer pstRxDataInfo contains a pointer to the location where the received message is to be saved. This message also initiates the flow control message when necessary. This function is used by other functions within the stack and cannot be called by the application.

11.2.6 static void fnNormalFlowControl(*pMessage, *pstTxDataInfo)

This function handles the processing of a received Normal flow control message. pMessage is a pointer to the received message. pstTxDataInfo is a pointer to the buffer containing the transmit message for which the received flow control message pertains.

11.2.7 static void fnExtendedFlowControl(*pMessage, *pstTxDataInfo)

This function handles the processing of a received Extended flow control message. pMessage is a pointer to the received message. pstTxDataInfo is a pointer to the buffer containing the transmit message for which the received flow control message pertains.

11.2.8 static BOOLEAN fnTxFlowControlStatus(*pMessage, u8Status, u8AddressingMode)

This function causes the transmission of the flow control message. The passed pointer pMessage contains the data from the received message that fulfilled the requirements to cause a flow control message to transmit. The variable u8Status contains the status field information for the flow control message. The variable u8AddrssingMode contains information concerning what type of message is to be broadcast (Normal physical, Normal functional, Extended physical or Extended functional). This function is used by other functions within the stack and cannot be called by the application.

11.2.9 static void fnCheckChannelTimeout()

This function checks to see if the stack is ready to transmit another consecutive data packet and initiates the transmission if necessary. This function is used by other functions within the stack and cannot be called by the application.

11.2.10 BOOLEAN fnISO_15765NormalExtendedLoadRxMessages(*pstData)

This function is called by the application code to load the passed listed of received messages into the stack. If this function is called again by the same channel that called it before, the previous list of received messages will be erased and the new list will take its place. A Boolean value of TRUE/FALSE is returned depending on the success/failure of the function to load the list of passed received messages into the stack.

11.2.11 BOOLEAN fnLoadNormalExtendedMessageToTx(*pstData)

This function is called by the application to load a message that needs to be transmitted. The passed structure pointer contains the necessary information for building the message. Once a message is loaded into the stack, the stack will handle all of the timing and packetizing of the data. The transmit buffer used my a message will automatically be released for reuse once the message is finished broadcasting. A Boolean value of TRUE/FALSE is returned to indicate the success/failure of loading the message into the stack.

11.2.12 void tISO_15765_NORMAL_EXTENDED_RX_MESSAGE *fnGetISO15765NormalExtendedMessages ()

This function is called by the application to receive a pointer to a structure containing a received message. The function fnReleaseISO15765NormalExtendedMessage() must be called after calling after the data within the returned structure pointer is consumed. This will release the buffer to receive a new message. Calling the function fnGetISO15765NormalExtendedMessages() twice in a row without calling fnReleaseISO15765NormalExtendedMessage() will cause a pointer to the same received message to be returned twice in a row.

11.2.13 void fnReleaseISO15765NormalExtendedMessage (*pBuffer)

This function release the passed receive buffer to be used for another message. The pointer that is passed by this function is the one that is returned by the calling function fnGetISO15765NormalExtendedMessages().

11.2.14 __weak void fnISO15765NormalExtendedErrorCallback(*pstError)

This function is a callback that should be overridden by the application software. It fills the passed pointer structure with data concerning the Normal or Extended error that has occurred.

12. ISO 14229 Modules

12.1 ISO 14229 Setup

Most of the ISO 14229 setup is done in the ISO_14229_Main.h module. This module contains #defines for all of the services defined in the ISO 14299-1 and -2 standards documents. There are also #defines for any sub-functions that are associated with each service.

12.1.1 Services

The list of the defined services is as follows:

```
#define DIAGNOSTIC_SESSION_CONTROL      0x10
#define ECU_RESET                       0x11
```

```
#define SECURITY_ACCESS                         0x27
#define COMMUNICATION_CONTROL                   0x28
#define TESTER_PRESENT                          0x3E
#define ACCESS_TIMING_PARAMETER                 0x83
#define SECURED_DATA_TRANSMISSION               0x84
#define CONTROL_DTC_SETTING                     0x85
#define RESPONSE_ON_EVENT                       0x86
#define LINK_CONTROL                            0x87
#define READ_DATA_BY_IDENTIFIER                 0x22
#define READ_MEMORY_BY_ADDRESS                  0x23
#define READ_SCALING_DATA_BY_IDENTIFIER         0x24
#define READ_DATA_BY_PERIODIC_IDENTIFIER        0x2A
#define DYNAMICALLY_DEFINE_DATA_IDENTIFIER      0x2C
#define WRITE_DATA_BY_IDENTIFIER                0x2E
#define WRITE_MEMORY_BY_ADDRESS                 0x3D
#define CLEAR_DIAGNOSTIC_INFORMATION            0x14
#define READ_DTC_INFORMATION                    0x19
#define INPUT_OUTPUT_CONTROL_BY_IDENTIFIER      0x2F
#define ROUTINE_CONTROL                         0x31
#define REQUEST_DOWNLOAD                        0x34
#define REQUEST_UPLOAD                          0x35
#define TRANSFER_DATA                           0x36
#define REQUEST_TRANSFER_EXIT                   0x37
#define REQUEST_FILE_TRANSFER                   0x38
```

If a service listed above is commented out, the code concerning that service will be removed from the compiled program. If the client requests a service that has been removed the stack will respond to the request with a negative response message if the request was made using a physical ISO 15765 transport protocol.

12.1.2 Sub-functions

Most services contain a number of sub-functions that aid in defining different processes that can be requested by the client. These sub-functions are #defined and as such can be removed from the operation of the stack by commenting out the #define associated with a given sub-function. If the client requests a sub-function that has been removed the stack will respond to that request with a negative response message.

12.1.3 __weak Defined Functions

Most of the service requests will contain a __weak function definition. These functions reside on the stack side of the code but can and in most cases should be strongly redefined on the application code side. The purpose of the redefinition of these functions is so the application code an properly respond to the client request. Most service request will contain response information or functionality that is manufacturer specific and thus outside the scope of the stack.

