Document Overview

This document describes how to define a CAN network within the calvos system, how to generate the corresponding source code, integrate it and use it in the user's target project.

This document applies for calvos version 0.0.2.

Preparing Calvos Engine

Installing Calvos

From PyPi

Calvos engine python package is indexed at PyPi and, therefore, can be installed with a `pip` command from python:

`python -m pip install calvos`

Alternatively, it can be installed "manually" by downloading the code and installing it locally. See section below.

Manual Installation

Here we'll exemplify the installation of a calvos under MS Windows.

1. Ensure python v3.7 or greater is installed
2. Create a temporal folder *c:\calvos* (you can use any drive letter and folder name as desired).
3. Get source code from https://github.com/calcore-io/calvos
4. Unzip (if zip was downloaded) and copy the downloaded files to the temporal folder *c:\calvos*.

   Folder structure should look as follows (not all contents are shown, just an example)...

   ```
   c:\calvos
   +-- calvos-engine
   |   +-- calvos
   |       +-- comgen
   |       +-- common
   |       +-- doc
   |       +-- __init__.py
   |       +-- __main__.py
   |   +-- .project
   |   +-- .pydevproject
   |   +-- README.md
   +-- doc
   +-- project_example
   +-- LICENSE
   +-- README.md
   ```

5. Get into the *calvos/calvos-engine* folder

   `c:\>cd calvos\calvos-engine`

6. Run python setup.py install command:

`c:\calvos\calvos-engine>python setup.py install`

7. Verify calvos was installed properly by getting the calvos version.

   `c:\>python -m calvos -v`

   Version should like like: `calvos v0.0.2`

Command Line Arguments

Calvos accepts the following command-line arguments:

Argument	Usage
-h, --help	show help message and exit.
-d DEMO, --demo DEMO	Will provide an example calvos project with user input templates in the given DEMO path. No project will be processed if this argument is provided.
-p PROJECT, --project PROJECT	Required (if -d was not provided). Full path with file name of the calvos project to be processed.
-l LOG_LEVEL, --log LOG_LEVEL	Optional. LOG_LEVEL: 0 - Debug, 1 - Info, 2 - Warning, 3 - Error. Default is 1 - Info.
-e EXPORT, --export EXPORT	Optional. Generated C-code will be exported (copied) into the provided EXPORT path.
-b BACKUP, --backup BACKUP	Optional. Backups of the overwritten C-code during an export operation will be placed in the provided BACKUP path. This is only used if -e argument was provided.
-V, --version	show program's version number and exit
-v, --ver	show program's version number and exit

Demonstration Project

As a starting point, a demonstration project can be generated by the calvos system and exported to a given path. This is achieved by using the line argument `-d` as in the example below:

`python -m calvos -d c:\demo_project`

This will create the following files:

- **calvos_project.xml**: XML project file

  location: *c:\demo_project\calvos_project.xml*

- **template - CAN Network Definition.ods**: Template for CAN Network definition

  location: *c:\demo_project\usr_in\template - CAN Network Definition.ods*

- **log.log**: Log file

  location: *c:\demo_project\log.log*

It is useful to create this demo project since by doing so, also the **template** for the CAN network definition "template - CAN Network Definition.ods" is exported. In the guide below, this exported template is used.

CAN Network definition

The CAN network to generate needs to be defined first using the "template - CAN Network Definition.ods" template. Currently only ODS format is supported, however, it is possible to edit the template in MS Excel and then save it as ODS.

Different tabs within the template need to be filled-up in order to fully define the network. A file based on this template shall be created per each required network. These files defining the networks can be freely named.

Calvos supports a project with multiple networks and each network with multiple nodes. Provided that network names and not the same and that each node gets allocate a CAN peripheral in the target MCU.

Network definitions

General network definitions need to be specified in tab "Network_and_Nodes".

- **Network Name:** a name for the given network. This is to be used for documentation purposes only (the name is not used for code generation). For example: "CAN Test Network".
- **Network ID:** This will be used to uniquely identify the network. Value shall comply with C-identifier syntax. This value is to be used extensively across the generated source code in names for files, symbols, etc. so it is desirable that this ID is as short as possible (even a single letter). For example, CT (to identify a "CAN Test Network"), B (to identify a CAN-B network), etc.

  Note: Don't use quotes in the network ID definition (since quotes are not C-identifier compliant).
- **Network description:** Some textual description for the given network. Used only for documentation purposes. Not used for code generation. Can be any text string consisting of multiple sentences, paragraphs, etc. as desired.

Nodes definitions

Node definitions need to be specified in tab "Network_and_Nodes".

A row per each node is expected in tab "Network_and_Nodes" starting below the corresponding titles row.

