Actisense Comms SDK

User Manual

Issue 1.07 (November 2010)
For use with NGT-1 firmware v2.180 (and above)
and ActisenseComms dll v1.1.2.0 (and above)

- Creates a simple and easy to use communications interface to any **Actisense Comms API** compatible product
- Multi-threaded buffered bidirectional interface for reliable communications
- Converts from the Actisense proprietary message format (BST) in to an easy to read and use structure interface
- Enables fast integration of an Actisense product interface into a users software product

Contents

Important Notices

When using this document, keep the following in mind:

The products described in this manual and the specifications thereof may be changed without prior notice. To obtain up-to-date information and/or specifications, contact Active Research Limited or visit the *Actisense website (www.actisense.com)*.

Active Research Limited will not be liable for infringement of copyright, industrial property right, or other rights of a third party caused by the use of information or drawings described in this manual.

Active Research Limited will not be held responsible for any damage to the user that may result from accidents or any other reasons during operation of the user's unit according to this document.

Foreword

Actisense recognises that instructions are often skipped, so we have aimed to write this document in an informative, yet direct manner that will aid the user. We have tried to cover all the points a typical user may need to know.

Please read all sections before using the **Actisense Comms SDK** in a Higher Level Application (HLA) and any related hardware products. Actisense will be better placed to help support the user who has a good understanding of this document as a whole.

Introduction

The **Actisense Comms SDK** has been developed to help simplify the integration of compatible Actisense products into a users software environment.

This SDK documentation, the Visual C++ example programs and the C# wrapper code should allow a software programmer to implement a communications link to a compatible Actisense product in a very short period of time.

The current list of Actisense products that are compatible with the Actisense Comms SDK are:

- NGT - NMEA 2000 PC Interface Gateway
- NGW - NMEA 2000 to NMEA 0183 Gateway

Full information on the complete **Actisense** product range can be found on the *Actisense website*.

General features

C-code function interface

All access functions required to send data to and receive data from the Actisense hardware product use a flexible C-code interface using __stdcall to maximise compatibility with the users software design compiler environment.

Multi-threaded bi-directional interface

The Actisense Comms API uses a very efficient multi-threaded bi-directional interface to minimise CPU usage requirements. Full buffering of data ensures secure communications even under very high data loads.

Converts from the Actisense proprietary message format (BST)

Allows the Actisense proprietary message format (BST) to be hidden from the users software, and instead offers up a simple structure interface that is easy to read and use.

Example programs showing usage

The included example programs in the SDK show real world usage of the Actisense Comms DLL and help to explain how the c-code function interface should be used to get the most out of it.

API modules

This section details the access functions that can be found in each module, their intended usage and any extra information that may help the software developer.

All standard API functions are of the following form:

```
int <ACommsFunctionName>( int Handle, ... )
```

1) All take an integer Handle as their first argument, this is the handle returned by the ACommsCreate function allowing these functions to access the required Actisense Comms resource pointed to by that handle.

2) All return an integer Error code. This error code should be zero (ES_NO_ERROR) if no error has occurred. If an error been detected by the API, the negative error code returned is defined in the "ARLErrorCodes" header file.

It is important to understand that any error code returned **immediately** by an **API** command function can only indicate a**n** error that is detectable by the **API** when it decodes the HLA's request parameters. The return of the ES_NO_ERROR code, should in no way be understood by the HLA as the API command having been successfully received and handled by the attached hardware device. No API-Hardware communication will have occurred at the time the API function returns it's error code value.

The attached hardware that the API is communicating with will return its own error code value when it acknowledges the requested API command. The HLA should decode this acknowledgement message to correctly understand the outcome of its request and therefore know what action it should do next.

API_AComms

This is the core interface module of the Actisense Comms library. It is the only header required to be added to any module that uses the Actisense Comms API - as it includes / brings in all the other API headers.

The functions contained in this module form the lowest level Actisense Comms object access functions and can allow creation of multiple handles to multiple Comms ports. In the future, this will be extended beyond serial ports to other transport media.

All the communications functions in this API allow access to the underlying communications objects. The library handles all the real time receiving and sending of data to all Comms resources that have been created and opened.

All AComms functions are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsCreate

Before using an Actisense Comms object, a handle must be obtained to a Comms resource using this function:

```
int ACommsCreate (int * pHandle)
```

Creates a new "Actisense Comms" resource, and returns a handle (via the passed in pointer reference) to allow any external API functions to access this resource through the API function set. The returned integer value indicates the error status (as detailed in the *API modules* section).

Valid Comms object handles created by the library will be positive 32-bit integers in the range of 1 - 0x7FFFFFFF.

If a handle could not be created, the returned handle will be set to ACOMMS_INVALID_HANDLE (zero) and one of these ARL error codes will be returned from the function:

ES11_INVALID_HANDLE
Unable to reserve resources for this handle. This is an internal Windows error
ES11_TOO_MANY_HANDLES
Maximum number of handles has been exceeded

ACommsDestroy

To destroy a Comms object no longer required, removing it from the system, use this function:

```
int ACommsDestroy (int Handle)
```

Destroys an "Actisense Comms" resource referred to by the supplied handle, to free up all memory and resources used by the given handle. Can be used to selectively remove a single Comms object's resources.

If an error occurs, this ARL error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsDestroyAll

To remove resources from all currently active Actisense Comms objects with one call, use this function:

```
int ACommsDestroyAll (void)
```

Destroys all "Actisense Comms" resources that are currently active, and frees up all memory and resources used by them. This function should be called as part of the clean up routine when the application is terminated.

If an error is detected, a negative ARL error code will be returned (refer to ARLErrorCodes header).

ACommsOpen

To open a Comms resource (previously created) for use by the Actisense Comms object, use this function:

```
int ACommsOpen (int Handle, int PortNumber,
                int Baudrate)
```

After a Comms resource has been created, a Comms port may be opened for this resource. This function opens a serial UART Comms stream for this handle.

The required system “COM” port number and Baud rate are supplied by the calling function. If the given handle has been perviously used by the API to open a Comms port, the open port will be automatically closed prior to being reopened with the new parameters.

If an error is detected during this operation, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_PORT_NUMBER_OUT_OF_RANGE
  Only port numbers from 1 to 250 are valid
ES11_COMMS_CANNOT_CLOSE
  Handle is already in use, and failed to close the port prior to being reopened with the new parameters
ES11_PORT_NUMBER_CANNOT_OPEN
  A Windows handle was not given for this resource (normally because port does not exist)
ES11_COMMS_CANNOT_GET_DCB
  An internal Windows error
ES11_COMMS_CANNOT_SET_DCB
  An internal Windows error
ES11_COMMS_CANNOT_SET_TIMEOUTS
  An internal Windows error

ACommsClose

To close a Comms resource that has been previously created and opened, use this function:

```
int ACommsClose (int Handle)
```

Closes the Comms port associated with the given handle. The Comms object is not destroyed, so this port can be reopened whenever required in the future.

If an error is detected, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMS_CANNOT_CLOSE
  An internal Windows error

ACommsGetPortNumber

To obtain the Comms port number associated with a particular Comms object handle, use this function:

```
int ACommsGetPortNumber (int Handle,
                         int *PortNumber)
```

Returns the physical / hardware number of the Comms port associated with the given Comms object handle.

If an error occurs, this ARL error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsGetPortBaudrate

To obtain the Comms Baud rate associated with a particular Comms object handle, use this function:

```
int ACommsGetPortBaudrate (int Handle, int *Baudrate)
```

Returns the physical / hardware baud rate of the Comms port associated with the given Comms object handle.

If an error occurs, this ARL error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsEnumerateSerialPorts

To obtain a list of all Enumerated Comms ports currently available to the system, use this function:

```
int ACommsEnumerateSerialPorts
                    (sPortEnum *PortEnum)
```

The user defined structure that is passed by reference will be filled with the active enumerated ports list currently available to the system.

On some PC installations that have a large number of virtual serial ports installed, this API function may take a noticeable amount of time to return. Under these exceptional conditions, it is recommended that the HLA find an alternative method of enumerating serial ports.

If an error occurs, this ARL error code will be returned:

ES11_PORT_ENUMERATOR_ERROR
  An internal Windows error

ACommsEnumerateSerialPortsGetName

To convert an Enumerated Comms port number in to a more user friendly text description, use this function:

```
char* ACommsEnumerateSerialPortsGetName
                                    (u32 PortNum)
```

As a companion / helper function to the Comms port Enumeration function (above), this takes the given enumerated port number and returns the 'description' text name associated with that port - which can be very helpful to the user to identify the correct Comms port.

As this is a helper function that cannot fail, it differs from the normal error return function format.

The local function string storage will only persist (without change) until this function called again, so the returned pointer cannot be used for a permanent reference. Instead, the calling function should copy this string as soon as the function returns the reference to it.

Note: Not all Comm ports have 'useful' description names registered to them - in these rare cases, a Null string will be returned by this function.

ACommsGetRxLoading

To obtain the Comm port's current Receive Loading factor associated with a particular Comms object handle, use this function:

```
int ACommsGetRxLoading (int Handle, int *pLoading)
```

Returns a percentage factor that indicates how much load the Comm port's receive channel, associated with the given Comms object handle, is currently experiencing.

The returned integer load factor has a resolution of 3 decimal places and must be divided by 1000 to get the true floating number.

A load factor of 100% indicates that the maximum possible number of bytes (set by the Comm port's baud rate) have been received in the last second.

If an error occurs, this ARL error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsGetTxLoading

To obtain the Comm port's current Transmit Loading factor associated with a particular Comms object handle, use this function:

```
int ACommsGetTxLoading (int Handle, int *pLoading)
```

Returns a percentage factor that indicates how much load the Comm port's transmit channel, associated with the given Comms object handle, is currently experiencing.

The returned integer load factor has a resolution of 3 decimal places and must be divided by 1000 to get the true floating number.

A load factor of 100% indicates that the maximum possible number of bytes (set by the Comm port's baud rate) have been transmitted in the last second.

If an error occurs, this ARL error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

API_BST

This module forms the low-level BST Protocol interface of the Actisense Comms library. Direct access to the functions in this module is not normally required by the Higher Level Application (HLA) written by the software developer.

Most developers can skip this section: the BST Protocol definition is not available for external use, so the contents of any messages read will be meaningless.

The normal procedure for gaining access to the data contained within BST messages is by setting up the associated callback functions for each message type so the data can be returned in easy to access structures.