12.1.4 Diagnostic Sessions

The server is setup to run in different diagnostic sessions. The manufacturer may wish that a limited set of services be available for a given session. For each service defined in ISO_14229_Main.h there is a #define that will allow the manufacturer to define which session a given service can be accessed during. The #define (XXX_DURING_SESSION) is a bit mapped value where the locations correspond to a session. The XXX represents the particular service being referenced. If the bit is set the service is active during that session.

The following are the possible bit mapped values:
0001b = DEFAULT_SESSION
0010b = PROGRAMMING_SESSION
0100b = EXTENDED_DIAGNOSTIC_SESSION
1000b = SAFETY_SYSTEM_DIAGNOSTIC_SESSION

The server will always have at least one session active. The default session is active at power-up and must always be a supported option. The other sessions are optionally supported. By default the stack is setup to allow all services in every session. See the section on fnProcessDiagnosticSessionControl for details on changing the current diagnostic session control message.

12.1.5 Security

Most services have a security option with must be satisfied before the server will accept a service request from the client. This security option can be disabled by the manufacturer by commenting out the security #define (XXX_SECURITY) associated with each service at compile time. The XXX represents the particular service being referenced. By default the stack is setup to require security for every service. See the section on fnProcessSecurityAccess for details on receiving the message that unlocks the security access for a given set of services.

12.2 ISO_14229_Main.c

12.2.1 Globals

tISO14229_SESSION_DATA gastSession – This structure contains the session information for the connection between the server and the client. The information such as session state, client source address, CAN port, ISO 15765 transport type, CAN ID,… are saved within this structure.

tISO14229_NORMAL_EXTENDED_PAIR gstPair– This structure contains information about the ISO 15765's normal and extended transport protocols. Some of the data that is stored in this structure are the transmit and received ID pair, CAN port, ISO 15765 transport type,…

12.2.2 void fnISO_14229Init()

This function does the initialization of any variables that are global to this module.

12.2.3 void fnISO_14229SetNormalExtendedMessages(*pstData, *pstPair)

This function initializes the normal or extended transport protocol transmit/receive pair and ISO15765 stack module. The passed pointer pstData contains initialization data for the normal/extended ISO 15765 transport protocol. The pointer pstPair contains transmit and receive IDs pairs that are used by the normal/extended ISO 15765 transport protocol.

The passed data contains:
pstData – pointer to the data used to setup the ISO 15765 normal or extended transport protocol functionality.
pstPair – pointer to the transmit and receive pairs used by the normal or extended transport protocols.

12.2.4 void fnISO_14229Engine ()

This function is the main engine for processing ISO 14229 transport protocol messages. It is called at a periodic rate by an application function. It handles the retrieval of received messages from the ISO 15765 module.

12.2.5 void fnProcessRequestMessage(*pstSession)

This function handles the routing of each received message to the proper processing function. The passed pointer contains the message and session data.

The passed data contains:
pstSession – pointer to the received message and the current session.

12.2.6 static void fnCheckTimeouts()

This function handles all communications timing events within the module..

12.2.7 tECU_STATE_DATA *fnGetECUData()

This function return a pointer to the ECU session control and security state data.

12.2.8 tISO14229_NORMAL_EXTENDED_PAIR_ID *fnGetMatchingPair(u32Id)

This function return a pointer to a pair of transmit and receive IDs that match the passed receive ID.

12.2.9 __weak BOOLEAN fnChangeAcknowledgmentCallback(u8Service, u8Subfunction)

This function is a week function that needs to be strongly implemented on the application code side. It is meant to return an indication that the application code can and is ready to support the requested service and sub-function indicated in the passed data. Reasons for not being ready to support the service being requested might be that the engine is running or the vehicle is not in park.

The returned data contains:
A Boolean indicating that “TRUE” the service can be run or “FALSE” the service cannot run.

12.3 ISO_14229_TxResponse.c

12.3.1 Globals

None

12.3.2 void fnSendError(*pstData, u8Error)

This function builds the error message based on the passed u8Error code. It then passes the message to the ISO 15765 module for transmission.

12.3.3 void fnSetPositiveResponse(*pstRxData, *pau8TxData, u16TxCount)

This function builds the positive response message and passes it to the ISO 15765 module for transmission.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a buffer containing the data to be transmitted to the client.
u16TxCount – Number of significant bytes in the pau8TxData buffer.

12.4 ISO_14229_DiagAndCommManagement.c

12.4.1 Globals

None

12.4.2 void fnProcessDiagnosticSessionControl(*pstRxData)

This function handles the reception and processing of the Diagnostic Session Control message. The Diagnostic Session Control message allows the client to change the current session state of the server.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.3 void fnProcessTesterPresent(*pstRxData)

This function handles the reception of the Tester Present message. The tester present message allows a session to remain open for an extended period of time without the client sending other service requests to the server.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.4 void fnProcessSecurityAccess(*pstRxData)

This function handles the reception of the Security Access message. The Security Access message allows a client to unlock a manufacture specific list of services that can be run on the server. Multiple security access levels are available to unlock different functionalities.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.4.1 __weak U_8 fnSecurityAccessCallback(*pstRxData, *pau8TxData, *pu8TxCount)

This function is a weak function that needs to be strongly implemented on the application code side or finished on the stack side. It is meant to provide the client with a security seed and authenticate the security key passed to it by the client. See ISO 14229-1 Section 9.4.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where the security key will be placed and/or any manufacture specific positive response information.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.5 void fnProcessECUReset(*pstRxData)

This function handles the reception of the ECU Reset message. The ECU Reset message allows a client force the server to reset. There are 3 different reset states that can be called:

Key On/Off reset
Soft reset
Rapid power shutdown

The passed data contains:

pstRxData – Pointer to a structure containing the received message and session data.