- **Node Name:** Indicates the name of the node. Needs to comply with C-identifier syntax. This name will be used extensively across the generated code for the given node. Hence, it is recommended that the name is short. For example: NODE_1, DCM, BCM, etc.
- **Description:** A textual description of the node. Used only for documentation purposes. Not used for code generation. Can be any text string consisting of multiple sentences, paragraphs, etc. as desired.

Messages definitions

Message definitions need to be specified in tab "Messages".

- **Message Name:** Name of the message. Needs to comply with C-identifier syntax and needs to be unique per message.

- **Message ID:** message identifier. An integer number not exceeding 11-bits for standard CAN id messages and 29-bits for extended id messages. Needs to be unique for each message within the same network. It can be entered in decimal format or hexadecimal format (with 0x prefix).
- **Extended Frame?:** Indicates if the given message has extended id. Use values "yes" or "no" (without quotes).
- **Data Length (bytes):** Indicates the length of the data for the message. It can be from 1 to 8.
- **Description:** A textual description of the message. Used only for documentation purposes. Not used for code generation. Can be any text string consisting of multiple sentences, paragraphs, etc., as desired.
- **Publisher:** node publishing (transmitting) the message. It shall be one of the node names defined in tab "Network_and_Nodes".
- **Subscribers:** Node or nodes subscribing to (receiving) the message. Timeout in milliseconds for each subscribing node can be specified within parentheses. Shall be a comma separated list of node names (valid nodes defined in "Network_and_Nodes"). If a timeout definition is required for a given subscribing node it shall be specified between parentheses next to the node's name. Examples:
  - Node_1, Node_2 (100), Node_3, Node_4 (1000)   --> Message is subscribed by nodes Node_1, Node_2, Node_3 and Node_4. Timeout of 100 ms is defined for Node_2 and a timeout of 1000 ms is defined for Node_4.
  - Node_5 --> Only Node_5 is a subscriber of the node.
- **Tx Type:** Indicates the transmission type from the publishing node perspective. Allowed values are:
  - cyclic: message is cyclic. Period needs to be defined.
  - cyclic_spontan: Message is periodic and also expects to have spontaneous transmission instances. Period needs to be defined.
  - spontan: message is of spontaneous transmission. Period doesn't need to be defined (is not applicable).
  - BAF: by-active-function transmission. Message will be sent only upon activation from an event from application. Period needs to be defined (will be the period to be used while the active function is present).
- **Period (ms):** Period in milliseconds for the transmission messages of type cyclic, cyclic_spontan or BAF.
- **Repetitions (only for BAF):** number of instances to be transmitted when the BAF function/event is present. If left empty or set to value 0 (zero) it means that the BAF transmission will continue until the active function/event is cleared by application.

Signal definitions

Signal definitions need to be specified in tab "Signals".

- **Signal Name:** Name of the signal. Needs to comply with C-identifier syntax and needs to be unique per network.
- **Data Type:** Indicates the type associated to the signal. Allowed values are:
  - array: indicates that the signal is an array of bytes.
  - scalar: indicates that the signal is of an scalar type (integer, etc.).
  - *custom data* type: can use any name of an user-defined enumerated data type within tab "Data Types" (see section below).

- **Length (bits):** length in bits of the signal. Can be between 1 and up-to the message size (e.g., 64 bits for an 8-byte length message). The signal must fit in the message available space (space not occupied by other message signals).
- **Conveyor Message:** name of the message transporting the signal. Shall be a defined message name within tab "Messages".
- **Start Byte:** Layout information: starting byte position of the signal within the conveyor message.
- **Start Bit:** Layout information: starting bit position (with respect to the start byte) of the signal within the conveyor
  message. E.g. if a signal starts at absolute bit position 33 (starting counting with byte 0 and bit 0), then its start byte shall be 4 and its start bit shall be 1.

  Note: Bits occupied by the signal (as determined by its starting byte, starting bit and length) shall not overlap with other signals conveyed by the same message.
- **Initial Value:** value of the signal to be set during initialization. Shall be a valid value as per the associated signal type. For signals of type array, this value will be used in all the array's bytes. Left empty if no initial value is required (by default all data is initialized with all zeros).
- **Fail Safe value:** value for the signal to be set when the conveyor message times-out (from subscribing node's perspective). Shall be a valid value as per the associated signal type. If left empty then no change will be performed upon detection of conveyor message timeout (last known value will be kept). This is meaningless if the conveyor message doesn't have a defined timeout.
- **Offset:** if signal is of scalar type, it is often useful to define an offset and a resolution (see next parameter) to give a physical meaning to the signal's raw value. The offset and resolution are used to defined a linear equation for the physical value of the signal:
  - *physical_value* = **resolution** * *raw_value* + **offset**

  This information is used for documentation/display purposes only (not used for code generation). Can be any string (freely set) preferably representing a number (integer, decimal, etc.).
- **Resolution:** if signal is of scalar type, it is often useful to define a resolution to give a physical meaning to the signal's raw value. This information is used for documentation/display purposes only (not used for code generation).