All API_BST functions (except for the 'Write') are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsBST_Write

To send (write) a raw BST message to the BST transmit queue of the given Comms port handle, use this function:

```
int ACommsBST_Write (int Handle, sBSTMsg *msg)
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_BUFFER_OVERFLOW
  Given handle's queue is full - write operation had to remove the oldest queued message first

ACommsBST_Read

To get (read) the next raw BST message from the BST receive queue of the given Comms port handle, use this function:

```
int ACommsBST_Read  (int Handle, sBSTMsg *msg)
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_BUFFER_UNDERFLOW
  Given handle's queue is empty - read operation has failed to find a message

ACommsBST_GetRxQSize

To get the total number of BST messages (still to be read) that are waiting in the BST receive queue of the given Comms port handle, use this function

```
int ACommsBST_GetRxQSize (int Handle,
                          size_t *BufferSize)
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsBST_FlushRx

To empty (flush) all unread messages from the BST receive queue of the given Comms port handle, use this function:

```
int ACommsBST_FlushRx (int Handle);
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsBST_FlushTx

To empty (flush) all unsent messages from the BST transmit queue of the given Comms port handle, use this function:

```
int ACommsBST_FlushTx (int Handle);
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsBST_SetRxCallback

To setup a BST receive message type Callback function for the given Comms port handle, use this function:

```
int ACommsBST_SetRxCallback (int Handle,
                             void (*pFunc)(void*), void*p)
```

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

API_Command

This is the main operating module that handles the sending of commands and the receiving of responses from the Actisense hardware product.

Each command can have a callback function setup that will be actioned when the target device responds with its acknowledgement to the issued API command. The callback method is the simplest way to complete the 'send command, get the acknowledgement' sequence. **For full details, refer to section *Setting up Callbacks*.**

Some commands use the full acknowledgement format of 'Tag' and 'Data' sections, whilst the simpler ones contain the reduced format of 'Tag' section only.

For each command that has a defined acknowledgment, the required 'Decode' function to call from within the message callback handler and the required 'DataType' it uses is defined thus:

Decode : <Decode function to call>
DataType : <Decoded data type>

If an error is detected by the API in any of the Command functions detailed below, an ARL error codes will be returned by the API. The list of error codes common to all functions are:

ES11_INVALID_HANDLE
Given handle does not match any active handles
ES11_DECODE_NO_DEFINITION
Command undefined. Please contact Actisense
ES11_COMMAND_MESSAGE_UNINITIALISED
Command uninitialised. Please contact Actisense
ES13_BUFFER_OVERRUN
Encapsulate buffer overrun. Please contact Actisense

All API_Command functions (except for 'GetStream', 'SetStream', 'GetN2KAddress' and 'SetN2KAddress') are EXTERNAL to the PC and do communicate externally with the attached ARL device. Depending on the current 'SetStream', these are either [LOCAL] or [REMOTE] EXTERNAL messages.

ACommsCommand_GetStream

To retreive the current Command 'Stream' that is set in the DLL, use this function:

```
int ACommsCommand_GetStream (int Handle,
                             stream_t *Stream)
```

The 'stream_t' enumerator defines all the possible streams that API commands can be sent over. Refer to *ACommsCommand_SetStream* for full enum definitions.

The returned value could for example be kept hidden from the basic user (to keep it simple), and reflected on to the GUI for the advanced user (so they know where the commands are currently being sent).

All API detectable errors are part of the common error list detailed at the beginning of this section.

ACommsCommand_SetStream

To change the current Command 'Stream' to the required stream, use this function:

```
int ACommsCommand_SetStream (int Handle,
                             stream_t Stream)
```

The 'stream_t' enumerator defines all the possible streams that API commands can be sent over:

COMMANDSTREAM_BST
Command is sent in Actisense proprietary BST format over the local serial link. Use with Actisense BST interfaced devices connected to the local serial port e.g. NGT

COMMANDSTREAM_NMEA0183
Command is wrapped as a proprietary NMEA 0183 string formatted as Actisense $PARL and sent over the local serial link. Use with Actisense NMEA 0183 interfaced devices connected to the local serial port e.g. NGW

COMMANDSTREAM_NMEA2000
Command is wrapped as a proprietary NMEA 2000 message and sent to the target device whose address can be set using the function ACommsCommand_SetN2KAddress. Used for the targeting of remote Actisense devices over the NMEA 2000 network (via a local NGT device).

Sets the command 'stream' over which any subsequent commands will be sent to the value given (as long as it is a valid stream_t value).

When talking to a local NGT device, the stream must be left set to COMMANDSTREAM_BST - otherwise all local NGT communication will be lost.

If the requested stream is invalid, this ARL error code will be returned:

ES11_COMMAND_INVALID_STREAM
Command stream is invalid and has been ignored

ACommsCommand_GetN2KAddress

To retrieve the currently set target address that will be used to send remote commands on the NMEA 2000 bus, use this function:

```
int ACommsCommand_GetN2KAddress (int Handle,
                                 u8 *N2KAddress)
```

Returns the NMEA 2000 address to which any commands will be sent to when the stream is also set to 'NMEA 2000' (using *ACommsCommand_SetStream*). This value is a useful addition to any GUI to inform the user what device on the NMEA 2000 bus is currently targetted / active.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response.**

ACommsCommand_SetN2KAddress

To set the target address to be used when sending remote commands on the NMEA 2000 bus, use this function:

```
int ACommsCommand_SetN2KAddress (int Handle,
                                 u8 N2KAddress)
```

Sets the NMEA 2000 address to be used in subsequent commands when the stream is also set to 'NMEA 2000'.

This command should only be used when it is required to send commands to a remote NGT / NGW device on the NMEA 2000 network.

This ARL error code will be returned if the requested address is deemed invalid:

ES11_COMMAND_INVALID_ADDRESS
Commanded address not valid and has been ignored

This command does not affect the NMEA 2000 address of the local NGT/NGW device - as that address is claimed during startup by the gateway itself. However, the API can *influence* the first address that the gateway will try to claim at the next startup by sending the *ACommsCommand_SetCanConfig* command to set the 'Preferred Address' (that the gateway will try to claim on the next startup),

Note: **This does not guarantee you will get the preferred address**, only that the gateway will try to use this address for its first address claim attempt: if a higher priority device has already claimed this address, the NGT/NGW device will claim the first 'free' address available **after** the commanded preferred address.

ACommsCommand_Reboot

To cause a full re-boot of the Actisense target device, so that the Bootloader becomes active for its defined period, use this function:

int ACommsCommand_Reboot (int Handle)

Forces the target device to re-boot / restart. This is a full hardware re-boot that cycles the gateway back to Bootloader, then through to that, back to the Main Application firmware. Normally this is only used to force a return to Bootloader to allow reprogramming of target device's firmware.

This command is not normally of interest to the 'Higher Level Application' (HLA) software developer.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTag
DataType : ddConfirmSoftwareReset

ACommsCommand_ReInitMainApp

To cause a re-initialisation of the Main Application, use this function:

int ACommsCommand_ReInitMainApp (int Handle)

Forces the Main Application firmware on the target device to re-initialise without resetting back to the Bootloader.

This command is normally only required as a quick method of returning the device to the defined configuration when a 'session only' configuration has been used temporarily. Any configuration change commands that indicate that a re-initialisation will automatically occur do not need to be followed by this command.

Naturally, there will be a short period (of typically 250 milliseconds) whilst the device is re-initialising when it is unable to communicate and will ignore any further command messages. The end of this 'unavailable' period is indicated by the device sending the ddStartupStatus message, which can prove to be a very useful callback trigger to continue normal communications.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTag
DataType : ddReInitMainApp

ACommsCommand_CommitToEEPROM

To commit any config setting changes held in the target device to the EEPROM, so they are remembered for the next session, use this function:

int ACommsCommand_CommitToEEPROM (int Handle)

Any commanded changes to the device config settings are by default "**session-only**" changes. This means, for example, it is possible to change the baud rate and enable proprietary code output settings, but have these settings revert back to those stored in the EEPROM after the target device has reset / re-initialised.

If it is required for these setting changes to be remembered permanently, use this command to copy the "session-only" values in to the EEPROM, and in doing so protect them for future sessions.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTag
DataType : ddCommitToEEPROM

ACommsCommand_CommitToFlash

To commit any config setting changes held in the target device to the Flash config sector, so they are remembered for the next session, use this function:

int ACommsCommand_CommitToFlash (int Handle)

Currently this function is not active in any devices.

All detectable errors are part of the common error list. **Refer to the** *Setting up Callbacks* **section to handle the device response message.**

Decode : ACommsDecode_GetTag
DataType : ddCommitToFlash

ACommsCommand_GetHardwareInfo

To request the 'Hardware Information' from the target device, use this function:

int ACommsCommand_GetHardwareInfo (int Handle)

Requests the ARL product 'Hardware Information' from the target device, which includes:

- Bootloader's software version number
- Bootloader's Date and Time of program
- Main Application's software version number
- Main Application's Date and Time of program
- ARL Hardware (PCB) version number
- Total Operating Time (in seconds) since new
- Model Sub ID number
- 'Operating Mode' number

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetHardwareInfo
DataType : ddHardwareInfo

ACommsCommand_GetOperatingMode

To read the 'Operating Mode' from the target device, use this function:

int ACommsCommand_GetOperatingMode (int Handle)

Retrieves the current 'Operating Mode' from the target device. This mode indicates what operating rules the target is currently using. Valid modes for an NGT-1 are:

- **Transfer data using 'Normal rules'**.
  This is the default mode that the target will use from power-on. The currently defined 'Receive PGN list' determines what PGNs are transferred to PC, and which are ignored / blocked. This is the preferred operating mode if the required number of PGNs will fit in the Rx PGN Enable list limitations.

- **Transfer data using 'Receive All PGN rules'**.
  This is a special mode that will transfer all PGNs that are received to the PC. No filtering is currently available in this mode.

Refer to BEMProtocolEnums.h for the full list of 'Operating Mode' enumerations.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsCommand_GetOperatingMode
DataType : ddOperatingMode

ACommsCommand_SetOperatingMode

To set the 'Operating Mode' of the target device, use this function:

int ACommsCommand_SetOperatingMode (int Handle, u16 OperatingMode)

Sends the new 'Operating Mode' to the target device. This new mode indicates what operating rules the target will use for the remainder of this session (until a power or software reset occurs). Valid modes for an NGT-1 are:

- **Transfer data using 'Normal rules'**.
  This is the default mode that the target will use from power-on. The currently defined 'Receive PGN list' determines what PGNs are transferred to PC, and which are ignored / blocked.

- **Transfer data using 'Receive All PGN rules'**.
  This is a special mode that will transfer all PGNs that are received to the PC. No filtering is currently available in this mode.

This new mode will only operate until the current session ends (due to a reset), at which point the default mode of the device will be started.

Refer to BEMProtocolEnums.h for the full list of 'Operating Mode' enumerations.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsCommand_GetOperatingMode
DataType : ddOperatingMode

ACommsCommand_GetHardwareBaudCodes

To read the 'Hardware Baud codes' from the target device, use this function:

```
int ACommsCommand_GetHardwareBaudCodes
                                    (int Handle)
```

Retrieves the current 'Hardware Baud code' list from the target device. This list details what Baud rates are currently in use in the target's hardware ports, and has the format:

- Number of Channels / Baud rate codes
- Channel 0: Hardware Protocol being used
- Channel 0: Baud rate Code
  ... repeated for each channel in device.

The Hardware Protocol enumeration can be used to understand what each channel's physical layer / type is (refer to ARLBaudCodes.h for details).

These codes can be passed to the two API helper functions *ACommsDecode_GetCANBaudCodeName* and *ACommsDecode_GetUARTBaudCodeName* in order to convert the codes into more meaningful text strings.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetHardwareBaudCodes
DataType : ddHardwareBaud

ACommsCommand_SetHardwareBaudCodes

To command the 'Hardware Baud codes' of the target device, use this function:

```
int ACommsCommand_SetHardwareBaudCodes
        (int Handle, int nBaudCodes, u8 *BaudCodes)
```

Sends a new 'Hardware Baud code' list to the target device, detailing the requested Baud Code for each channel.

If a particular channel's Baud Code is not required to be changed, the DONOT_CHANGE_U8 definition should be used (refer to BEMProtocolEnums.h). Likewise, if the default Baud Code for a channel is required, the USE_DEFAULTS_U8 define should be used.

The device will respond to this command with its standard acknowledgement and then change its hardware Baud rates to match the new values (if valid). Therefore, The Higher Level Application that triggered this Baud change must reopen its Comms port using the matching Baud rate, otherwise communication with the target will be lost after receiving the acknowledge.

This function does not update the Port Baud Code EEPROM config values, so these changes will be lost when a re-initialisation (or power cycle) occurs.

If any Baud rate is deemed invalid by the target, the acknowledgment will detail the error and what the new Baud rate list is. The Higher Level Application should decode this reply to handle this situation correctly.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetHardwareBaudCodes
DataType : ddHardwareBaud

ACommsCommand_GetPortBaudCodes

To read the 'Port Baud codes' from the target device, use this function:

int ACommsCommand_GetPortBaudCodes (int Handle)

Retrieves the current 'Port Baud code' list from the target device. This list details what Baud rates are currently set in the target's EEPROM configuration, and not necessarily the actual Baud rates currently in use by the hardware. The message format is:

- Number of Channels / Baud rate codes
- Channel 0: Hardware protocol being used
- Channel 0: Baud rate code
  ... repeated for each channel in device.

The Hardware Protocol enumeration can be used to understand what each channel's physical layer / type is (refer to ARLBaudCodes.h for details).

To obtain the actual hardware Baud code values, use the function *ACommsCommand_GetHardwareBaudCodes*.

These codes can be passed to the two API helper functions *ACommsDecode_GetCANBaudCodeName* and *ACommsDecode_GetUARTBaudCodeName* in order to convert the codes into more meaningful text strings.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortBaudCodes
DataType : ddPortBaudCfg

ACommsCommand_SetPortBaudCodes

To command the 'Port Baud codes' of the target device, use this function:

int ACommsCommand_SetPortBaudCodes (int Handle, int nBaudCodes, u8 *BaudCodes)

Sends a new 'Port Baud code' list to the target device.

If a particular channel's Baud Code is not required to be changed, the DONOT_CHANGE_U8 definition should be used (refer to BEMProtocolEnums.h). Likewise, if the default Baud Code for a channel is required, the USE_DEFAULTS_U8 define should be used.

The device will respond to this command with its standard acknowledgement and automatically store the Baud rate changes to its EEPROM, hence making these changes persistent.

These new Baud code values will not be used to configure the hardware ports until a re-initialise event occurs (through power cycle or command), so the Higher Level Application should not change its Comms port settings after sending this command.

Using the *ACommsCommand_SetHardwareBaudCodes* command after this function will allow the new baud rates to be used for all new communications.

If any Baud rate is deemed invalid by the target, the acknowledgment will detail the error and what the new baud rate list is. The Higher Level Application should decode this reply to handle this situation correctly.

There is **no need** to follow this command with the EEPROM *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortBaudCodes
DataType : ddPortBaudCfg

ACommsCommand_GetPortPCodes

To read the 'Port P-codes' from the target device, use this function:

int ACommsCommand_GetPortPCodes (int Handle)

Retrieves the current 'Port P-code' list from the target device. This list details what ports have their ARL 'P-code' message output enabled (and which do not), and has the format:

- Channel 0: Proprietary Code enabled / disabled
  ... repeated for each channel in device.

The channel's enable / disable state can be:

- **0**: P-Codes are disabled permanently. No proprietary ARL message will be transmitted on this channel.
- **1**: P-codes are enabled permanently. The device will transmit proprietary ARL messages on this channel in this session and any future sessions.
- **2**: P-codes are enabled for this session only. The device will transmit proprietary ARL messages on this channel during this session, but will stop when a reset occurs (from a power-cycle or command).

From firmware v2.172 and onwards these current values are the same as the EEPROM configuration values.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortPCodes
DataType : ddPortPCodeCfg

ACommsCommand_SetPortPCodes

To command the 'Port P-code' settings of the target device, use this function:

int ACommsCommand_SetPortPCodes (int Handle,
int nPCodes, u8 *PCodes)

Sends a new 'P-code' list to the target device. The target device is capable of sending additional status and debug information from the target's ports, as NMEA proprietary messages.

This information can be suppressed by setting a "0", or enabled by setting a "1" in the corresponding channel location in the list. Setting to a "2" has the useful feature of enabling the transmission of proprietary messages for this session only - once a reset occurs (from a power-cycle or command), the P-Code enable state will return to disabled ("0").

If a particular channel's P-Code is not required to be changed, the DONOT_CHANGE_U8 definition should be used (refer to BEMProtocolEnums.h). Likewise, if the default P-Code state for a channel is required, the USE_DEFAULTS_U8 define should be used.

If any P-code setting is deemed invalid by the target, the acknowledgment will detail the error and what the new P-code list is. **The Higher Level Application (HLA) should always decode this reply to correctly handle this feedback situation**.

If the new 'P-code' settings should be made permanent / remembered for future sessions, then follow with the *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortPCodes
DataType : ddPortPCodeCfg

ACommsCommand_GetPortDupDelete

To read the 'Port Duplicate Delete' settings from the target device, use this function:

int ACommsCommand_GetPortDupDelete (int Handle)

Retrieves the current 'Port Duplicate Delete' list from the target device. This list details what ports will have their 'Duplicate' messages (if the overall bandwidth requirements become overloaded (and which do not), and has the format:

- Channel 0: Duplicate Deletion enabled / disabled
  ... repeated for each channel in device.

The channel's enable / disable state can be:

- **0**: Duplicate Deletion is disabled permanently. No duplicate deletions will be performed on this channel.
- **1**: Duplicate Deletion is enabled permanently. The device will perform duplicate deletions on this channel (when the specific conditions are met) in this session and any future sessions.

From firmware v2.172 and onwards these current values are the same as the EEPROM configuration values.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortDupDelete
DataType : ddPortDupDelete

ACommsCommand_SetPortDupDelete

To command the 'Port Duplicate Delete' settings of the target device, use this function:

int ACommsCommand_SetPortDupDelete (int Handle,
int nPortDupDelete, u8 *PortDupDelete)

Sends a new 'Duplicate Delete' list the target device. The target device defaults to deleting any duplicate messages it receives, to reduce the required bandwidth.

This operation can be suppressed by setting a "0", or enabled by setting a "1" in the corresponding channel location in the list.

If a particular channel's 'Duplicate Delete' is not required to be changed, the DONOT_CHANGE_U8 definition should be used (refer to BEMProtocolEnums.h). Likewise, if the default 'Duplicate Delete' state for a channel is required, the USE_DEFAULTS_U8 define should be used.

If any 'Duplicate Delete' setting is deemed invalid by the target, the acknowledgment will detail the error and what the new 'Duplicate Delete' list is. The Higher Level Application (HLA) should decode this reply to handle this situation correctly.

If the new 'Duplicate Delete' settings should be made permanent / remembered for future sessions, then follow with the *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetPortDupDelete
DataType : ddPortDupDelete

ACommsCommand_GetTotalTime

To get the 'Total (Operating) Time' from the target device, use this function:

int ACommsCommand_GetTotalTime (int Handle)

Retrieves the current total operating time from the target device. The time is expressed in seconds, and is an unsigned 32-bit number.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTotalTime
DataType : ddTotalTime

ACommsCommand_SetTotalTime

To set the 'Total (Operating) Time' to a required value (in seconds), use this function:

```
int ACommsCommand_SetTotalTime (int Handle,
                                u32 TotalTime, u32 Passkey)
```

Sets the 'Total (Operating) Time' to the time (in seconds) requested, as long as the required Passkey value is valid. **This command should not normally need to be used** - as this value should normally remain unchanged, hence the requirement to use a valid passkey to limit its use to qualified software only.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTotalTime
DataType : ddTotalTime

ACommsCommand_GetProductInfoN2K

To get the NMEA 2000 'Product Information' details from the target device, use this function:

```
int ACommsCommand_GetProductInfoN2K
                                (int Handle)
```

Retrieves the NMEA 2000 'Product Information' from the target device. This data corresponds to that accessible via the NMEA 2000 bus by requesting PGN 126996 'Product Information', which includes:

- NMEA 2000 Support version number
- NMEA 2000 Certification Level
- NMEA 2000 Load Equivalency
- Manufacturers Model ID text string
- Manufacturers Software version code text string
- Manufacturers Model / Hardware version text string
- Manufacturers Model serial Code text string

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetProductInfoN2K
DataType : ddProductInfoN2K

ACommsCommand_GetCanConfig

To get the CAN 'Configuration' details from the target device, use this function:

```
int ACommsCommand_GetCanConfig (int Handle)
```

Retrieves the CAN 'Configuration' from the attached device. The 'CAN Name' part of this data corresponds to that accessible via the NMEA 2000 bus by requesting PGN 60928 'Address Claim'. In total, this configuration message includes:

- CAN 'Preferred Address' (to start address claim with)
- CAN Name (used to claim address)
- CAN 'Previously Claimed Address' (last session)
- CAN 'Source Address' claimed (this session)
- CAN 'Source Address' valid ('claimed' / 'not claimed')

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetCanConfig
DataType : ddCANConfig

ACommsCommand_SetCanConfig

To command the target device to use a requested 'Preferred Address', with a requested 'CAN Name' to claim its next NMEA 2000 address, use this function:

```
int ACommsCommand_SetCanConfig (int Handle,
  int Preferred, int SystemInstance, int DeviceInstance)
```

Sets the targeted Actisense NMEA 2000 device's 'Preferred Address' and CAN Name instances to those requested. Only the 'System instance' and 'Device instance ' fields in the CAN Name may be changed by the Higher Level Application (HLA). Modification of any other fields would invalidate the NMEA 2000 specification.

Once set, the target device will immediately perform a new Address Claim operation using the new 'Preferred Address' and 'CAN Name'. Whilst the target device will start its claiming process with the requested 'Preferred Address', if a higher priority device has already claimed that address, the device will claim the first 'free' address available **after** the commanded address).

There is **no need** to follow this command with the EEPROM *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetCanConfig
DataType : ddCANConfig

ACommsCommand_SetCanInfoField1
ACommsCommand_SetCanInfoField2
ACommsCommand_SetCanInfoField3

To set the CAN 'Information' fields 1 or 2 (Installation Description) in a target device to a required text string, use the corresponding function:

```
int ACommsCommand_SetCanInfoField1 (int Handle,
                                    const char *InfoString)
int ACommsCommand_SetCanInfoField2 (int Handle,
                                    const char *InfoString)
```

Sets the corresponding CAN 'Information' text string in the target device. This command corresponds to that accessible via the NMEA 2000 bus by commanding PGN 126998 'CAN Config Information'.

CAN 'Installation Description fields (1 and 2) are intended to be 'filled in' by the installer when the device is installed to add supplementary information about the product's location, power supply source / breaker etc. This will allow future identification of this device on the NMEA 2000 bus.

Currently, it is not possible for a Higher Level Application to set the CAN 'Information' field 3 (Manufacturer Information) - as this string must retain the Manufacturer's details. The target device will reply with a 'Not supported' response.

There is **no need** to follow this command with the EEPROM *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section, except:

ES11_COMMAND_UNEXPECTED_DATATYPE
DataType is not 'CAN Info Field' 1, 2, or 3

Refer to *Setting up Callbacks* to handle the device response message.

Decode : ACommsDecode_GetCanInfoField1-3
DataType : ddCANInfoField1-3

ACommsCommand_GetCanInfoField1
ACommsCommand_GetCanInfoField2
ACommsCommand_GetCanInfoField3

To get the CAN 'Information' fields 1, 2 or 3 from the target device, use the corresponding function:

```
int ACommsCommand_GetCanInfoField1 (int Handle)
int ACommsCommand_GetCanInfoField2 (int Handle)
int ACommsCommand_GetCanInfoField3 (int Handle)
```

Retrieves the CAN 'Information' from the target device. This data corresponds to that accessible via the NMEA 2000 bus by requesting PGN 126998 'CAN Information', this includes:

- CAN Installation Description, field 1
- CAN Installation Description, field 2
- CAN Manufacturer Information, field 3

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetCanInfoField1-3
DataType : ddCANInfoField1-3

ACommsCommand_SetRxPGN

To enable or disable a receive PGN in the target device, using the default PGN Mask, use this function:

```
int ACommsCommand_SetRxPGN (int Handle,
                            u32 PGN, PGNEnable_t Enable)
```

Uses a simplified version of the 'Extended' version (defined below) that enables or disables a PGN using the library default value for the PGN Mask (defined within the target). This function is recommended when a single PGN is to be set in a single Enable list 'slot'. For special cases where multiple PGN's are to be set in a single Enable list 'slot', refer to the *Rx PGN Enable list* section.

This function is the equivalent of calling the 'Extended' version (ACommsCommand_SetRxPGNEx) with the mask value set to USE_DEFAULT_PGN_MASK.

Refer to the 'Extended' version below for maximum size restrictions on the 'Rx PGN Enable list'.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetRxPGN
DataType : ddEnableRxPGN

ACommsCommand_SetRxPGNEx

To enable or disable a receive PGN in the target device, using a user defined PGN Mask, use this function:

```
int ACommsCommand_SetRxPGNEx (int Handle,
    u32 PGN, PGNEnable_t Enable, PGNMask_t Mask)
```

The 'Extended' version that enables or disables a PGN using the user defined PGN Mask. This function is recommended for special cases where multiple PGN's are to be set in a single Enable list 'slot' (except for the Proprietary 256 PGN blocks that can be set using the standard version above), refer to the *Rx PGN Enable list* section for further details.

The 'real' Rx PGN Enable List has a size of 28 'slots'.

The 'virtual' Rx PGN Enable List size is 7 'slots' - these are the core PGNs which are always received (and processed) by the target device, but can be optionally 'copied' back to the PC, if added to the 'Enable list'.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetRxPGN
DataType : ddEnableRxPGN

ACommsCommand_GetRxPGN

To get the required PGN's parameters (PGN & Mask, plus Enable status) from the target device, use this function:

```
int ACommsCommand_GetRxPGN (int Handle,
                            u32 PGN)
```

Simple operation to retrieve the single specified PGN's parameters (PGN & Mask, plus the Enable status) from the target device.

When it is more useful to retrieve the entire 'Rx PGN Enable list' in a single operation, the *ACommsCommand_GetRxPGNList* command should be used.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetRxPGN
DataType : ddEnableRxPGN

ACommsCommand_SetTxPGN

To enable or disable a transmit PGN in the target device, using the default PGN Tx Rate and/or Tx Timeout, use this function:

```
int ACommsCommand_SetTxPGN  (int Handle,
                             u32 PGN, PGNEnable_t Enable)
```

Uses a simplified version of the 'Extended' version (defined below) that enables or disables a PGN using the library default values for PGN Tx Rate and/or Tx Timeout (defined within the targets database).

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTxPGN
DataType : ddEnableTxPGN

ACommsCommand_SetTxPGNEx

To enable or disable a transmit PGN in the target device, using a user defined PGN Tx Rate and/or Tx Timeout, use this function:

```
int ACommsCommand_SetTxPGNEx (int Handle,
                              u32 PGN, PGNEnable_t Enable,
                              u32 Rate, u32 Timeout)
```

The 'Extended' version that enables or disables a PGN using the user defined PGN Tx Rate and/or Tx Timeout. This function is recommended for special cases where the default Tx Rate and/or Tx Timeout values are not acceptable to the user.

The 'virtual' Tx Enable List has a size of 30 'slots'.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetTxPGN
DataType : ddEnableTxPGN

ACommsCommand_GetTxPGN

To get the required PGN's parameters (PGN & Mask, plus Enable status) from the target device, use this function:

```
int ACommsCommand_GetTxPGN (int Handle,
                            u32 PGN)
```

Simple operation to retrieve the single specified PGN's parameters (PGN & Mask, plus the Enable status) from the target device.

When it is more useful to retrieve the entire 'Tx PGN Enable list' in a single operation, the *ACommsCommand_GetTxPGNList* command should be used.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetTxPGN
DataType : ddEnableTxPGN

ACommsCommand_GetRxPGNList

To get the current Rx PGN Enable List from the target device, use this function:

```
int ACommsCommand_GetRxPGNList (int Handle)
```

Retrieves the current RAM (session only) 'Rx PGN Enable list' from the target device. If the 'Activate' command has not been sent following any changes to the 'Rx PGN Enable list', the returned list will be the list being built up (by the API) and not the list actually being used by the target to pass PGN data on to the API. Refer to ACommsCommand_ActivatePGNEnableLists for more details on this distinction.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetRxPGNList
DataType : ddRxPGNEnableList

ACommsCommand_GetTxPGNList

To get the current Tx PGN Enable List from the target device, use this function:

```
int ACommsCommand_GetTxPGNList (int Handle)
```

Retrieves the current RAM (session only) 'Tx PGN Enable list' from the target device. If the 'Activate' command has not been sent following any changes to the 'Tx PGN Enable list', the returned list will be the list being built up (by the API) and not the list actually being used by the target to pass PGN data on to the NMEA 2000 bus. Refer to ACommsCommand_ActivatePGNEnableLists for more details on this distinction.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetTxPGNList
DataType : ddTxPGNEnableList

ACommsCommand_ClearPGNList
ACommsCommand_ClearRxPGNList
ACommsCommand_ClearTxPGNList

To clear the entire Rx, Tx or both 'PGN Enable lists' in the target device, use the corresponding function:

```
int ACommsCommand_ClearPGNList (int Handle,
                                PGNEnableList_t ListID)
int ACommsCommand_ClearRxPGNList (int Handle)
int ACommsCommand_ClearTxPGNList (int Handle)
```

Commands the clearing of the requested 'PGN Enable list' - resulting in all PGN data being lost. This command is normally actioned immediately before building up a new list from scratch. The two separate functions of 'ClearRxPGNList' and 'ClearTxPGNList' allow the clearing of an individual list without sending the 'ListID' specifier.

All API detectable errors are part of the common error list detailed at the beginning of this section, except for:

ES11_COMMAND_DATA_OUT_OF_RANGE
'List ID' is an invalid value

Refer to *Setting up Callbacks* to handle the device response message.

Decode : ACommsDecode_GetTag
DataType : ddDeletePGNEnableList

ACommsCommand_ActivatePGNEnableLists

To activate both 'PGN Enable lists' in the target device, use this function:

```
int ACommsCommand_ActivatePGNEnableLists
                                    (int Handle)
```

Activates the new 'PGN Enable lists' in the target device by causing a re-initialisation of the CAN hardware so that the new values can be actively used. The simple command response indictes that the required activation has been triggered. This command will normally follow the building up of a new list by using the 'SetRxPGN' and 'SetTxPGN' commands.

This allows Higher Level Application (via the API) to clear the lists, build up a new set of lists, and then activate them while the unit is still processing data based upon the list that is currently active in the device. In this way, the time taken to 'swap' from one 'Enable list' to another is kept to the absolute bare minimum - reducing the 'down-time'.

If the new 'PGN Rx/Tx Lists' should be made permanent / remembered for future sessions, then follow with the *ACommsCommand_CommitToEEPROM* command.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to *Setting up Callbacks* to handle the device response message.**

Decode : ACommsDecode_GetTag
DataType : ddActivatePGNEnableLists

ACommsCommand_ SetDefaultPGNEnableList

To command the target device to reset the requested 'PGN Enable list' back to the internally held (factory) defaults list, use this function:

```
int ACommsCommand_SetDefaultPGNEnableList
                    (int Handle, PGNEnableList_t ListID)
```

Returns the target device back to the (permanently stored in the firmware) factory defaults for the requested Rx, Tx or both 'PGN Enable lists'. Can be used as a quick 'reset to a known position' operation.

All API detectable errors are part of the common error list detailed at the beginning of this section, except for:

ES11_COMMAND_DATA_OUT_OF_RANGE
'List ID' is an invalid value

Refer to *Setting up Callbacks* to handle the device response message.

Decode : ACommsDecode_GetTag
DataType : ddDefaultPGNEnableList

ACommsCommand_ GetParamsPGNEnableLists

To get the Rx and Tx 'PGN Enable lists' parameters from the target device, use this function:

int ACommsCommand_GetParamsPGNEnableLists (int Handle)

Retrieves the current Rx and Tx 'PGN Enable list' parameters from the target device. This includes:

- Rx PGN Enable List
  - Number of 'real' PGNs in use in
  - Maximum capacity of 'real' PGNs
  - Number of 'virtual' PGNs in use
  - Maximum capacity of 'virtual' PGNs

- Tx PGN Enable List
  - Number of 'virtual' PGNs in use
  - Maximum capacity of 'virtual' PGNs

- 'Hardware - Enable List' synchronised status. If this is returned as 'false' /'Not synchronised', then the HLA must 'Activate' the new PGN lists in order to copy the new values to the hardware registers. A subsequent use of this command will then show this value as 'true' / 'Synchronised'.

All API detectable errors are part of the common error list detailed at the beginning of this section. **Refer to** *Setting up Callbacks* **to handle the device response message.**

Decode : ACommsDecode_GetParamsPGNEnableLists
DataType : ddParamsPGNEnableLists

API_CommsLog

This is a useful Actisense Comms data logging module. It allows for a logging output to an 'Enhanced Binary Log' (EBL) format. This is a pure binary format with embedded control codes to add time-stamping and other information to the file (to aid in reconstruction of the data timeline for replay and viewing of the data).

All API_CommsLog functions are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsLog_Enable

To enable / activate the logging of a given Comms port to a requested file, use this function:

```
int ACommsLog_Enable (int Handle,
                      const char* FileStem, u32 EnableMask)
```

Enables data logging to file (or files) for the supplied Actisense Comms port handle. The supplied file name is extended to include whether it has received ('Rx') or transmit ('Tx') data stored in it, plus the 'format' type used (currently limited to EBL only).

For example, FileStem of "Log", creates full File names of "Log.rx.ebl" (for the Rx EBL data) & "Log.tx.ebl" (for the Tx EBL data).

To turn off logging of all resources, pass in an 'Enable Mask 'of zero (i.e. no log bit flags are set).

If it is required to log both the Rx and Tx data to file, this must be 'Enabled' with a **single** call to this function. Any subsequent call to this function will close any previously enabled / active log files.

For example:
If the first call uses the mask of COMMSLOG_EBL_RX (to setup the 'Rx' data log file), and then a subsequent call uses the new mask COMMSLOG_EBL_TX (to setup the 'Tx' data log file), the second call will actually close the 'Rx' data log file setup by the first call (as the COMMSLOG_EBL_RX bit flag is zero in the second call).

To enable both Tx and Rx logging of Comms port 'MyHandle', to the log file 'log' (detailed above), use:

```
ACommsLog_Enable (MyHandle, "Log",
        (COMMSLOG_EBL_RX | COMMSLOG_EBL_TX))
```

If an error is detected, one of these ARL error codes will be returned by the API:

ES11_COMMS_LOG_FILE_ERROR
  FileStem text length is zero (Null)
ES11_COMMSLOG_FILENAME_TOOLONG
  FileStem text length plus extensions is too long
ES11_COMMSLOG_CANNOT_OPEN
  An internal Windows error

API_Decode

This module contains all the required decode functions necessary to understand the Actisense proprietary BST messages received from the attached Actisense device.

The information sent by the device using this proprietary binary coded format 'BST', is either:

- In response to an API command that is specified in the *API_Command* section of this document.
- Sent automatically, at startup of the device, when an operational error is detected by the device, or at regular intervals to provide system status data.

When decode callbacks are enabled for a particular data type, the API will automatically action a call to the Higher Level Application (HLA) defined handling function. The HLA may then retrieve the decoded message data for that data type, in an easy to access API structure.

In this way, the complexity of the BST message format is hidden and is replaced by an easy to read structure.

The API will automatically handle any API command response timeouts. The API and Higher Level Application (HLA) should follow this sequence of events:

- If the expected response has not been received by the API within the specified time (defaults to 2.0 seconds), the callback associated with that data type is called with the 'Decode Reason' set to 'drDecodeTimeout'.
- The HLA should handle this 'Decode Timeout' callback as it requires - e.g. by triggering a re-transmission of the original message, or by displaying the failure / timeout on the GUI for the user to interpret.
- The HLA must avoid calling the data types associated decode function - as this operation will naturally fail due to the response message not being received.

The data structures that are returned (by reference) in the following API 'Decode Rx message' functions, are fully documented in the 'API_Decode_Datatypes' header file.

All API_Decode functions are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsDecode_GetAge

To determine the 'Age' of the given received message 'Tag', use this function:

```
u32 ACommsDecode_GetAge
                    (sDecodeTag* DecodeTag)
```

Calculates the time difference (in milliseconds) between when the given message 'Tag' was received and the current system time.

Returned 32-bit value can hold a time difference of almost 50 days, with a millisecond resolution (far longer than the Rx buffer would be able to hold such a message for).

ACommsDecode_GetDataTypeName

To obtain the 'Data Type' name as a pointer to a text string, use this function:

```
const char * ACommsDecode_GetDataTypeName
                    (DecodeData_t DataType,
                     DecodeDetail_t DecodeDetail)
```

Returns a pointer to the null-terminated text string that corresponds to the given 'Data Type'. The referenced 'Name' text string can be useful when displaying callback's decoded data.

If an error is detected whilst retrieving the requested text string pointer, the pointer will be set to "Datatype Not Found" and this ARL error code will be returned:

```
ES11_DECODE_NO_DEFINITION
  Given 'Data Type' not recognised (has no definition)
```

ACommsDecode_GetUARTBaudCodeName

To obtain the UART 'Baud Code' name as a pointer to a text string, use this function:

```
con char* ACommsDecode_GetUARTBaudCodeName
                    (u32 Code, DecodeDetail_t DecodeDetail)
```

Returns a pointer to the null-terminated text string that corresponds to the given UART 'Baud Code'. The referenced 'Name' text string can be either 'Brief' or 'Full' (determined by the 'DecodeDetail' setting) and is and normally useful when displaying the decoded value of the target's UART Baud rate to the user.

This can help decode the Baud code values returned by the two API commands:

```
ACommsCommand_GetPortBaudCodes
ACommsCommand_SetPortBaudCodes
```

ACommsDecode_GetCANBaudCodeName

To obtain the CAN 'Baud Code' name as a pointer to a text string, use this function:

```
const char* ACommsDecode_GetCANBaudCodeName
                (u32 Code, DecodeDetail_t DecodeDetail)
```

Returns a pointer to the null-terminated text string that corresponds to the given CAN 'Baud Code'. The referenced 'Name' text string can be either 'Brief' or 'Full' (determined by the 'DecodeDetail' setting) and is normally useful when displaying the decoded value of the target's CAN Baud rate to the user.

This can help decode the Baud code values returned by the two API commands:

ACommsCommand_GetPortBaudCodes
ACommsCommand_SetPortBaudCodes

ACommsDecode_GetModelIDName

To obtain the ARL 'Model ID code' name as a pointer to a text string, use this function:

```
const char* ACommsDecode_GetModelIDName
                (u32 Code, DecodeDetail_t DecodeDetail)
```

Returns a pointer to the null-terminated text string that corresponds to the given ARL 'Model ID code'. The referenced 'Name' text string can be either 'Brief' or 'Full' (determined by the 'DecodeDetail' setting) and is normally useful when displaying the decoded values of the 'Tag' section of any message received from the target.

This can help decode the Model ID value found in all BST messages received from the target (in the 'Tag' section) via the decode function:

ACommsDecode_GetTag

This can be very useful in determining what the target is (an NGT-1 or an NGW-1) so the Higher Level Application (HLA) can alter the possible options that are available to the user accordingly.

ACommsDecode_SetCallback

To set up a callback function that is triggered by the reception of a specific BST message, use this function:

```
int ACommsDecode_SetCallback (int Handle,
                DecodeData_t Datatype,
void (*pFunc)(void*, DecodeData_t, DecodeReason_t),
                void* pUserCallbackData)
```

Sets up a callback to be actioned when the decoded data has been received by the API and is waiting to be collected. Setting the callback causes a tracking object to be created for this 'Data Type'. A data type is composed of a BST or a BST-BEMP pair which will be tracked and whose BST messages will be stored in a tracker buffer until all packets for that message have been received.

No windows should be created by the callback handling function, as the wait loop within the Comms thread does not process windows messages to reduce overhead; if the callback function creates a window, it could cause a system deadlock due to the unprocessed message queue.

To disable the callback for this function ID, call this 'Set' function again with a NULL function pointer, thus stopping any further callbacks.

This function can be called again to change the callback to a different one in cases where different redirections are required by the Higher Level Application (HLA).

Full callback function parameter details can be found in the 'API_Decode' header file.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_DECODE_NO_DEFINITION
  Given 'Data Type' not recognised (has no definition)

ACommsDecode_SetCallbackGroup

To set up a group of callback functions that are individually triggered by the reception of specific BST messages, use this function:

```
int ACommsDecode_SetCallbackGroup (int Handle,
                                   DecodeData_t *Datatype,
                                   int DataTypeSize,
void (*pFunc)(void*, DecodeData_t, DecodeReason_t),
                                   void* pUserCallbackData)
```

Sets up a **group** of callbacks to be actioned when the decoded data has been received by the API and is waiting to be collected. One callback to one message type. Setting the callbacks causes a tracking object to be created for each 'Data Type' requried. A data type is composed of a BST or a BST-BEM pair which will be tracked and whose BST messages will be stored in a tracker buffer until all packets for that message have been received.

Useful extension to the *ACommsDecode_SetCallback* base function when more than one callback is required to be setup for a common group of BST messages. The same result can achieved by calling the base function multiple times, however, this extended version allows multiple callbacks to be grouped.

No windows should be created by the callback handling function, as the wait loop within the Comms thread does not process windows messages to reduce overhead; if the callback function creates a window, it could cause a system deadlock due to the unprocessed message queue.

To disable the callback for a particular function ID, call the 'SetCallback' function with a NULL function pointer, thus stopping any further callbacks.

This function can be called again to change the callbacks to different ones in cases where different redirections are required by the Higher Level Application (HLA).

Full callback function parameter details can be found in the 'API_Decode' header file.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_DECODE_NO_DEFINITION
  Given 'Data Type' not recognised (has no definition)

ACommsDecode_GetTag

To get the decoded 'Tag' information section of a received BST message, use this function:

```
int ACommsDecode_GetTag (int Handle,
        sDecodeTag *pTag, DecodeData_t DataType)
```

All BST messages contain a 'Tag' section, with a subset of these messages also containing a 'Data' section.

This generic function returns a structure containing the basic tracking data from the given message 'Tag', this function should be called from within a callback handler that's actioned when a message of the data type DecodeData_t passed into this function has arrived.

As well as showing that the message was received, decoded and acted upon by the target device, the decode 'Tag' also provides essential information about the status of the processed message and the unit that responded.

The sDecodeTag structure is:

- **ResponseTime**: The command response time, or non-command message interval, in milliseconds.
- **PCTickCount**: Tick count at time of decode.
- **Error**: Message error code (Refer to ARLErrorCode.h).
- **Serial**: Unique ARL serial number of the device that returned this message.
- **ModelID**: ARL Model ID number
- **DataType**: The data/message type this tag is attached to (identifies the message contents)
- **SourceStream**: Stream that message arrived from.
- **SourceAddress**: Address that message arrived from.

This function can be used for all message data types, and is most useful when the received message only has a 'Tag' section (no 'Data' section). This is typically when the command response simply indicates that a command has been received and acted upon by the target device:

Command:
  ACommsCommand_CommitToEEPROM
  ACommsCommand_CommitToFlash
  ACommsCommand_Reboot
  ACommsCommand_ReInitMainApp
  ACommsCommand_ActivatePGNEnableLists
  ACommsCommand_SetDefaultPGNEnableList
  ACommsCommand_ClearPGNList
  ACommsCommand_ClearRxPGNList
  ACommsCommand_ClearTxPGNList

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

ACommsDecode_GetHardwareInfo

To decode the 'Hardware Information' response message that was received from a device, use this function:

```
int ACommsDecode_GetHardwareInfo (int Handle,
                                  sDecodeHardwareInfo *pHardInfo)
```

When a callback occurs and the enumerated data type is 'ddHardwareInfo', this function can be called to obtain the easy to use 'Hardware Information' data structure that has been decoded from the received message.

See the *ACommsCommand_GetHardwareInfo* command description for further details.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetHardwareInfo
DataType : ddHardwareInfo

ACommsDecode_GetOperatingMode

To decode the 'Operating Mode' response message that was received from a device, use this function:

```
int ACommsDecode_GetOperatingMode (int Handle,
                  sDecodeOperatingMode *pOperatingMode)
```

When a callback occurs and the enumerated data type is 'ddOperatingMode', this function can be called to obtain the easy to use 'Operating Mode' data structure that has been decoded from the received message.

See the *ACommsCommand_GetOperatingMode* command description for further details.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetOperatingMode
Command : ACommsCommand_SetOperatingMode
DataType : ddOperatingMode

ACommsDecode_GetHardwareBaudCodes

To decode the 'Hardware Baud codes' response message that was received from a device, use this function:

```
int ACommsDecode_GetHardwareBaudCodes
  (int Handle, sDecodePortBaudCodes *pBaudCodes)
```

When a callback occurs and the enumerated data type is 'ddHardwareBaudCfg', this function can be called to obtain the easy to use 'Baud codes' data structure decoded from the received message. The contents of the structure indicate the current baud rates actually being used by the target device to communicate, and are not necessarily the same as the baud rates actually stored in the targets EEPROM.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetHardwareBaudCodes
Command : ACommsCommand_SetHardwaretBaudCodes
DataType : ddHardwareBaudCfg

ACommsDecode_GetPortBaudCodes

To decode the 'Port Baud codes' response message that was received from a device, use this function:

```
int ACommsDecode_GetPortBaudCodes (int Handle,
                  sDecodePortBaudCodes *pBaudCodes)
```

When a callback occurs and the enumerated data type is 'ddPortBaudCfg', this function can be called to obtain the easy to use 'Baud codes' data structure that has been decoded from the received message. The contents of the structure indicate the baud rates stored (in EEPROM) inside the target device. The stored baud rates will be used on all future device initialisations and are not necessarily the same as the current Hardware baud rates.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetPortBaudCodes
Command : ACommsCommand_SetPortBaudCodes
DataType : ddPortBaudCfg

ACommsDecode_GetPortPCodes

To decode the 'Port P-codes' response message that was received from a device, use this function:

```
int ACommsDecode_GetPortPCodes (int Handle,
                                sDecodeArray_u8 *pPCodes)
```

When a callback occurs and the enumerated data type is 'ddPortPCodeCfg', this function can be called to obtain the easy to use 'Port P-codes' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetPortPCodes
Command : ACommsCommand_SetPortPCodes
DataType : ddPortPCodeCfg

ACommsDecode_GetTotalTime

To decode the 'Total Operating Time' response message that was received from a device, use this function:

```
int ACommsDecode_GetTotalTime (int Handle,
                               sDecodeTotalTime *pTotalTime)
```

When a callback occurs and the enumerated data type is 'ddTotalTime', this function can be called to obtain the easy to use 'Total Operating Time' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetTotalTime
Command : ACommsCommand_SetTotalTime
DataType : ddTotalTime

ACommsDecode_GetPortDupDelete

To decode the 'Port Duplicate Delete' response message that was received from a device, use this function:

```
int ACommsDecode_GetPortDupDelete (int Handle,
                                   sDecodeArray_u8 *pDupDelete)
```

When a callback occurs and the enumerated data type is 'ddPortDupDelete', this function can be called to obtain the easy to use 'Port Duplicate Delete' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid

Command : ACommsCommand_GetPortDupDelete
Command : ACommsCommand_SetPortDupDelete
DataType : ddPortDupDelete

ACommsDecode_GetProductInfoN2K

To decode the 'NMEA 2000 Product Information' response message that was received from the target device, use this function:

```
int ACommsDecode_GetProductInfoN2K (int Handle,
                                    sDecodeProductInfoN2K *pProductInfoN2K)
```

When a callback occurs and the enumerated data type is 'ddProductInfoN2K', this function can be called to obtain the easy to use 'NMEA 2000 Product Information' data structure decoded from the received message.

Refer to the *ACommsCommand_GetProductInfoN2K* command description for further details.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetProductInfoN2K
DataType : ddProductInfoN2K

ACommsDecode_GetCanConfig

To decode the 'CAN Configuration' response message that was received from the target device, use this function:

```
int ACommsDecode_GetCanConfig (int Handle,
                               sDecodeCanConfig *pCanConfig)
```

When a callback occurs and the enumerated data type is 'ddCanConfig', this function can be called to obtain the easy to use 'CAN Configuration' data structure that has been decoded from the received message.

Refer to the *ACommsCommand_GetCANConfig* command description for further details.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetCanConfig
Command : ACommsCommand_SetCanConfig
DataType : ddCanConfig

ACommsDecode_GetCanInfoField1-3

To decode a 'CAN Information Field 1-3' response message received from a device, use this function:

```
int ACommsDecode_GetCanInfoField1-3 (int Handle,
                                     sDecodeCanInfoField *pCanInfoField,
                                     DecodeData_t DataType)
```

When a callback occurs and the enumerated data type is either 'ddCanInfoField1', 'ddCanInfoField2' or 'ddCanInfoField3', this function can be called to obtain the easy to use 'CAN Information Field 1-3' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES10_BST_INVALID_STRING_LEN
  String length exceeded the allowable size

Command : ACommsCommand_GetCanInfoField1-3
Command : ACommsCommand_SetCanInfoField1-3
DataType : ddCanInfoField1-3

ACommsDecode_GetRxPGN

To decode the 'Enable / Disable Rx PGN' response message that was received from the target device, use this function:

```
int ACommsDecode_GetRxPGN (int Handle,
                           sDecodeRxPGN *pDecodeRxPGN)
```

When a callback occurs and the enumerated data type is 'ddEnableRxPGN', this function can be called to obtain the easy to use 'Enable / Disable Rx PGN' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetRxPGN
Command : ACommsCommand_SetRxPGN
Command : ACommsCommand_SetRxPGNEx
DataType : ddEnableRxPGN

ACommsDecode_GetTxPGN

To decode the 'Enable / Disable Tx PGN' response message that was received from the target device, use this function:

```
int ACommsDecode_GetTxPGN (int Handle,
                           sDecodeTxPGN *pDecodeTxPGN)
```

When a callback occurs and the enumerated data type is 'ddEnableTxPGN', this function can be called to obtain the easy to use 'Enable / Disable Tx PGN' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetTxPGN
Command : ACommsCommand_SetTxPGN
Command : ACommsCommand_SetTxPGNEx
DataType : ddEnableTxPGN

ACommsDecode_GetRxPGNList

To decode the 'Rx PGN Enable List' response message that was received from the device, use this function:

```
int ACommsDecode_GetRxPGNList (int Handle,
                sDecodeRxPGNList *pDecodeRxPGNList)
```

When a callback occurs and the enumerated data type is 'ddRxPGNEnableList', this function can be called to obtain the easy to use 'Rx PGN Enable List' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetRxPGNList
DataType : ddRxPGNEnableList

ACommsDecode_GetTxPGNList

To decode the 'Tx PGN Enable List' response message that was received from the device, use this function:

```
int ACommsDecode_GetTxPGNList (int Handle,
                sDecodeTxPGNList *pDecodeTxPGNList)
```

When a callback occurs and the enumerated data type is 'ddTxPGNEnableList', this function can be called to obtain the easy to use 'Tx PGN Enable List' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetTxPGNList
DataType : ddTxPGNEnableList

ACommsDecode_GetParamsPGNEnableLists

To decode the 'Parameters of the PGN Enable Lists' response message that was received from the target device, use this function:

```
int ACommsDecode_GetParamsPGNEnableLists
      (int Handle,
       sDecodePGNEnableListStatus* pPGNListStatus)
```

When a callback occurs and the enumerated data type is 'ddParamsPGNEnableLists', this function can be called to obtain the easy to use 'Parameters of the PGN Enable Lists' data structure that has been decoded from the received message.

For a description of the information in this message, refer to the *ACommsCommand_GetParamsPGNEnableLists* command description.

Decoding this response message can be very useful to the Higher Level Application (HLA) - as this details the number of Rx and Tx Real and Virtual PGN 'slots' that are currently used and the maximum number that are available of each type. In this way you will always know exactly how many PGN 'slots' are in use and how many you have left available to you.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : ACommsCommand_GetParamsPGNEnableLists
DataType : ddParamsPGNEnableLists

ACommsDecode_GetStartupStatus

To decode the 'Startup Status' response message that was received from the target device, use this function:

```
int ACommsDecode_GetStartupStatus (int Handle,
                    sDecodeStartupStatus* pStartupStatus)
```

When a callback occurs and the enumerated data type is 'ddStartupStatus', this function can be called to obtain the easy to use 'Startup Status' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : None - device sends once at startup
DataType : ddStartupStatus

ACommsDecode_GetSystemStatus

To decode the 'System Status' response message that was received from the target device, use this function:

```
int ACommsDecode_GetSystemStatus (int Handle,
                    sDecodeSystemStatus *pSystemStatus)
```

When a callback occurs and the enumerated data type is 'ddSystemStatus', this function can be called to obtain the easy to use 'System Status' data structure that has been decoded from the received message.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : None - device sends once per second
(if P-codes' enabled for that port)
DataType : ddSystemStatus

ACommsDecode_GetDbgTimeProfiler

To decode the 'Debug Time Profiler' response message that was received from the device, use this function:

```
int ACommsDecode_GetDbgTimeProfiler (int Handle,
                    sDbgTimeProfiler* pTimeProfile)
```

When a callback occurs and the enumerated data type is 'ddDebugTimeProfiler', this function can be called to obtain the easy to use 'Debug Time Profiler' data structure that has been decoded from the received message.

Under normal operations, this message will not be transmitted by the target device. This message will only ever be made available as part of a 'debug' firmware version for use in 'beta' test operations.

If an error was detected by the API, one of these ARL error codes will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_COMMAND_TRACKER
  Message tracker not found - check Tag pointer valid
ES11_COMMS_DECODE_BAD_DATA
  Unexpected extra data found in response message

Command : None - device sends once per second
(if P-codes' enabled for that port AND
using a special 'debug' firmware)
DataType : ddDebugTimeProfiler

API_NMEA0183

This module forms the low-level NMEA 0183 Protocol interface of the Actisense Comms Library.

These functions would typically only be of interest when using the API in conjunction with the Actisense NGW-1 (NMEA 2000 to NMEA 0183) device. However, they will work just as well with any NMEA 0183 device.

Care must be taken to ensure that the Comms port has been opened using the same Baud rate of the NMEA 0183 device that is connect to that port. Typically this will be 4800 Baud for standard devices, or 38400 Baud for 'High Speed' (HS) devices.

All API_NMEA0183 functions (except for 'Write') are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsN183_Write

To send (write) an NMEA 0183 message to the NMEA 0183 transmit queue of the given Comms port handle, use this function:

```
int ACommsN183_Write (int Handle, sN183Msg *msg)
```

Adds the given NMEA 0183 message to the transmit queue of the given Comms port handle. The queue handler will transfer all new messages to the Windows COM port at the next opportunity.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles
ES11_BUFFER_OVERFLOW
Given handle's queue is full - write operation had to remove the oldest queued message first

ACommsN183_Read

To get (read) the next received NMEA 0183 message from the NMEA 0183 receive queue of the given Comms port handle, use this function:

```
int ACommsN183_Read  (int Handle, sN183Msg *msg)
```

Retrieves the next NMEA 0183 message from the receive queue of the given Comms port handle. Typically this function is called successively until the receive queue has been emptied.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles
ES11_BUFFER_UNDERFLOW
Given handle's queue is empty - read operation has failed to find a message

ACommsN183_GetRxQSize

To get the total number of NMEA 0183 messages (still to be read) that are waiting in the NMEA 0183 receive queue of the given Comms port handle, use this function:

```
int ACommsN183_GetRxQSize (int Handle,
                           size_t *BufferSize)
```

Returns the current size of the NMEA 0183 receive queue, which can be used to determine if any NMEA 0183 data has been received (and requires processing) if the 'data polling' method is preferred by the Higher Level Application (HLA). However, it is more normal to use the NMEA 0183 callback functionality - as this helps keep the CPU loading to a minimum.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsN183_FlushRx

To delete (flush) all unread NMEA 0183 messages from the NMEA 0183 receive queue of the given Comms port handle, use this function:

```
int ACommsN183_FlushRx (int Handle)
```

Clears the NMEA 0183 receive queue. This will cause all messages/data to be lost so this must be performed with care to prevent unwanted data loss.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsN183_FlushTx

To delete (flush) all unsent NMEA 0183 messages from the NMEA 0183 transmit queue of the given Comms port handle, use this function:

```
int ACommsN183_FlushTx (int Handle)
```

Clears the NMEA 0183 transmit queue. This will cause all messages/data to be lost so this must be performed with care to prevent unwanted data loss.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsN183_SetRxCallback

To setup an NMEA 0183 receive message type Callback handling function for the given Comms port handle, use this function:

```
int ACommsN183_SetRxCallback (int Handle,
                              void  (*pFunc)(void*), void* p)
```

Defines the callback function that will be called when an NMEA 0183 message has been successfully received and is waiting in the NMEA 0183 receive queue. This callback should keep reading (removing) messages from the NMEA 0183 receive queue until the queue is empty.

If messages remain in the queue when the callback function returns, the callback function will be actioned again immediately, so it is possible to only process one message per function call, but this may be less efficient.

No windows should be created by the callback handling function, as the wait loop within the Comms thread does not process windows messages to reduce overhead; if the callback function creates a window, it could cause a system deadlock due to the unprocessed message queue.

To disable the callback for this function ID, call this 'Set' function again with a NULL function pointer, thus stopping any further callbacks.

This function can be called again to change the callback to a different one in cases where different redirections are required by the Higher Level Application (HLA).

Full callback function parameter details can be found in the 'API_NMEA0183' header file.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

API_NMEA2000

This module forms the low-level NMEA 2000 Protocol interface of the Actisense Comms Library.

These functions are useful to the Higher Level Application (HLA) when it is required to send NMEA 2000 (via the local NGT) to the NMEA 2000 bus, and also when it is required to listen to various PGNs that are being sent by other NMEA 2000 devices on the NMEA 2000 bus.

Care must be taken to ensure that the Comms port has been opened using the same Baud rate as the local NGT NMEA 2000 PC interface, which has a default Baud rate of 115200.

All API_NMEA2000 functions (except for 'Write') are [INTERNAL] to the PC and do not generate any external communication messages to the attached ARL device.

ACommsN2K_Write

To send (write) an NMEA 2000 message to the NMEA 2000 transmit queue of the given Comms port handle, use this function:

```
int ACommsN2K_Write (int Handle, sN2KMsg *msg)
```

Adds the given NMEA 2000 message to the transmit queue of the given Comms port handle. The queue handler will transfer all new messages to the Windows COM port at the next opportunity.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_BUFFER_OVERFLOW
  Given handle's queue is full - write operation had to remove the oldest queued message first

ACommsN2K_Read

To get (read) the next received NMEA 2000 message from the NMEA 2000 receive queue of the given Comms port handle, use this function:

```
int ACommsN2K_Read  (int Handle, sN2KMsg *msg)
```

Retrieves the next NMEA 2000 message from the receive queue of the given Comms port handle. Typically this function is called successively until the receive queue has been emptied.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles
ES11_BUFFER_UNDERFLOW
  Given handle's queue is empty - read operation has failed to find a message

ACommsN2K_GetRxQSize

To get the total number of NMEA 2000 messages (still to be read) that are waiting in the NMEA 2000 receive queue of the given Comms port handle, use this function:

```
int ACommsN2K_GetRxQSize (int Handle,
                          size_t *BufferSize)
```

Returns the current size of the NMEA 2000 receive queue, which can be used to determine if any NMEA 2000 data has been received (and requires processing) if the 'data polling' method is preferred by the Higher Level Application (HLA). However, it is more normal to use the NMEA 2000 callback functionality - as this helps keep the CPU loading to a minimum.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
  Given handle does not match any active handles

ACommsN2K_FlushRx

To delete (flush) all unread NMEA 2000 messages from the NMEA 2000 receive queue of the given Comms port handle, use this function:

```
int ACommsN2K_FlushRx (int Handle)
```

Clears the NMEA 2000 receive queue. This will cause all messages/data to be lost so this must be performed with care to prevent unwanted data loss.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsN2K_FlushTx

To delete (flush) all unsent NMEA 2000 messages from the NMEA 2000 transmit queue of the given Comms port handle, use this function:

```
int ACommsN2K_FlushTx (int Handle)
```

Clears the NMEA 2000 transmit queue. This will cause all messages/data to be lost so this must be performed with care to prevent unwanted data loss.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

ACommsN2K_SetRxCallback

To setup an NMEA 2000 receive message type Callback handling function for the given Comms port handle, use this function:

```
int ACommsN2K_SetRxCallback (int Handle,
                             void  (*pFunc)(void*), void* p)
```

Defines the callback function that will be called when an NMEA 2000 message has been successfully received and is waiting in the NMEA 2000 receive queue. This callback should keep reading (removing) messages from the NMEA 2000 receive queue until the queue is empty.

If messages remain in the queue when the callback function returns, the callback function will be actioned again immediately, so it is possible to only process one message per function call, but this may be less efficient.

No windows should be created by the callback handling function, as the wait loop within the Comms thread does not process windows messages to reduce overhead; if the callback function creates a window, it could cause a system deadlock due to the unprocessed message queue.

To disable the callback for this function ID, call this 'Set' function again with a NULL function pointer, thus stopping any further callbacks.

This function can be called again to change the callback to a different one in cases where different redirections are required by the Higher Level Application (HLA).

Full callback function parameter details can be found in the 'API_NMEA2000' header file.

If an API error occurs, this error code will be returned:

ES11_INVALID_HANDLE
Given handle does not match any active handles

Using the Actisense API

This sections details how to use the Actisense Comms API to communicate with the NGT-1 'NMEA 2000 PC Interface' and NGW 'NMEA 2000 to NMEA 0183' devices.

Reference is made to the 'ActisenseN2KDemo' Visual C++ project that is included as part of the SDK (in both source file and Windows executable form).

Initialise for each use

The normal operation of the Higher Level Application (HLA) software and the NGT is to initialise the device at startup with the full list of Rx and Tx PGNs that are required for operation at that time. **Alternatively, the new 'Receive All' mode can now be used to allow all received NMEA 2000 PGN messages to be transferred to the HLA software. Refer to command *ACommsCommand_SetOperatingMode* for full details.**

With this sequence in mind, the NGT configuration operation has been designed to be flexible and quick:

1. Get the Rx and Tx 'PGN Enable lists' from the NGT using the commands *ACommsCommand_GetRxPGNList* and *ACommsCommand_GetTxPGNList*.

2. If the PGN Enable lists are not as currently required, either delete the individual PGNs one at a time (by using the commands *ACommsCommand_SetRxPGN* or *ACommsCommand_SetTxPGN*), or delete the entire list with the single *ACommsCommand_ClearRxPGNList* or *ACommsCommand_ClearTxPGNList* commands.

The choice of which method to use is entirely up to the programmer, although it can be more logical when any more than a few PGNs need to be changed, to use the '**Clear PGN List**' commands first.

3. Add any new Rx and Tx PGNs as required to the PGN Enable lists, one command per PGN required, using the commands *ACommsCommand_SetRxPGN* or *ACommsCommand_SetTxPGN*.

4. The final operation (to make the new Enable lists operational) is to activate the PGN Enable lists (using command *ACommsCommand_ActivatePGNEnableLists*). This command re-initialises the hardware to use the new PGN Enable list settings.

Making the '**Activate**' separate from the request to change the list is important because the '**Activate**' operation causes a small pause in the NMEA 2000 data being sent to and from the PC - as the hardware receive/transmit operations must be paused temporarily to allow the necessary register changes to be performed.

For this reason, the HLA will not want to action this operation after every single PGN Enable list change.

5. Lastly, if it is required that these PGN Enable list changes should remain set after the NGT-1 loses power (unplugged from the USB port), then the '**Activate**' command must be followed with the *ACommsCommand_CommitToEEPROM* command. This command will copy / commit the new Enable lists to the nonvolatile memory inside the target device, where it can be read from at all future device startups.

However, many setups do to not require the use of the '**Commit**' command - as allowing the Enable List settings to be changed purely for the current session (RAM only), allows a quick reset to be performed to get the NGT back to the default list (held in EEPROM).

The idea of different 'functioning modes' has been proven to be very useful to software engineers using the Actisense Comms API. Each 'functioning mode' has its own group of PGNs that are required - and so its own 'Enable lists'.

By being able to very quickly swap between these different 'Enable lists', the behaviour of the NGT can be fine-tuned for each 'functioning mode', helping the software designer control/reduce the PGN message decode overheads.

This feature is particularly useful when the HLA program needs to operate in different 'guises': configuration (in the factory), or installation of the device (on the vessel) - both of which can require different PGNs from the normal day-to-day functional mode.

Rx PGN Enable list

When required to add an Rx PGN to the 'Rx PGN Enable list', the Mask value associated with the PGN will normally be kept to the 'Use Defaults' value. However, there are special situations when a different setting is useful.

The PGN and Mask are effectively combined together by the hardware to create the required response. Using the 'Water Depth' PGN as an example:

Water Depth PGN (128267)	1F50B xx	hex
Default Mask	3FFFF 00	hex

All bits in the Mask that are set to '1' indicate that a received PGN must match that bit exactly to be accepted. Any bits set to '0' indicates that the corresponding bit in the PGN is ignored (can be either '0' or '1') - shown as an 'x' above.

The above example defines that the unique PGN of 1F50B (hex) must be received for it to pass the PGN & Mask filter. Any other PGNs will fail - as their bits will differ in the 'active' bits set in the Mask. The Mask does however allow this PGN to be received from any 'Source Address' (indicated by the least significant 8 bits being '0').

If the Mask was instead set to 3FF00 00 (hex):

Water Depth PGN (128267)	1F5xx xx	hex
Non-default Mask	3FF00 00	hex

This allows (in theory) all PGNs from 1F500 to 1F5FF to be accepted, however, this is cannot be allowed in practice because this PGN-Mask combination would then allow both single-packet and fast-packet messages in through the same filter - which is not possible (and so is blocked by the NGT firmware).

This Mask (3FF0000 hex) can however be used for the two blocks of 256 Proprietary PGNs: 65280 to 65535 (0FF00 to 0FFFF - all single-packet PGNs), and 130816 to 131071 (1FF00 to 1FFFF - all fast-packet PGNs).

It is acceptable to set up a single PGN-Mask combination to accept / receive all 256 Proprietary PGNs of a block - as all 256 messages are of the same message type. This is the normal process when a large group of proprietary PGNs are required:

All Proprietary PGNs (65280 to 65535)	0FF00 00 hex
Mask	3FF00 00 hex
All Proprietary PGNs (130816 to 131071)	1FF00 00 hex
Mask	3FF00 00 hex

The 7 PGNs defined between 59392 to 61184 and 126208 to 126720 are special 'Core Control' PGNs:

- ISO Acknowledge PGN (59392/0E800)
- ISO Request PGN (59904/0EA00)
- ISO Address Claim PGN (60928 / 0EE00)
- N2K Proprietary Manufacturer PGN (61184/0EF00)
- N2K Rqst/Ack/Cmd Group PGN (126208/01ED00)
- N2K Tx & Rx PGN list PGN (126464/01EE00)
- N2K Proprietary Manufacturer PGN (126720/01EF00)

These PGNs are always received and processed by the NGT. When they are added to the 'Rx PGN Enable list', the 'copy to the PC' operation is enabled. In this way, these 'Core Control' PGNs are classed as Rx 'Virtual' PGN's (as they are not used to set up the 'Real' hardware). This means they are effectively free - as they do not use up one of the 28 PGN 'slots'.

In summary: If the total number of required Rx PGNs required at any one time fits in to the maximum of 28 'slots' (ignoring the 7 core PGNs that are always available), then it is normal to set up each Proprietary PGN separately with its own PGN-Mask combination (using the 'Use Default' mask option).

However, if more than 28 PGN 'slots' are required (at any one time), again ignoring the 7 core PGNs that are always available, the Proprietary PGNs can be grouped together in to the two groups detailed above using the 'Match PDU Format' mask option (that only matches on the PDU Format (and page) bits of the message.

In this way, all 512 Proprietary PGNs only occupy 2 PGN 'slots' in the hardware, leaving 26 remaining 'slots' for the standard PGNs.

The 2 additional 'Core Control' PGNs are a special case and do not exist outside of the NMEA 2000 network - they are used purely to transfer other PGN messages using the ISO Transport Protocol. These two PGNs are handled transparently by the NGT-1 and can be ignored:

- ISO Transport Protocol (Data) PGN (60160/0EB00)
- ISO Transport Protocol (Man) PGN (60416/0EC00)

The NGT-1 comes with a factory default list of Rx and Tx PGNs - its '**PGN Enable Lists**'. These lists determine what PGNs are required to be transferred from the NMEA 2000 bus to the PC (Rx) and what PGNs are required to be sent from the PC (Tx).

Normally the HLA (PC) software will adjust these Enable lists to suit its current PGN requirements, and once set, can also be remembered for the next session using the *ACommsCommand_CommitToEEPROM* command.

Tx PGN timings

Taking the PGN 127488 (Engine Parameters, rapid) as an example, this has a default 'Transmit (Tx) rate / interval' of 100 milliseconds.

If this rate / interval has either been set using the value '100' or by using the 'Use Default' value, when added to the '**Tx PGN Enable list**', the NGT will limit this PGN to a **maximum** transmit speed of *once per 100 milliseconds*. In doing so, the NGT satisfies the NMEA 2000 specification of maintaining the defined Tx rate for each PGN.

The 'Timeout' value for a transmit PGN has now been disabled and can be ignored. This is due to a change in the NMEA 2000 specification regarding the **minimum** PGN transmit speed. The new requirement states that there should not be a minimum transmit speed, hence there is no longer a need for the 'Timeout' value.

Therefore, this new rule states that if no new PGN (127488) message is received from the PC (by the NGT) within the required 100 milliseconds Tx rate interval, the NGT will no longer generate a copy message, and the Tx rate will be allowed to drop below the defined Tx rate.

Conversely, if the PC starts sending the PGN (127488) to the NGT-1 faster than the defined 100 millisecond interval, the gateway must limit the Tx rate of this PGN on the NMEA 2000 bus to the defined interval of 100 milliseconds. **A Tx PGN will never be set by the NGT gateway faster than the stipulated Tx rate.**

To help visualise this behaviour, it can be useful to temporarily add an 'ID' value to a PGN message data field sent to the NGT. For example, if the "Engine Speed" value was replaced by a simulated speed that increments by a single value each time it was transmitted from the PC, each message will be unique and traceable when viewed on the NMEA 2000 bus.

In this way, the NGT Tx rate interval behaviour will become visible and understandable to the software developer.

API & Device Error Codes

There is a major difference between an API function call that changes something in the API, and an API function call that requests information from or changes in the NGT-1/NGW-1 (or other Actisense device):

1. **API function calls that changes something in the API** can immediately return a meaningful error code because it can immediately determine if something is wrong with the request itself. This is true of the *ACommsCreate* and *ACommsOpen* functions: they are changing settings in the API itself, so they can immediately flag up an error.

2. **API function calls that requests information from or changes in the device** cannot always return immediately with a meaningful error code because it cannot always know if the request is valid - only the device can do that. If an invalid handle was passed to the function, then that is within its power to flag up an error immediately, however, it cannot understand if the request message contents (to be sent to the device) are valid - that is the job of the device itself. All *ACommsCommand* functions fall in to this category.

Therefore, all *ACommsCommand* functions may well return a "No error" status - this informs the Higher Level Application (HLA) that the request was **sent** successfully, it **does not** however indicate if the request that was sent was actually **valid**.

The answer to the second question is supplied as a response message from the device itself and it is that response message that must be read in order to understand if there was an error or not with the API Command request.

Reset/Re-initialisation sources

There are three ways a device can be re-initialised / reset. A re-initialisation process will occur for any one of these:

1. **A power reset of the device** (unplugging an ISO gateway variant from the NMEA 2000 bus, or unplugging a USB gateway variant from the PC).

2. Sending the API *ACommsCommand_Reboot* or *ACommsCommand_ReInitMainApp* commands.

3. Sending a command that must force a re-initialisation to complete the request. Currently this only applies to: *ACommsCommand_SetOperatingMode* and *ACommsCommand_SetDefaultPGNEnableList*

Therefore, simply closing and reopening the Comm port to a new Baud rate will not reset the device at all.

Application thread restrictions

There are no threading restrictions placed on the programmer, beyond those of standard safe coding practices. All API communication commands are protected by a critical section, so that only one thread can gain access to important resources at any one time. The second thread will wait for the first to complete its operation before being given access.

For example, in theory any number of threads could be used to call *ACommsN2K_Write* and *ACommsN2K_Read*, there is no need to limit this to a single thread, however, normal practices will normally limit this to a maximum of two threads (1 Write and 1 Read).

Standard safe programming practices do however dictate that the same thread that calls *ACommsOpen*, should also call *ACommsClose*, **but** only after all open threads that are linked to the port about to be closed, are first destroyed by calling *ACommsDecode_SetCallback* with a NULL pointer for that callback function.

Application-API thread efficiency

The API uses the overlapped method in the COM port Tx and Rx threads, so in theory there shouldn't be too much overhead - as the Comms threads suspend themselves until data arrives to be processed.

Testing this on a four year old laptop (Windows XP, Pentium-M, 1900 MHz), the Actisense Comms library only used approximately 1% of the CPU utilisation, even with a large data load flowing through the Comms port (115200 Baud at 40+% load).

Automatically detecting an installed Actisense device's port

Detecting an Actisense device's port is really easy:

1. The API function *ACommsEnumerateSerialPorts* returns a list of the enumerated serial ports available to the system.

2. The 'really useful' name of each port is obtained by calling the *ACommsEnumerateSerialPortsGetName* API function.

3. Performing a quick search through the 'port descriptions name list' for the "Actisense NGT" or other device text will result in the port description strings and port numbers of each and every Actisense device.

'Receive All Transfer' Operating Mode

A new operating mode that offers to transfer all NMEA 2000 messages to the HLA software. The HLA can easily send the single *ACommsCommand_SetOperatingMode* command to enable this 'Receive All Transfer' operating mode, which is perfect for diagnostic or logging software that requires access to all NMEA 2000 messages currently on the network.

This operating mode change is a session only feature so that the NGT-1 will reset back to using the built-in Rx PGN List when a power or command reset occurs. The Tx PGN List is unaffected by this new mode, and operates exactly as it did before.

It is important to note that use of the 'Receive All Transfer' operating mode is not recommended when the total number of PGNs that the HLA software requires access to will fit in to the Rx PGN List (28 Real and 7 Virtual PGNs).

Unnecessary use of the 'Receive All Transfer' mode will place an extra load on both the NGT-1 microcontroller and more importantly, on the HLA software, to process the greatly increased volume of NMEA 2000 PGN messages.

Proprietary 'P-code' messages

The 'Rx & Tx PGN Enable lists' are set up to match the API user requirements to transfer data from each input to its opposite output on the NGT. That is: Serial -> NMEA 2000, and NMEA 2000 -> Serial.

However, Manufacturer Proprietary messages (such as the '**System Status**' message sent once per second by an NGT / NGW) are 'internally generated' messages: they are not triggered from any data coming in to the NGT / NGW, and so are not controlled (enabled and disabled) by changing the '**Tx PGN Enable list**'.

On the NMEA 2000 port, all the Actisense Proprietary messages from an NGT or NGW via the NMEA 2000 bus use the PGN 126720.

The serial port on the NGT or NGW can also be configured to send the same Actisense Proprietary messages using the either the BST or NMEA 0183 Protocol respectively.

These Proprietary-Codes can be turned on and off via the *ACommsCommand_SetPortPCodes* API command:

The first message byte relates to the CAN (NMEA 2000) port, and the second to the Serial / USB port.

The default is to have the serial port P-codes ON, and the CAN (NMEA 2000) OFF: 0 and 1.

Setting up Callbacks

A full example of the suggested way to create and setup the necessary callback functions required by the API is given in the 'ActisenseN2KDemo' source code.

Below is a short description of the procedure used to create the 'ActisenseN2KDemo' source code using the Visual C++ environment:

1. Create a Class to handle the data

'N2KViewForm' module handles the reception and corresponding display of the NMEA 2000 data:

```
class CN2KViewForm : public CFormView
```

The header file details all the 'protected' and 'public' variables and functions.

2. Create a static callback function of the Class

The callback function that is called by the Actisense Comms dll must be a static member of the class.

It can be useful to prefix all these callback functions with the name "Callback" so that they can be easily identified as such:

```
static void CallbackFuncN2K (void* p);
```

The only allowed function parameter of the callback function is a void pointer, shown as "(void* p)":

```
void CallbackFuncN2K (void* p)
```

3. Create the content of the callback function

The void pointer is used to pass in the system "this" variable pointer so the static callback function can gain access to the non-static members of the class. This operation requires the re-casting of the void pointer back to the Class pointer:

```
CN2KViewForm* pView =
                    static_cast<CN2KViewForm*>(p)
```

To retrieve all unread NMEA 2000 messages from the receive buffer, a do-while loop is created to keep calling the 'API Read' function until the queue is empty:

```
do { } while (size && !Error);
```

Call the API to retrieve the next NMEA 2000 message:

```
Error = ACommsN2K_Read (pView->ACommsHandle,
                        &msg)
```

Store the read message in the Classes storage:

```
pView->N2KStore.AddMessage(msg)
```

Ask the API how many unread messages are still in the NMEA 2000 receive queue (so the do-while loop can be aborted if none remain):

```
ACommsN2K_GetRxQSize (pView->ACommsHandle,
                      &size)
```

Finally, cause a refresh / redraw of the GUI to allow the newly received data to be displayed to the user:

```
int WM_REDRAW_LIST_ITEMS =
   RegisterWindowMessage ("REDRAW_LIST_ITEMS")

BOOL Result = pView->PostMessage (WM_REDRAW_
                                  LIST_ITEMS, 0, 0)
```

4. Create a 'Set/Clear callback' function of the Class

To allow quick 'setting' and 'clearing' of the callback function, create a simple function to encapsulate this operation. This has the major benefit of separating the module that 'sets' and 'clears' the callback from the class module that knows what callback function to setup of for that data type:

```
void SetOrClearDataCallbacks (bool Set)
```

If requested to setup a callback for this data type, call the 'SetRxCallback' function with the Class known callback function:

```
ACommsN2K_SetRxCallback (ACommsHandle,
                         CallbackFuncN2K, this)
```

Alternatively if the callback is required to be disabled, call the 'SetRxCallback with a NULL pointer (the void pointer is ignored by the 'Set' function under this condition, and so its value is not important):

```
ACommsN2K_SetRxCallback (ACommsHandle,
                         NULL, NULL)
```

Changing the device's baud rate

Every Actisense device connected via its serial or USB port talks to the PC initially using its 'Port baud rate'. This default baud rate is **stored** permanently in the devices EEPROM and is read during start-up.

The EEPROM setting for 'Port baud rate' can be set using the *ACommsCommand_SetPortBaudCodes* command, however, this change will not be actioned / used until the device is *reset*.

The baud rate every Actisense device is currently **using** on its serial port is called the 'Hardware baud rate'. This rate can be changed without affecting the 'Port baud rate' (stored in EEPROM that the device will adopt when a *reset* occurs) by using the *ACommsCommand_SetHardwareBaudCodes* command.

As an example, the Actisense NGT-1 will normally be set up to work at 115200 or 230400 baud. This baud rate is fast enough to communicate efficiently and quickly with the host PC. However, some products, such as an Actisense NGW-1 gateway, or NDC-4 multiplexer may require to talk at 4800 baud. This communication rate is the correct rate to talk to NMEA 0183 products, but slows down communication with PC (configuration) software.

To improve the speed of configuration, all Actisense devices support the 'Hardware baud rate' method of allowing the PC software to temporarily change a devices communication speed until the end of that 'session'.

To change the devices 'Hardware baud rate':

1. Use *ACommsCommand_SetHardwareBaudCodes* command to send the baud rate of each port in the attached device. An NGT-1 or NGW-1 gateway has **two** communication ports inside: the CAN bus / NMEA 2000 port, and the Serial / NMEA 0183 port.

   Therefore, **both** baud codes must be sent, and in the correct order: 'CAN' first, 'Serial' second. Any other number of baud codes will result in the relevant error code being returned.

2. The device will send back its response message at the original baud rate. This allows the PC software to understand if the baud rate change was successful.

   The PC software must check the error code returned in the command respond message. If no error was indicated, the device will automatically start using the new baud rate, and the PC software must change it's own baud rate to maintain communications.

API source code (C, C++, C#)?

The Actisense Comms API source code is written in **Visual C++** (2008), however, the programming interface (API) itself is written in **C** to increase the compatibility with software vendor's products. The Actisense Comms API is currently provided as a dll file that allows easy and quick inclusion in to existing software.

However, under a 'Non-Disclosure Agreement' (NDA) Actisense can now offer access to a low-level alternative to the dll - primarily for non-PC based systems.

There are a good number of companies currently using the Actisense Comms dll without problem, and it is proving to be a reliable and quick solution to implement and more importantly, maintain.

ActisenseComms dll C# 'wrapper'

The new ActisenseComms .Net C# wrapper Test Project has recently been released on the Actisense website. This source code example project gives the .Net C# software developer all that they need to easily access the Actisense Comms dll in their C# environment.

The working Visual Studio test project is a perfect starting point that allows very quick integration of the Actisense Comms API in to an existing software package, allowing full debug to help the developer learn how it's achieved.

Please contact Actisense for details.

NMEA 2000 PGN options

Any NMEA 2000 PGN that has been declared in the NMEA 2000 specification, which includes all of the Proprietary PGN's can be sent and/or received by the NGT-1.

The block of 256 single-packet (non-addressable) proprietary PGNs (0x0FF00 to 0x0FFFF), the block of 256 fast-packet (non-addressable) proprietary PGNs (0x1FF00 and 0x1FFFF), plus the two addressable proprietary PGNs (0x0EF00 (single-packet) and 0x1EF00 (fast-packet)) are able to be sent and received using the NGT-1.

The current NMEA 2000 library supports all PGNs in the v1.300 revision of the specification.

NMEA 2000 certification

The Actisense NGT-1 product is NMEA 2000 certified, furthermore, it will be certified as an '**Intelligent Gateway**' / '**Third Party Gateway**' by the NMEA.

What this offers to the software developer is the ability to also get their software NMEA 2000 certified (when used with the NGT-1) at the highly reduced cost of $100 (plus a nominal cost for Actisense to perform the required certification tests and send them to the NMEA). This new method allows smaller developers to get in on the NMEA 2000 action easier and much cheaper.

The cost of performing this certification yourself, including registering with the NMEA, buying the complete NMEA 2000 documentation, the NMEA 2000 certification tool, a Manufacturer code and a Product code, plus completing the certification process totals a massive $9150!

'Intelligent Gateway' and 'Third Party Gateway' (TPG)

The NGT-1 performs all NMEA 2000 network operations and will not allow illegal operations to be performed on the NMEA 2000 bus. It sanitises all requests made of it by the PC so that only legal messages are sent to the NMEA 2000 bus.

This ability is why the PC software can be NMEA 2000 certified for use with the NGT-1 at such a reduced cost - because the NMEA understands that the PC software cannot impair the NMEA 2000 bus.

The NGT-1 will be certified as a "Third Party Gateway" in 2010, which will allow each PC software program that uses the NGT-1 as its NMEA 2000 interface to subsequently be NMEA 2000 certified by Actisense and the NMEA.

NMEA 2000 Address Claiming

The NGT-1 takes care of all NMEA 2000 network issues such as '**Address Claiming**'. That is the only way that the '**Intelligent Gateway**' can hope to work correctly - as allowing the PC software to perform this operation will incur response delays and perhaps incorrect operation if not handled correctly. This removes this requirement from the HLA and as a result, the software developer need not even know how addresses are claimed or even purchase the '**Address Claim**' NMEA 2000 document.

Converting NMEA 2000 to NMEA 0183

The NGT-1 does not currently perform any conversion on the NMEA 2000 messages received, other than to transfer them to the PC and the software running on it. This results in no loss of data resolution that can occur when converting between NMEA 2000 and NMEA 0183.

The requirement for compatibility with legacy NMEA 0183 software via USB is handled perfectly by the Actisense **NGW-1-USB** product. However, Actisense has considered this option, and does plan to add this extra feature in a future dll upgrade, which would remove the need for the current **NGW-1-USB**.

Full (2500 volts) galvanic isolation

The NMEA 2000 specification requires a fully isolated interface to the NMEA 2000 bus. By using the NMEA 2000 network power and opto-isolation, the NGT-1 maintains the integrity of the NMEA 2000 bus and meets the specification requirements by offering 2500 volts of galvanic isolation in both directions.

Cost effective interface

The NGT-1 has been designed with quality components to maintain our reputation for high quality products.

However, an eye was also kept on the end customer cost, to prevent it from becoming too high. Actisense is conscious of making NMEA 2000 attractive to the end customer and plans to reduce the cost of the NGT-1 further when volume allows.

Company Information

Active Research Limited
5, Wessex Trade Centre
Ringwood Road
Poole
Dorset
UK
BH12 3PF

Telephone: 01202 746682 (International : +44 1202 746682)
Fax: 01202 746683 (International : +44 1202 746683)

Actisense on the Web: For advice, support and product details

E-mail: support@actisense.com
Website: www.actisense.com

“Actisense” is a registered trademark of Active Research Limited.