12.4.5.1 __weak U_8 fnECUResetCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the ECU reset message. The application code should reset the ECU as indicated by the requested sub-function. See ISO 14229-1 Section 9.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.

pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.6 void fnProcessCommunicationControl(*pstRxData)

This function handles the reception of the Communication Control message. The Communication Control message allows the client to disable/enable the transmission and reception of message on the server.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.6.1 __weak U_8 fnECUResetCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Communication Control message. The application code should evaluate the requesting sub-function and enable/disable communications where applicable. See ISO 14229-1 Section 9.5.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.7 void fnProcessAccessTimingParameter(*pstRxData)

This function handles the reception of the Access Timing Parameter message. The Access Timing Parameter message allows the client to request the communications timeouts used be the stack.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.7.1 __weak U_8 fnAccessTimingParameterCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Access Timing Parameter message. The application code should evaluate the requesting sub-function and respond with its timing parameters or set its timing parameters accordingly. See ISO 14229-1 Section 9.7.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.8 void fnProcessControlDTCSetting(*pstRxData)

This function handles the reception of the Control DTC Settings message. The Control DTC Settings message allows the client to turn ON/OFF the DTC functionality of the server.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.4.8.1 __weak U_8 fnControlDTCSettingCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Control DTC Settings message. The application code should evaluate the requesting sub-function and DTC setting control option record, if one is provided and respond with the appropriate actions. See ISO 14229-1 Section 9.9.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.9 void fnProcessLinkControl(*pstRxData)

This function handles the reception of the Link Control message. The Link Control message allows the client to change baud rate used by the communications bus. Under J1939 there are 5 options:

125k baud
250k baud
500k baud
1M baud
User defined

The passed data contains:

pstRxData – Pointer to a structure containing the received message and session data.

12.4.9.1 __weak U_8 fnLinkControlCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Link Control message. The application code should evaluate the requested sub-function and change the baud rate as required. See ISO 14229-1 Section 9.11.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.4.10 void fnProcessSecuredDataTransmission(*pstRxData)

This function handles the reception of the Secured Data Transmission message. The Secured Data Transmission message allows the client encrypt/decrypt any data passed on the bus. The passed data contains:

pstRxData – Pointer to a structure containing the received message and session data.

12.4.10.1 __weak U_8 fnSecuredDataTransmissionCallback(*pstRxData, u8Direction, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Secured Data Transmission message. The application code should encrypt/decrypt the message based on the passed u8Direction variable. See ISO 14229-1 Section 9.8.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
u8Direction – Variable indicating whether the data is to be encrypted and placed in the transmission buffer or decrypted and replace the received encrypted data with the decrypted data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

12.5 ISO_14229_UploadDownload.c

12.5.1 Globals

static tUPLOAD_DOWNLOAD gstUploadDownload – Structure containing the data used for a given upload or download session.

12.5.2 void fnProcessRequestDownload(*pstRxData)

This function handles the reception and processing of the Request Download message. The Request Download message allows the client to inform to the server that it wishes to perform transfer of data from the client to the server. It also specifies where the data is to be saved and how much data is to be transferred.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.5.2.1 __weak U_8 fnRequestDownloadCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Request Download message. The application code should evaluate the request, check for manufacture specific errors and define a u16MaxNumberOfBlockLength value. See ISO 14229-1 Section 14.2.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.5.3 void fnProcessRequestUpload(*pstRxData)

This function handles the reception and processing of the Request Upload message. The Request Upload message allows the client to inform to the server that it wishes to perform transfer of data from the server to the client. It also specifies where the data is to be read from and how much data is to be transferred.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.5.3.1 __weak U_8 fnRequestUploadCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Request Upload message. The application code should evaluate the request and

generate a manufacture specific error if needed or define a u16MaxNumberOfBlockLength value if the received request is valid. See ISO 14229-1 Section 14.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.5.4 void fnProcessRequestFileTransfer (*pstRxData)

This function handles the reception and processing of the Request File Transfer message. The Request File Transfer message is used when the data to be transferred is in a file system format.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.5.4.1 __weak U_8 fnRequestFileTransferCallback (*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Request File Transfer message. The application code should evaluate the request and generate a manufacture specific error if needed or if the request is valid, define a u16MaxNumberOfBlockLength value and fill out the tFILE_TRANSFER data structure with the data that needs to be passed back to the client. See ISO 14229-1 Section 14.6

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.5.5 void fnProcessTransferData(*pstRxData)

This function handles the reception and processing of the Transfer Data message. The Transfer Data message is the vehicle for transmitting the actual data specified in the Download/Upload/File Transfer request messages, between the client and the server.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.5.5.1 __weak U_8 fnTransferDataCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Transfer Data message. The application code should evaluate the request and generate a manufacture specific error if needed or respond by reading or writing the data as required. See ISO 14229-1 Section 14.4.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.5.6 void fnProcessRequestTransferExit (*pstRxData)

This function handles the reception and processing of the Request Transfer Exit message. The Request Transfer Exit message is used by the client to end or terminate the current upload/download/file transfer session.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.5.6.1 __weak U_8 fnRequestTransferExitCallback (*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Request Transfer Exit message. The application code should return any transfer related data to the client and end the session. See ISO 14229-1 Section 14.5. The passed data contains:

pstRxData – Pointer to a structure containing the received message and session data.

12.6 ISO_14229_GenericReadWrite.c

12.6.1 Globals

static tPERIODIC_IDENTIFIER gstPeriodicData – Structure containing the data necessary to cyclically broadcast a requested parameter or data from a given memory address.