  Can be any string (freely set) preferably representing a number (integer, decimal, etc.).
- **Unit:** if signal is of scalar or array type, it is often useful to define a unit to give a physical meaning to the signal's raw value. Physical units can be meters, kilograms, degrees, etc. Can be any string (freely set).

Custom Data Types Definitions

User can define custom enumerated data types. These data types need to be defined in tab "Data Types".

- **Type name:** name of the user-defined enumerated data type. Needs to comply with C-identifier syntax and be unique across all networks.
- **Enum Values:** definition of enumerated symbols and their explicit values (optional). Needs to be a list of comma-separated symbol definitions for the enumeration (a standard C enumeration will be generated for each user-defined type). If an explicit integer value is desired for a given symbol, it needs to be specified between parentheses next to the symbol's name.

Symbol names need to comply with C-identifier syntax and shall be *unique* amongst all enumerated symbols of the user-defined enumerated data types.

Examples:

- *Type name:* IgnitionState

  Enum values: Lock (0), Accessory (1), Run (2), Start (3), SNA (7)

  This defines an enumerated type named "IgnitionState" with symbols Lock (with an explicit value of 0), Accessory (with an explicit value of 1), Run (with an explicit value of 2), Start (with an explicit value of 3) and SNA (with an explicit value of 7).
- *Type name:* WeekDays

  Enum values: Moday, Tuesday, Wednesday, Thursday, Friday

  Symbols do not have explicit values so they will get values as per a standard C enumeration definition, i.e., Monday will get zero, Tuesday will get 1, etc..
- *Type name:* MixedSymbols

  Enum values: SomeSymbol, AnotherSymbol (3), OneMoreSymbol

  SomeSymbol doesn't have an explicit values so it will get a zero by default (C-enumeration behavior) AnotherSymbol will get a value of 3 and OneMoreSymbol will automatically get a value of 4 (C- enumeration behavior).

Configuring the calvos-engine generator

The calvos-engine responsible of generating C source code based on the CAN network definition can be configured. Main configurable parameters are listed in tab "Config" of the CAN Network Definition template.

In tab "Config", following fields are present:

- **Parameter:** contains the name of the parameter (this field shall not be modified).
- **Type:** type of the parameter (this field shall not be modified).
- **Description:** textual description of the parameter function/use (this field shall not be modified).
- **Default value:** value assumed by default by calvos-engine (this field shall not be modified).
- **User value:** for user-configurable parameters (see below), the user can enter here a custom value. If no user value is specified or it is wrongly specified (wrong syntax, not allowed value), then the parameter will assume its default value.

User-configurable parameters are listed below:

- `CAN_gen_nodes`: A list of nodes for which C-code is going to be generated. Only node names defined in tab "Network_and_Nodes" are allowed.

  The list needs to be contained between `[]`.

  Each node needs to be encompassed by double quotes `"Node_Name"`

  Each node needs to be comma separated.

  If User value is empty or is equal to `[]`, then code for all network nodes will be generated.

  Examples:

  - ["Node_1"] --> will generate code only for Node_1
  - ["Node_2", "Node_4"] --> will generate code for nodes Node_2 and Node_4
- `CAN_gen_file_full_names`: If set to TRUE will force full-names for generated source code and symbols. Refer to section Source Code Names Optimization for more details on this parameter.

- `CAN_tx_confirm_msg_id`: If True, TX confirmation will need to check for transmitted ID, otherwise confirmation will be done without confirming the transmitted msg ID from CAN HAL.
- `CAN_tx_task_period`: Period in milliseconds of the Task for transmitting *cyclic* and *cyclic_spontan* messages.
- `CAN_tx_period_tolerance`: Tolerance in percentage to deviate from TX message period before throwing a warning message.
  Shall be between 0 and 100.
- `CAN_rx_data_init_val`: Integer value to be used for initializing all data in the RX unified data buffer. Can be decimal or hex number (with 0x prefix). Shall be between 0 and 255 (1 byte).
- `CAN_tx_data_init_val`: Integer value to be used for initializing all data in the TX unified data buffer. Can be decimal or hex number (with 0x prefix). Shall be between 0 and 255 (1 byte).
- `CAN_enum_sym_prefix`: Prefix for generated symbols of the signal enumerations for custom data types. Use $ to indicate an empty string (meaning no prefix will be added). Use wildcard ${NWID} to indicate the network ID. By default a prefix "`kCAN_`" plus the network name is added to each symbol. This means that by default, this parameter has the value: `kCAN_${NWID}_` .
- `CAN_enum_sym_prefix`: Prefix for generated type names of signal enumerations. Leave empty to name the generated types exactly as the source data_type. Use wildcard ${DATATYPE} to indicate the Data_Type name. By default prefix "`t_`" will be added to the source data_type name. This means that by default, this parameter has the value: `t_${DATATYPE}`.