12.6.2 void fnProcessReadDataByIdentifier(*pstRxData)

This function handles the reception and processing of the Read Data By Identifier message. The Read Data By Identifier message allows the client specify a given data identifier who's data it wishes to receive.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.2.1 __weak U_8 fnReadDataByIdentifierCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Read Data By Identifier message. The application code should evaluate the request and respond with the requested data or an error. See ISO 14229-1 Section 10.2.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.3 void fnProcessReadMemoryByAddress(*pstRxData)

This function handles the reception and processing of the Read Memory By Address message. The Read Memory By Address message allows the client specify a memory address and number of bytes that it wishes to read.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.3.1 __weak U_8 fnReadMemoryByAddressCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Read Memory By Address message. The application code should evaluate the request and respond with the requested data or an error. See ISO 14229-1 Section 10.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.4 void fnProcessReadScalingDataByIdentifier(*pstRxData)

This function handles the reception and processing of the Read Scaling Data By Identifier message. The Read Scaling Data By Identifier message allows the client specify a data identifier who's scaling data it wishes to read.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.4.1 __weak U_8 fnReadScalingDataByIdentifierCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Read Scaling Data By Identifier message. The application code should evaluate the request and respond with the requested data or an error. See ISO 14229-1 Section 10.4.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.5 void fnProcessReadDataByPeriodicIdentifier(*pstRxData)

This function handles the reception and processing of the Read Data By Periodic Identifier message. The Read Data By Periodic Identifier message allows the client specify at what cyclic rate a data identifier will broadcast.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.5.1 __weak U_8 fnReadDataByPeriodicIdentifierCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Read Data By Periodic Identifier message. The application code should evaluate the request and respond with conformation that the identifier is setup to broadcast at a cyclic rate or an error. See ISO 14229-1 Section 10.5.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.5.2 void fnCheckPeriodicDataIdentifiers()

This function checks to see if there are any periodic data identifiers active and ready to transmit. If so it builds the necessary message and passes it to the ISO 15765 module for transmission.

12.6.5.3 tPERIODIC_IDENTIFIER * fnGetPeriodicData()

This function returns a pointer to the structure that contains the list periodic identifiers.

12.6.6 void fnProcessDynamicallyDefineDataIdentifier(*pstRxData)

This function handles the reception and processing of the Dynamically Define Data Identifier message. The Dynamically Define Data Identifier message allows the client create a data identifier that contains a grouping of other data identifiers or a block of memory.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.6.1 __weak U_8 fnDynamicallyDefineDataIdentifierCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Dynamically Define Data Identifier. The application code should evaluate the request and respond by building the dynamic data identifier or generating an error. See ISO 14229-1 Section 10.6.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.7 void fnProcessWriteDataByIdentifier(*pstRxData)

This function handles the reception and processing of the Process Write Data By Identifier message. The Process Write Data By Identifier message allows the client to write data to the server in the location specified by the passed data identifier.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.7.1 __weak U_8 fnWriteDataByIdentifierCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Write Data By Identifier message. The application code should evaluate the request and respond by writing the data to memory or returning an error. See ISO 14229-1 Section 10.7.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.6.8 void fnProcessWriteMemoryByAddress(*pstRxData)

This function handles the reception and processing of the Process Write Memory By Address message. The Process Write Memory By Address message allows the client to write data to the server in the specified memory location.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.6.8.1 __weak U_8 fnWriteMemoryByAddressCallback(*pstRxData)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Write Memory By Address message. The application code should evaluate the request and respond by writing the data to memory or returning an error. See ISO 14229-1 Section 10.8.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.7 ISO_14229_RoutineControl.c

12.7.1 Globals

None

12.7.2 void fnProcessRoutineControl(*pstRxData)

This function handles the reception and processing of the Routine Control message. The Routine Control message allows the client specify a given routine that is to be enabled/disabled.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.7.2.1 __weak U_8 fnRoutineControlCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Routine Control message. The application code should evaluate the requesting sub-function and enable/disable the called for routine or generate an error. See ISO 14229-1 Section 13.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any manufacture specific positive response information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

12.8 ISO_14229_DTC.c

12.8.1 Globals

static tDTC gstDTCs – Structure containing a list of DTC, their status bit, severity bits,…
static tDTC gstDTCMirrorMemory – Structure containing a mirrored copy of the structure gstDTCs.

12.8.2 void fnDTCListInit(u8DTCStatusAvailabilityMask, u8FormatIdentifier)

This function handles the initialization of all variables in the ISO_14229_DTC module. The passed data contains:

u8DTCStatusAvailabilityMask – Value to set the DTC status availability mask. This mask represents the significant bits within the status bits mask. This mask resides in the gstDTCs structure.
u8FormatIdentifer – Value to the format identifier. This value should represent the method used to define the DTCs. This mask resides in the gstDTCs structure.

12.8.3 BOOLEAN fnAddDTCToList(*pstDTC)

This function adds an item to the DTC list used by the stack.

The passed data contains:
pstDTC – Pointer to a structure containing the information for the DTC that is to be added to the list of DTCs.

The returned data contains:
A Boolean value where TRUE (1) indicates that the DTC was successfully added to the list, FALSE (0) indicates that the DTC was not added to the list.

12.8.4 BOOLEAN fnRemoveDTCFromList(u32DTC)

This function removes an item from the DTC list used by the stack.

The passed data contains:
u32DTC – The DTC who's information is to be removed from the list of DTCs.

The returned data contains:
A Boolean value where TRUE (1) indicates that the DTC was successfully removed from the list, FALSE (0) indicates that the DTC was not removed from the list.

12.8.5 tDTC_INFO *fnGetDTCFromList(u32DTC)

This function returns a pointer to an item in the list of DTCs used by the stack.

The passed data contains:
u32DTC – The DTC who's information is to be found in the list of DTCs.

The returned data contains:
A pointer to the item containing the requested DTC information.

12.8.6 tDTC_INFO * fnGetDTCFromMirrorMemory(u32DTC)

This function returns a pointer to an item in the list of DTCs in mirrored memory.

The passed data contains:
u32DTC – The DTC who's information is to be found in the list of mirrored memory DTCs.

The returned data contains:
A pointer to the item containing the requested DTC information.

12.8.7 void fnSetTestFailedDTC(u32DTC, flClear)

This function sets the first test failed and most recent test failed variables.

The passed data contains:
u32DTC – The DTC who's value is saved.
flClear - TRUE = clear the u32FirstTestFailedCode and u32MostRecentTestFailed variables. FALSE = set the u32MostRecentTestFailed variable with the passed DTC and the u32FirstTestFailedCode variable if it hasn't already be set.

12.8.8 void fnSetConfirmedDTC(u32DTC, flClear)

This function sets the first test failed and most recent test failed variables.

The passed data contains:
u32DTC – The DTC who's value is saved.
flClear - TRUE = clear the u32FirstConfirmedCode and u32MostRecentTestFailedCode variables. FALSE = set the u32MostRecentTestFailedCode variable with the passed DTC and the u32FirstConfirmedCode variable if it hasn't already be set.

12.8.9 void fnProcessReadDTCInformation(*pstRxData)

This function handles the reception and processing of the Read DTC Information message. The Read DTC Information message allows the client read the DTCs that are present in the stack.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.8.9.1 __weak U_8 fnReportNumberOfDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Number Of DTC By Status Mask message. The application code should respond with the number of DTC that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.2 __weak U_8 fnReportDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC By Status Mask message. The application code should respond with list of DTCs that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.3 __weak U_8 fnReportDTCSnapshotIdentificationCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Snapshot Identification message. The application code should respond with a list of snapshot identifiers. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.4 __weak U_8 fnReportDTCSnapshotRecordByDTCNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Snapshot Record By DTC Number message. The application code should respond with the snapshot record that matches the passed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.5 __weak U_8 fnReportDTCStoredDataByRecordNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Stored Data By Record Number message. The application code should respond with the DTC information corresponding to the passed record number. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.6 __weak U_8 fnReportDTCExtDataRecordByDTCNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Ext Data Record By DTC Number message. The application code should respond with the extended data record for the passed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.7 __weak U_8 fnReportNumberOfDTCBySeverityMaskRecordCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Number Of DTC By Severity Mask message. The application code should respond with the number of DTC that match the passed status mask and severity mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.8 __weak U_8 fnReportDTCBySeverityMaskRecordCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC By Severity Mask Record message. The application code should respond with a list of DTCs that match the passed status mask and severity mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.9 __weak U_8 fnReportSeverityInformationOfDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Severity Information Of DTC message. The application code should respond with the DTC status and severity bits for the passed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.10 __weak U_8 fnReportSupportedDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Supported DTC message. The application code should respond with a list of all available DTCs. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.11 __weak U_8 fnReportFirstTestFailedDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the First Test Failed DTC message. The application code should respond with the DTC and status bits of the first test failed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.12 __weak U_8 fnReportFirstTestConfirmedDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the First Confirmed DTC message. The application code should respond with the DTC and status bits of the first Confirmed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.13 __weak U_8 fnReportMostRecentTestFailedDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Most Recent Test Failed DTC message. The application code should respond with the DTC and status bits of the most recent test failed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.14 __weak U_8 fnReportMostRecentConfirmedDTCCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Most Recent Confirmed DTC message. The application code should respond with the DTC and status bits of the most recent confirmed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.15 __weak U_8 fnReportMirrorMemoryDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Mirror Memory DTC By Status Mask message. The application code should respond with a list of DTCs from the mirror memory that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.16 __weak U_8 fnReportMirrorMemoryDTCExtDataRecordByDTCNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Mirror Memory DTC Ext Data Record By DTC Number message. The application code should respond with extended data record from mirrored memory that matches the passed DTC. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.17 __weak U_8 fnReportNumberOfMirrorMemoryDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Number Of Mirror Memory DTC By Status Mask message. The application code should respond with the number of DTC in mirrored memory that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.18 __weak U_8 fnReportNumberOfEmissionsOBDByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Number Of Emissions OBD By Status Mask message. The application code should respond with the number of OBD DTC that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.19 __weak U_8 fnReportEmissionsOBDDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Emissions OBD DTC By Status Mask message. The application code should respond with a list of OBD DTCs that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.

pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.20 __weak U_8 fnReportDTCFaultDetectionCounterCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Fault Detection Counter message. The application code should respond with a list of all DTCs that contain a fault detection counter greater than 0. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.21 __weak U_8 fnReportDTCWithParameterStatusCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC With Parameter Status message. The application code should respond with a list of DTCs that have the permanent DTC flag set. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.22 __weak U_8 fnReportDTCExtDataRecordByRecordNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the DTC Ext Data Record By Record Number message. The application code should respond with a list of extended data records that match the passed record number. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.23 __weak U_8 fnReportUserDefMemoryDTCByStatusMaskCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the User Def Memory DTC By Status Mask message. The application code should respond

with a list of DTC that are from a specific user defined memroy area that match the passed status mask. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.24 __weak U_8 fnReportUserDefMemoryDTCSnapshotRecordByDTCNumberCallback (*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the User Def Memory DTC Snapshot Record By DTC Number message. The application code should respond with the snapshot record from a specific user defined memory area that match the passed DTC number. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.25 __weak U_8 fnReportUserDefMemoryDTCExtDataRecordByDTCNumberCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the User Def Memory DTC Ext Data Record By DTC Number message. The application code should respond with the extended data record from a specific user defined memory area that match the passed DTC number. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.26 __weak U_8 fnReportWWHOBDDTCByMaskRecordCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the WWH OBD DTC By Mask Record message. The functionality of this function is defined in ISO 27145-3 and is beyond the scope of this stack. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.9.27 __weak U_8 fnReportWWHOBDDTCWithParameterStatusCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the WWH OBD DTC With Parameter Status message. The functionality of thisfunction is defined in ISO 27145-3 and is beyond the scope of this stack. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.8.10 void fnProcessClearDiagnosticInformation(*pstRxData)

This function handles the reception and processing of the Clear Diagnostic Information message. The Clear Diagnostic Information message allows the client clear information concerning the DTCs in the stack DTC list.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.8.10.1 __weak U_8 fnClearDiagnosticInformationCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that can be strongly implemented on the application code side. It is called when the server successfully receives the Number Of DTC By Status Mask message. The application code should respond by clearing the all DTC status bits, counters, first test failed, most recent test failed, confirmed and most recent confirmed data. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.9 ISO_14229_InputOutputControl.c

12.9.1 Globals

None

12.9.2 void fnProcessInputOutputControlByIdentifier(*pstRxData)

This function handles the reception and processing of the Input Output Control By Identifier message. The Input Output Control By Identifier message allows the client manipulate the content of variables defined by a given data identifier.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.9.2.1 __weak U_8 fnInputOutputControlByIdentifierCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Input Output Control By Identifier message. The application code should evaluate and act upon the request or generate and error. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

12.10ISO_14229_ResponseOnEvent.c

12.10.1 Globals

None

12.10.2 void fnProcessResponseOnEvent(*pstRxData)

This function handles the reception and processing of the Response On Event message. The Response On Event message allows the client to setup the server to transmit a given set of data when a predefined event occures.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.