Note: The rest of parameters are not expected to be defined by the user or are currently not implemented so then do not modify the "User Value" for them.

Calvos Project definition

Once the CAN network is defined it needs to be included in a calvos project for its processing and code generation.

For the instructions here we will start with an empty project and include our CAN Network definition. Will assume a CAN network definition contained in a file named "*CAN Network Definition.ods*" (any other name can be used by the user).

1. Define a location for the calvos project (referred as *root* folder in these instructions). For our example this project's *root* folder will be "*C:\my_calvos_project*".
2. Create an empty project in the project's root folder named "*calvos_project.xml*":

   C:\my_calvos_project\calvos_project.xml

   An empty project is an XML file with following contents:

```
<CalvosProject xmlns:clv="calvos" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../schemas/_calvos_project.xsd">
  <Name>Demo Calvos Project</Name>
  <Desc>Demo Calvos Project generated from found components with templates.</Desc>
  <Version>Version</Version>
  <Date>Date</Date>
  <Components>
  </Components>
  <Params />
  <Metadata>
    <!-- Do NOT modify these metadata information. -->
    <clv:TemplateId>calvos_project</clv:TemplateId>
    <clv:TemplateVer>0.1.0</clv:TemplateVer>
    <clv:TemplateTitle>Calvos Project</clv:TemplateTitle>
    <clv:TemplateDesc>Template for modeling a Calvos Project</clv:TemplateDesc>
    <clv:TemplateDate>2021-02-24</clv:TemplateDate>
  </Metadata>
</CalvosProject>
```

3. Create a sub-folder named "*usr_in*" in the project's *root* folder:

   C:\my_calvos_project\usr_in

4. Copy the network definition file *CAN Network Definition.ods* into the *usr_in* folder created above:

   C:\my_calvos_project\usr_in\CAN Network Definition.ods

5. Create the instance for the CAN network within the *calvos_project.xml* and set its input file information:

   - Create the following XML `Component` node within the `Components` node in *calvos_project.xml*:

   -
     ```
     <Component type="comgen.CAN">
       <Name>CAN Network</Name>
       <Desc>Models a network for Controller Area Network (CAN) protocol.
     </Desc>
       <Input type="ods">"usr_in/CAN Network Definition.ods"</Input>
     </Component>
     ```

6. Save the updated *calvos_project.xml* file.
   The contents should look as follows:

   ```
   <CalvosProject xmlns:clv="calvos"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="../schemas/_calvos_project.xsd">
     <Name>Demo Calvos Project</Name>
     <Desc>Demo Calvos Project generated from found components with templates.
   </Desc>
     <Version>Version</Version>
     <Date>Date</Date>
     <Components>
       <Component type="comgen.CAN">
         <Name>CAN Network</Name>
         <Desc>Models a network for Controller Area Network (CAN) protocol.
   </Desc>
         <Input type="ods">"usr_in/CAN Network Definition.ods"</Input>
       </Component>
     </Components>
     <Params />
     <Metadata>
       <!-- Do NOT modify these metadata information. -->
       <clv:TemplateId>calvos_project</clv:TemplateId>
       <clv:TemplateVer>0.1.0</clv:TemplateVer>
   ```

```
    <clv:TemplateTitle>Calvos Project</clv:TemplateTitle>
    <clv:TemplateDesc>Template for modeling a Calvos 
Project</clv:TemplateDesc>
    <clv:TemplateDate>2021-02-24</clv:TemplateDate>
  </Metadata>
</CalvosProject>
```

Our project is now ready to be processed.

Processing Calvos Project

In order to generate our project we need to run the following command:

`python -m calvos -p c:\my_calvos_project\calvos_project.xml`

If everything went Ok then the generated source code shall be located in the default output folder:

- *c:\my_calvos_project\out*

Information about the project processing and possible warnings/errors found during the processing can be consulted in the generated log file located in the project's *root* folder:

- *c:\my_calvos_project\log.log*

If `calvos` package failed to execute check if there are missing python dependencies that need to be installed or if the proper python version is being used (this can be checked by typing in `python -v` a console. Shown version shall be equal to or greater than 3.7).

Generated code shall correspond to the defined CAN network as well as some global files used across all calvos project.

Exporting the Generated Code

It is possible to automatically export the generated source code into an user-defined path. At the end, the goal of the calvos generated code is to be integrated into the user's project. For this, the optional "export" command line argument `-e` can be used to indicate such location where the generated code will be automatically copied to.

If the export argument `-e` is used then the files in the exporting folder will be overwritten without user prompt. This should normally not be a problem since the generated source code that requires user instrumentation is generated with prefix "USER_" and hence user instrumentation shouldn't be overwritten (user shall remove the prefix "USER_" once instrumentation is done). Nevertheless, it may be many reasons why a backup of the code that the export overwrites is desired. For this, a "backup" command line argument `-b` is in place. If `-b` is provided it shall indicate the path where the backup of the files to be replaced during the export operation is to be located.

The backup strategy produces multiple sets of backed-up files. Up-to 10 sets will be created if the calvos-engine is invoked multiple times with the export `-e` argument. These sets will be located in the path specified in the backup argument `-b`.

Instrumenting User Code

Generated source code is classified in two categories: static code and user code.

- **Static Code:** Is the generated source code that doesn't need any user instrumentation and hence, is not expected to be modified by the user.
- **User Code:** Is the generated source code that needs some user instrumentation. Source code files in this category are generated with a prefix "USER_" in the file name. The user *shall instrument* the required code and once done the files need to be re-named to remove such "USER_" prefix.

CAN Interaction Layer Generated Source Code

Global Generated Source Code

Couple of generated files are general ones not tied to the CAN network but required for a proper integration of the calvos generated source code into the user's target project.

Those global files are:

File Name	File Contents
`calvos_types.h`	Typedefs used across generated source code. If a type re-definition error is found when integrating this code into the target project then comment out the re-defined types in this file ensuring that the original definition in the target project matches the ones commented out.
`calvos.h`	Header file for including global headers like `calvos_types.h` across all calvos generated source code.

CAN Generated Source Code

The generated source code for the CAN network(s) is categorized in common code common to all CAN entities), network specific code and node specific code.

Common Code

Code common to all CAN generated source code. These set of files will be generated once regardless of the number of networks or nodes defined in the project.

File	File contents
`comgen_CAN_common.h`	Contains definitions used across all CAN generated code. For example, common data structures, symbol definitions, internally used common functions, etc.
`comgen_CAN_common.c`	Implements common functionality for CAN. For example, logic for queuing transmission messages, binary search tree for received messages, etc.

Network-specific Code

Code that needs to be generated per each defined network in the project. E.g., if two networks are defined, then two sets of these source code files will be generated, one per each network.

When referring to a file in this category or symbols within those files, the wildcard NWID (network ID) will be used in this documentation to represent the corresponding network ID.

For example, if the defined network has an ID equal to 'C', then a file referred as `comgen_CAN_NWID_network.h` corresponds to a generated file with name`comgen_CAN_C_network.h`.

File	File contents
`cog_comgen_CAN_NWID_network.h`	Defines network-wide symbols and data structures for messages information (lengths, ids, data structures, etc.), signals (APIs to access signals) and user-defined enumerated data types.

Node-specific Code

Code that needs to be generated per each defined node within a network that has been selected for code generation (selected in parameter `CAN_gen_nodes`). A set of these files will be generated for each selected node in each defined network. These source code files will contain the corresponding node's name in the file names and symbols within them.

When referring to a file in this category, the wildcard NODEID (node name) will be used in this documentation to represent the corresponding node. For example, if the generated node name is 'NODE_1', then a file referred as `comgen_CAN_NWID_NODEID_node_network.h` corresponds to a generated file with name `comgen_CAN_C_NODE_1_node_network.h`.

File	File contents
`comgen_CAN_NWID_NODEID_node_network.h`	Defines symbols for node specific message information (message directions, timeouts, etc.), direct signal access macros (directly accessing from node's data buffers), etc.
`comgen_CAN_NWID_NODEID_hal.h`	Header for node's interfaces with CAN hardware abstraction layer in the target MCU.