12.10.2.1 __weak U_8 fnResponseOnEventCallback(*pstRxData, *pau8TxData, *pu16TxCount)

This function is a weak function that needs to be strongly implemented on the application code side. It is called when the server successfully receives the Response On Event message. The application code should evaluate the requested sub-function and act accordingly. See ISO 14229-1 Section 11.3.

The passed data contains:
pstRxData – Pointer to a structure containing the received message and session data.
pau8TxData – Pointer to a transmit buffer where any DTC information is placed.
pu8TxCount – Pointer to a variable that user of this function will place the number of significant bytes entered into the pau8TxData buffer.

The returned data contains:
An error is returned if the request cannot be accomplished. The error is transmitted to the client.

13. System Test Modules

13.1 WarningFlags.c

13.1.1 Globals

U_8 gu8WarningFlags[LAST_MODULES] – This array of flags indicates if a warning has occurred with the stack.

```
enum eMODULES
{
    CAN_ENGINE,                 - tag for the CANEngine.c module flags
    INTERRUPT,                  - tag for the Interrupt.c module flags
```

```
    LOAD_ID,                        - tag for the LoadId.c module flags
    TX_RX_DRIVER,                   - tag for the TxRxDriver.c module flags
    ACK_MODULE,                     - tag for the Acknowledgment.c module flags
    ADDRESS_CLAIM_MODULE,           - tag for the AddressClaim.c module flags
    DIAGNOSTIC_MESSAGE,             - tag for the DiagnosticMessage.c module flags
    REQUEST_MODULE,                 - tag for the Request.c module flags
    TRANSPORT_PROTOCOL,             - tag for the TransportProtocol.c module flags
    TX_DATA,                        - tag for the TxData.c module flags
    FASTPACKET_MODULE,              - tag for the FastPacket.c module flags
    REQUEST_COMMAND_ACK_MODULE,     - tag for the RequestCommandAck.c module flags
    TRANSMIT_PGNS_MODULE,           - tag for the TransmitPGNs.c module flags
    LAST_MODULES                    - Count for the number of entries in the enum statement
};
```

13.1.2 void fnWarningFlagsInit()

This function does the initialization of any variables that are global to this module.

13.1.3 void fnSetWarningFlag (u8Module, u8Flag)

This function uses the passed module tag to set the bit flag corresponding to the passed flag value.

13.1.4 U_8 fnGetWarningFlag (u8Module)

This function uses the passed module tag to find the flags associated with it and returns those flags

13.1.5 void fnResetWarningFlag (u8Module, u8Flag)

This function uses the passed module tag to locate the flags that are associated with it. It then resets the flag that corresponds to the passed flag number.

13.1.6 CANEngine.c Flags

Bit	Function	Definition
0	**fnSetMessageAsReceived**	There are more messages waiting for the channel then will fit in the buffer. Increase The size of the **#define CHANNEL_BUFFER_SIZE** which is located in *includes.h* or release some of the messages waiting for this channel by calling the function **fnReleaseChannelMessage.**.
1	**fnProcessRxQue**	Receive queue overflow. Increase the size of **#define MESSAGE_BUFFER_SIZE** which is located in *includes.h*.
2	**fnProcessRxQue**	A channel pointer buffer overflow has occurred. Increase the size of **#define CHANNEL_BUFFER_SIZE** which is located in *includes.*h or release some of the messages waiting for this channel by calling the function **fnReleaseChannelMessage.**
3	**fnReleaseChannelMessage**	There are no messages to release for the channel.
4	**fnSetMessageAsReceived**	The passed message is not needed by any of the channels
5	**fnReleaseChannelMessage**	The passed channel has not been defined.
6	Reserved	
7	Reserved	

13.1.7 LoadId.c Flags

Bit	Function	Definition
0	**fnLoadIdList**	The channel passed to the function is greater than the number of channels define or is greater

		than 16. Either add the channel to the **enum tCHANNELS** or use an existing channel as there are more than 16 defined.
1	**fnLoadIdList`**	The ID list buffer has overflowed. Increase the size of the buffer by increasing the size of the **#define ID_BUFFER_SIZE**.
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

13.1.8 TxRxDriver.c Flags

Bit	Function	Definition
0	**fnBufferRawRxData**	Temporary Rx queue overflow. Either increase the size of the queue by adjusting the **#define RX_DATA_BUFFER_SIZE** or call the function **fnProcessRxQueue** more often.
1	**fnBufferTxData`**	Temporary Tx queue overflow. Increase the size of the queue by adjusting the **#define TX_DATA_BUFFER_SIZE.**
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

13.1.9 Acknowledgment.c Flags

Bit	Function	Definition
0	**fnRxACKMessage**	The receive acknowledgment queue has overflowed. Increase the size of the **#define RX_ACK_BUFFER_SIZE** which is located in *includes.h* or call the function **fnRxACKMessasge** more often or make sure the function **fnReleaseACKMessage** is called after every returned message using function **fnGetAckMessage** or reduce the timeout of the received messages by adjusting the **#define RX_ACK_TIME_OUT** which is located in *includes.h*.
1	**fnTxACKMessage**	Transmit queue overflow. Increase the size of **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

13.1.10 DiagnosticMessage.c Flags

Bit	Function	Definition
0	**fnEvaluateMessage**	No receive DM1 sessions available. Increase the number of simultaneous DM1 sessions that can be stored by adjusted the **#define MAX_DM1_SESSIONS** which is located in *includes.h*.
1	**fnEvaluateMessage**	No receive DM2 sessions available. Increase the number of simultaneous DM2 sessions that can be stored by adjusted the **#define MAX_DM2_SESSIONS** which is located in *includes.h*.
2	**fnEvaluateMessage**	Receive DM1 session buffer overflow. Increase the size of each receive DM1 faults session by adjusting the **#define RX_DM1_FAULTS_BUFFER_SIZE** which is located in *includes.h*.