`comgen_CAN_NWID_NODEID_hal.c`	Implementation of node's interfaces with CAN hardware abstraction layer in the target MCU.
`cog_comgen_CAN_NWID_NODEID_core.h`	Header for the node's CAN core functionality.
`cog_comgen_CAN_NWID_NODEID_core.c`	Implementation of the node's CAN core functionality.
`comgen_CAN_NWID_NODEID_callbacks.h`	Header for the node's callbacks
`comgen_CAN_NWID_NODEID_callbacks.c`	Implementation of the node's callbacks

Source Code Names Optimization

In order to avoid generating source code with long (and probably cumbersome) names (file names plus symbols names inside them). A name "optimization" is available. This optimization is only active if parameter `CAN_gen_file_full_names` is set to FALSE.

If only one network is defined in the project then the NWID wildcard will be replaced by an empty string rather than the network's ID (if only one network is defined then there is no need to generate multiple network specific files and hence there is no need to distinguish between the files and symbol names like in the case of having multiple networks).

If only one node is generated in a given network, the corresponding NODEID wildcard will be replaced by an empty string rather than the node's name. Similar logic as in the NWID optimization is applied (no need to distinguish symbols for several nodes).

For example, if only one network (network id: "CT") is defined in the project and only a single node ["NODE_1"] is selected for generation, the file `comgen_CAN_NWID_NODEID_node_network.h` will be generated with the name `comgen_CAN_network.h`. However, if parameter `CAN_gen_file_full_names` is set to TRUE, the optimization won't take place and the generated file will have its full name: `comgen_CAN_CT_NODE_1_node_network.h`.

Note: Optimization is also applied to the defined symbols within the generated files as required.

Integration into Target Project

The integration needs to be done in a per-node basis. This means, each generated node needs to be integrated in the target's project. Hence, the hardware abstraction layer needs to be also defined per each node.

It is assumed that a single node will own/use a single CAN peripheral in the target MCU. For example, if two nodes are generated for a single network then two CAN peripherals are required in the MCU, one per each node. However and even though this is the intention, user may decide to use one CAN peripheral for multiple nodes (not recommended).

Initialization

Function `can_NWID_NODEID_coreInit()` needs to be called during initialization phase of the SW. Header `comgen_CAN_NWID_NODEID_core.h` needs to be included in order to import this function.

HAL Integration

For each node, a HAL integration is required for the CAN reception and transmission by the target's MCU CAN peripheral.

HAL integration for reception

Following tasks are required in order to integrate the reception of CAN messages with the MCU's HAL (these steps need to be performed for each generated node):

1. Initialize CAN HAL. User shall instrument during SW initialization phase, code for initializing the CAN peripheral with following characteristics:
    - CAN reception shall allow all the defined node RX messages (set acceptance filters appropriately).
    - Preferably an ISR shall be configured and instrumented corresponding to a successful reception of a CAN msg.

  - Preferably an ISR shall be configured and instrumented corresponding to a successful transmission of a CAN msg.
2. Function `can_NWID_NODEID_HALreceiveMsg()` needs to be invoked within the HAL function that signals the reception of a CAN message by the CAN peripheral (typically within ISR context). If multiple nodes are generated, each of the generated functions need to be invoked from the corresponding CAN peripherals.

   Following arguments need to be provided:

   - `msg_id`: an unsigned integer containing the received message id (either and standard or an extended id value).
   - `data_in`: a byte-pointer pointing to the place where the received data resides from the HAL layer. Data will be copied from this pointer into the generated reception data buffer `can_NWID_NODEID_RxDataBuffer`.
   - `data_len`: Length (number of bytes) of the received data. The data copy operation from the HAL to the reception buffer will be done for this amount of bytes in the case that the message is a valid rx message for the node.

This function will first perform a search in order to determine if the received message id belongs to the node (its a subscribed message for the node). If it is then the received data will be copied to the reception buffer, corresponding message available flags will be set and the message reception callback will be invoked.

2. Write user-defined code within the reception message callback `can_NWID_NODEID_MESSAGEID_rx_callback()` as needed. MESSAGEID corresponds to the received message name. These callbacks are defined in file `comgen_CAN_NWID_NODEID_callbacks.c`.

   Note: If function `can_NWID_NODEID_HALreceiveMsg()` is called within ISR context, then the callbacks will be also called within ISR context so the user code shall be as small as possible.

   Is also possible to poll for the reception of messages based on their available flags (in case user doesn't want to use the callbacks directly).
3. Consume message's available flags (polling for received messages) if required.

HAL integration for transmission

Following tasks are required in order to integrate the transmission of CAN messages with the MCU's HAL:

1. Implement CAN HAL transmission. This means instrumenting code within function `can_NWID_NODEID_HALtransmitMsg`.

   HAL code to trigger a CAN message transmission by the associated CAN peripheral shall be put inside the body of this function. Data regarding the message to be transmitted can be taken from the provided argument `msg_info` as follows.

   - Message id to be transmitted can be extracted from `msg_info->id`.
   - Message length to be transmitted can be extracted from `msg_info->fields.len`.
   - Message data to be transmitted can be taken from `msg_info->data`. `msg_info->data` is a pointer to a byte array. Hence, individual bytes can be accessed using `msg_info->data[i]` (where `i` is the desired byte index) or simply use `msg_info->data` to point to the beginning of the array containing the transmission data.
   - Indication of extended id can be taken from `msg_info->fields.is_extended_id`. This is a boolean value with `1` indicating that the message has an extended id and `0` if it has an standard id.

2. Implement function for getting the ID of the message just transmitted by the HAL. If generation parameter `CAN_tx_confirm_msg_id` is set to `True` then function `can_NWID_NODEID_HALgetTxdMsgId` needs to be instrumented with code to return the ID of the CAN message just transmitted by the HAL. This function will be used by the CAN TX confirmation function `can_NWID_NODEID_HALconfirmTxMsg`. If generation parameter `CAN_tx_confirm_msg_id` is False then this function won't get generated. In this case, message confirmation will be done as long as `can_NWID_NODEID_HALconfirmTxMsg` is invoked regardless of the ID of the message transmitted by HAL.
3. Invoke callback for message transmission confirmation from HAL. Function `can_NWID_NODEID_HALconfirmTxMsg()` needs to be called within the HAL function that signals the confirmation of the latest CAN message transmission (typically within ISR context). This function doesn't require any argument.

Transmission Task integration

Function `can_task_<time>ms_NWID_NODEID_txProcess` is generated and is in charge of triggering the transmission of the messages defined as `cyclic` or `cyclic_spontan` with their defined periods. This function needs to be invoked from a periodic task of the target OS with a period equal to parameter `CAN_tx_task_period` in milliseconds. The function name will indicate the required periodicity. For example, if `CAN_tx_task_period` is set to 10ms, then generated function will be named `can_task_10ms_NWID_NODEID_txProcess` and shall be called with such periodicity. Default value of `CAN_tx_task_period` is indeed 10ms.

Transmission of non-cyclic messages

Function `can_task_<time>ms_NWID_NODEID_txProcess` only deals with cyclic transmissions of messages. If spontaneous transmissions are required then user can invoke function `can_NWID_NODEID_transmitMsg` as required. Refer to section Transmitting Messages for more information.

Transmission Retry Mechanism

Calvos CAN IL implements a queueing and retry mechanism for transmission messages. If for a given reason a transmission request is rejected by the HAL (e.g., due to it is busy in another transfer) then the requested TX message gets queued for a later retransmission attempt. The length of this queue is determined by parameter `CAN_tx_queue_len` which defaults to 5. Function `can_NWID_NODEID_txRetry` is indeed in charge of performing such queue navigation and retransmission attempts. Therefore, function `can_NWID_NODEID_txRetry` needs to be invoked by the user in one of the following two options:

1. Invoke `can_NWID_NODEID_txRetry` at a task level. For example, invoking this function in a 5ms periodic task will attempt a transmission retry every 5ms until all queued messages are transmitted. If it is up-to the user to define how fast to invoke this function. It can even be invoked in the same task where `CAN_tx_task_period` is called. In this last case, it is recommended to first call `can_NWID_NODEID_txRetry` and then `CAN_tx_task_period` in order to give priority to the retries.
2. Invoke `can_NWID_NODEID_txRetry` on event after a transmission of a CAN message by the HAL. In this option function `can_NWID_NODEID_txRetry` can be invoked right after a CAN transmission confirmation from the HAL. For example within the TX ISR. This will lead to a retry of a queued transmission as soon as possible most likely creating a back-to-back transmission of a queued message.

Which option to use (or even another one) is up-to the user to decide.

Application Usage

Transmitting messages

Task function `can_task_<time>ms_NWID_NODEID_txProcess` takes care of the transmission of *cyclic* messages as well as of the transmission of the cyclic instances of a *cyclic_spontan* message according to their defined periods.

For the transmission of spontaneous instances, user needs to call function `can_NWID_NODEID_transmitMsg` passing as argument the index of the message to be sent. Those indexes are named `kCAN_NWID_NODEID_txMsgIdx_MESSAGENAME` and are elements of an enumeration called `CAN_NWID_NODEID_txMsgs` declared in header file `comgen_CAN_NWID_NODEID_node_network.h`.

The data to be transmitted either by task function `can_task_<time>ms_NWID_NODEID_txProcess` or by function `can_NWID_NODEID_transmitMsg` is directly taken from the TX unified buffer (`can_NWID_NODEID_TxDataBuffer`) so user needs to first update such data to be transmitted. Refer to section "Accessing Data of Messages" for details on how to update TX data.

Getting Transmission State

Transmission messages can be in any of the following states:

- **Idle** (`kCANtxState_idle`): This is the initial value and represents that the message has never been requested for transmission.
- **Requested** (`kCANtxState_requested`): A TX message gets into this state if during a transmission request, the CAN HAL didn't accept the request and also it was not possible to queue the message for later retransmission (e.g., queue was full). In this state is recommended that the application manually performs a new transmission request.
- **Queued** (`kCANtxState_queued`): Message was requested for transmission but was not accepted by the CAN HAL and the message got queued for later retransmission (automatic re-transmission).
- **Transmitting** (`kCANtxState_transmitting`): Message was accepted by the CAN HAL for transmission. Message will remain in this state until its transmission is confirmed by the CAN HAL.
- **Transmitted** (`kCANtxState_transmitted`): CAN HAL confirmed the successful transmission of the message in the bus.

This states are modeled with enumeration data type `CANtxState` defined in header file *comgen_CAN_common.h*.

Application can read the state of a transmission message with generated macro `CAN_NWID_NODEID_get_tx_state_MESSAGENAME`. The variable receiving the macro value shall be of type `CANtxState`.

Example of a message transmission:

- Message:
  - Name: MESSAGEX
  - Message direction: transmission
  - Message transmission type: spontan
  - Length: 5 bytes

- MESSAGE12 signals:
    - Signal_X1: (start bit = 0, start byte = 0, length = 8)
    - Signal_X2: (start bit = 0, start byte = 1, length = 32)