3	**fnEvaluateMessage**	Receive DM2 session buffer overflow. Increase the size of each receive DM2 faults session by adjusting the **#define RX_DM2_FAULTS_BUFFER_SIZE** which is located in *includes.h*.
4	**fnStartDM1Tx**	The function **fnSetupDM1TxMessage** must be called before the function **fnStartDM1Tx**. If this does not happen this flag will be set. The function **fnSetupDM1TxMessage** sets the pointer to the DM1 faults list. This must be done before the faults can be broadcast.
5	**fnLoadDMTxBuffer**	Transmit queue overflow. Increase the size of the transmit queue by adjusting the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
6	**fnLoadDMTxBuffer**	No DM session available. Increase the number of DM sessions for a buffer by changing the **#define MAX_DM1_SESSIONS** for the DM1 buffer and the **#define MAX_DM2_SESSIONS** for the DM2 buffer. These **#defines** are located in *includes.h*.
7	**fnSendDM3**	Transmit queue overflow. Increase the size of the transmit queue by adjusting the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.

13.1.11 Request.c Flags

Bit	Function	Definition
0	**fnCheckTxRequest**	This flag indicates when multiple requests for the same message have occurred within the same cycle of the **fnCheckTxRequest** function. This situation can be remedied by calling the function **fnCheckTxRequest** more often. Also Increasing the request time multiplier **RX_REQUEST_TIME**, which is located in *includes.c*, will slow the rate of transmission of the cyclic request messages.
1	**fnTxRequest**	Transmit queue overflow. Increase the size of **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

13.1.12 TransportProtocol.c Flags

Bit	Function	Definition
0	**fnTP_CM_BAM_RTS_Rx**	Broadcast announce receive queue overflow. Increase the size of the queue by changing the value of the **#define RX_BAM_BUFFER_SIZE** which is located in *includes.h*.
1	**fnTP_CM_BAM_RTS_Rx**	Direct connect receive queue overflow. Increase the size of the queue by changing the value of the **#define RX_CONNECT_BUFFER_SIZE** which is located in *includes.h*.
2	**fnTP_DT_Tx**	Transmit queue overflow. Increase the size of **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
3	**fnTP_DT_Rx**	Broadcast announce or direct connect receive queue overflow. Change the size of the queue by changing the **#define TP_CM_BUFFER_SIZE** which is located in *includes.h*.
4	**fnTP_DT_Rx**	Broadcast announce or direct connect receive queue may have overflowed. This does not mean that the message was not received. It only means that part of the last packet was not saved as there was not enough buffer space. The part of the last packet that was discarded should be the part that contains 0xFFs which indicates that the data locations within the packet are not used. Changing the size of the receive and transmit session buffers can be accomplished by changing the value of the **#define TP_CM_BUFFER_SIZE** which is located in *includes.h*.
5	Reserved	
6	Reserved	
7	Reserved	

13.1.13 TxData.c Flags

Bit	Function	Definition
0	**fnLoadTxMessageIntoList**	Cyclic transmit queue parameter group buffer overflow. Increase the size of the cyclic transmit queue parameter group buffer by changing the **#define TX_PARAMETER_GROUP_BUFFER_SIZE** which is located in *includes.h*.
1	**fnLoadTxMessageIntoList**	Cyclic transmit queue parameter buffer overflow. Increase the size of the cyclic transmit queue parameter buffer by changing the **#define TX_PARAMETER_BUFFER_SIZE** which is located in *includes.h*.
2	**fnProcessTxMessage**	Multipacket transmit queue overflow. Increase the size of the multipacket transmit queue by changing the **#define TX_BAM_BUFFER_SIZE** or **#define TX_CONNECT_BUFFER_SIZE** which are located in *includes.h.*
3	**fnProcessTxMessage**	Transmit queue overflow. Increase the size of **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

13.1.14 FastPacket.c Flags

Bit	Function	Definition
0	**fnProcessFastPacket**	Receive fastpacket queue gstFastPacket_Rx overflow. Increase the size of the fastpacket receive queue parameter group buffer by changing the **#define NUMBER_OF_RX_FASTPACKET_QUEUE** which is located in *includes.h*.
1	**fnLoadNMEATxMessageInfoList**	Cyclic fastpacket transmit queue parameter group buffer overflow. Increase the size of the fastpacket transmit parameter group buffer by changing the **#define MAX_FASTPACKET_PARAMETER_GROUPS** which is located in *includes.h*.
2	**fnLoadNMEATxMessageInfoList**	Cyclic fastpacet transmit queue parameter buffer overflow. Increase the size of the fastpacket transmit parameter buffer by changing the **#define MAX_FASTPACKET_PARAMETERS** which is located in *includes.h.*
3	**fnBuildNMEAData**	Build queue overflow. This parameter group contains more parameters then will fit into the passed byte array. Increase the size of **#define MAX_TX_FASTPACKET_SIZE** which is located in *includes.h*.
4	**fnProcessNMEATxMessage**	Tx message could not be loaded into the Tx queue. Either the stack is not in the BUS_ACTIVE_STATE or the Tx queue is full. Wait for the stack to successfully claim an address or increase the size of the Tx queue by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
5	**fnSetUpFastPacketTx**	Tx message could not be loaded into the Tx queue. Either the stack is not in the BUS_ACTIVE_STATE or the Tx queue is full. Wait for the stack to successfully claim an address or increase the size of the Tx queue by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
6	Reserved	
7	Reserved	

13.1.15 RequestCommandAck.c Flags

Bit	Function	Definition
REQUEST_COMMAND_ACK_MODULE0:		
0	**fnProcessNMEARequest**	A parameter field being requested is not valid. A response will be sent by the Acknowledgment message indicating which field was invalid.
1	**fnProcessNMEARequest**	A parameter instance being requested could not be found. A response will be sent by the Acknowledgment message indicating which instance could not be found.
2	**fnProcessNMEARequest**	Tx message could not be loaded into the Tx queue. Either the stack is not in the BUS_ACTIVE_STATE or the Tx queue is full. Wait for the stack to successfully

		claim an address or increase the size of the Tx queue by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
3	**fnProcessNMEARequest**	Multipacket transmit queue overflow. Increase the size of the multipacket transmit queue by changing the **#define TX_BAM_BUFFER_SIZE** or **#define TX_CONNECT_BUFFER_SIZE** which are located in *includes.h.*
4	**fnProcessCommandMessage**	A parameter field being requested is not valid. A response will be sent by the Acknowledgment message indicating which field was invalid.
5	**fnProcessCommandMessage**	A parameter instance being requested could not be found. A response will be sent by the Acknowledgment message indicating which instance could not be found.
6	**fnProcessCommandMessage**	A parameter field being requested is not valid. A response will be sent by the Acknowledgment message indicating which field was invalid.
7	**fnProcessAckMessage**	Acknowledgment message buffer overflow. Increase the size of the acknowledgment buffer by changing the **#define RX_ACK_NMEA_BUFFER_SIZE** which is located in *includes.h*.
REQUEST_COMMAND_ACK_MODULE1:		
0	**fnNMEATxAckMessage**	Ack message could not be loaded into the Tx queue. Either the stack is not in the BUS_ACTIVE_STATE or the Tx queue is full. Wait for the stack to successfully claim an address or increase the size of the Tx queue by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
1	**fnNMEATxAckMessage**	Multipacket transmit queue overflow. Increase the size of the multipacket transmit queue by changing the **#define TX_BAM_BUFFER_SIZE** or **#define TX_CONNECT_BUFFER_SIZE** which are located in *includes.h.*

13.1.16 TransmitPGNs.c Flags

Bit	Function	Definition
0	**fnLoadTxMessageIntoList**	Transmit PGNs message could not be loaded into the Tx queue. Either the stack is not in the BUS_ACTIVE_STATE or the Tx queue is full. Wait for the stack to successfully claim an address or increase the size of the Tx queue by changing the **#define TX_DATA_BUFFER_SIZE** which is located in *includes.h*.
1	**fnLoadTxMessageIntoList**	Multipacket transmit queue oferflow. Increase the size of the multipacket transmit queue by changing the **#define TX_BAM_BUFFER_SIZE** or **#define TX_CONNECT_BUFFER_SIZE** which are located in *includes.h*.
2	Reserved	
3	Reserved	
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

14. Examples

Module: TestCode.c contains numerous examples of how to implement the different functions of this stack to achieve the desired results.

14.1 gastFastPacketRx

This function shows how to setup the stack to receive multiple fastpacket and one single packet NMEA message using a const array. A const struct like gastConstNMEARxData can then be used to parse the parameters from the received messages. See CANAppCode.c for an example of how to get the received parameters from the stack.

14.2 fnCyclicSinglePacketMultipleInstancesInit

This function shows how to setup a single packet NMEA message with mulitple instances to broadcast at a cyclic rate.

The setup of a broadcast of the single instance of an NMEA message is the same except the NumberOfInstances variable is set to 1 and the parameter variables need not be arrays.

14.3 fnTxCommandMessageInit

This function demonstrates how to setup the NMEA Complex Request Function message. This particular example sets the complex command message to change the value of a parameter on a different device. It uses the fastpacket transport protocol.

14.4 fnTxCyclicMultipacketMessageInit

This function demonstrates how to setup a mulitpacket message for cyclic transmission. The first 2 parameter are sent in the first packet and the 3rd parameter is sent in the second packet. The stack handles building and the timely transmission of each packet.

14.5 fnTxCyclicSinglePacketMessageInit

This function demonstrates how to setup a single packet J1939/ISO message to broadcast at a cyclic rate.

If you wish to not have the stack broadcast the message at a cyclic rate but only when your application code wishes, then set the u32BroadcastRate = 0 and call the function fnSetParameterGroupTx() to force an immediate broadcast of the message. The fnSetParameterGroupTx() function can be called even if the message is setup to broadcast cyclically to force an immediate broadcast.

14.6 fnTxCyclicMultiPacketP2PMessageInit

This function demonstrates how to setup a multipacket J1939/ISO message to broadcast at a cyclic rate and to a given source address. This is the same as the function fnTxCyclicSinglePacketMessageInit()except that the start byte for some of the parameters is greater than 8. This message will cause 3 data packets to be transmitted because the 3rd parameter ends begins in the 2nd packet but ends in the 3rd.

14.7 fnRxSinglePacketMessage

This function demonstrates how to setup a single packet or mulitpacket J1939/ISO message to be received by the stack.

See the module CANAppCode.c, function fnCANAppEngine() for an example of how to process received message from the stack.

14.8 fnRequestRxSinglePacketMessage

This function demonstrates how to tell the stack to request a single packet or mulitpacket J1939/ISO message. By setting the u8RequestType variable to J1939_REQUEST this will cause the stack to automatically request the needed PGN at a rate of RX_REQUEST_TIME (Defined in the includes.h file).

See the module CANAppCode.c, function fnCANAppEngine() for an example of how to process received message from the stack..

14.9 fnGetFixedMixedRxMessages

This function demonstrates how to request a received Normal Fixed or Mixed message that is buffered in the stack.

14.10 fnSetFixedMixedTxMessages

This function demonstrates how to transmit a Normal Fixed or Mixed message.

14.11 fnSetNormalExtendedRxMessages

This function demonstrates how to setup the stack to receive an ISO 15765 Normal and Extended message.

14.12 fnGetNormalExtendedRxMessages

This function demonstrates how get received Normal an Extended ISO15765 data from the stack.

14.13 fnSetNormalExtendedTxMessages

This function demonstrates how to setup the stack to transmit an ISO 15765 Normal and Extended message.

14.14 fnISO15765NormalExtendedErrorCallback

This function demonstrates how to setup the stack's ISO15765 Normal and Extended error callback function.

14.15 fnISO15765FixedMixedErrorCallback

This function demonstrates how to setup the stack's ISO15765 Normal Fixed and Mixed error callback function.