```
void some_app_function(void)
{
    CalvosError tx_return; /* Hold the TX request returned value */
    uint8_t my_Signal_X1; /* Local variable for signal Signal_X1 */
    uint32_t my_Signal_X2; /* Local variable for signal Signal_X2 */

    /* Application code to determine local values for Signals Signal_X1
       * and Signal_X2 */
    my_Signal_X1 = 0xDE;
    my_Signal_X2 = 0xFE00AABB;

    /* Directly update TX unified buffer with local data */
    /* Data can also be written using S_MESSAGEX structure. Refer
     * to section "Accessing Data of Messages" for more information. */
    CALVOS_CRITICAL_ENTER();
    CAN_NWID_NODEID_update_direct_Signal_X1(my_Signal_X1);
    CAN_NWID_NODEID_update_direct_Signal_X2(my_Signal_X2);
    CALVOS_CRITICAL_EXIT();

    /* Request message 'MESSAGEX' transmission. If this message was of type
     * 'cyclic' then it will be automatically transmitted when its
     * period expires. */
    tx_return = can_NWID_NODEID_transmitMsg(kCAN_NWID_NODEID_txMsgIdx_MESSAGEX);
    /* Check if transmission request was taken */
    if(tx_return == kError)
    {
        /* Message was not accepted for transmission by the CAN HAL and was
         * neither queued for automatic retransmission. Application needs
         * to manually trigger its transmission again. Probably some
milliseconds
         * later...*/
        do
        {
            /* Blocking API to wait for some time (OS specific), in
             * this example this "some_app_function" function gets
             * blocked for 10ms and then it continues its execution.
             * This logic doesn't apply if task was set as periodic!!
             * since it is not convenient to block the task by itself.
             * This 'wait_blocking_ms' function is just illustrative,
             * proper one from the target OS needs to be used. */
            wait_blocking_ms(10);
            /* Manually request message transmission again */
            tx_return = \
                can_NWID_NODEID_transmitMsg(kCAN_NWID_NODEID_txMsgIdx_MESSAGEX);
        }while(tx_return == kError);

        /* Reaching here means message transmission was accepted. */
    }
}
```

Receiving Messages

The processing of received messages is performed by function `can_NWID_NODEID_HALreceiveMsg`. That function will identify if the received message matches the expected ID and data length of a subscribed message by the node. If so, this function also copies the received data from the HAL into a unified reception buffer called `can_NWID_NODEID_RxDataBuffer`, sets the *available flags* (more information about available flags in following sections) for the received message/signals and invokes the corresponding reception callback.

The application can decide to use the reception callbacks directly (typically within ISR context) or to do a polling for the reception of the messages by polling its available flags (polling is done at task level and hence out of ISR context).

The received data is available from the data buffer `can_NWID_NODEID_RxDataBuffer` and can be accessed by means of generated data structures (refer to section "CAN Data Structures" for more details) or by means of access macros (refer to section "CAN Access Macros" for more details)

Using the reception callbacks

Upon reception of a valid subscribed message, a callback with the name `can_NWID_NODEID_MESSAGENAME_rx_callback` will be invoked. If function `can_NWID_NODEID_HALreceiveMsg` is called within an ISR then the reception callbacks will be also called within that ISR context.

Inside the callback, data can be accessed via the CAN Data Structures or the CAN Access Macros.

Using the available flags (polling for received messages)

A set of available flags is generated per each message and per each signal inside each message. User can decide which one to use as needed.

Message-Level available flags

Per each subscribed message a set of message-level available flags are generated. The quantity of flags is determined by the word-size of the target MCU. For example, for a 32-bit MCU, 32 available flags will be generated per message. Those available flags are 1-bit each and will be all set to `1` upon reception of the message.

These message-level available flags can be used by the application to identify the reception of a given message. The application is responsible of consuming (clearing, setting to `0`) those available flags as needed.

To read the message-level available flags, macros with name `CAN_NWID_NODEID_get_msg_avlbl_flags_MESSAGENAME` are generated. The returned value is a union of type `FlagsNative`. Once the macro is used, then each flag can be individually accessed (read, write) using the bitfield `flags` within the `FlagsNative` variable. Also the flags can be accessed as an array of bytes by means of the `all` element of the `FlagsNative` variable.

It is possible to clear all message available flags at once using macro `CAN_NWID_NODEID_clr_msg_avlbl_flags_MESSAGENAME`.

See example below for the usage of message-level available flags using all the available flags at once.

```
/* Call this some_app_function in a periodic tasks so that the message reception
 * is proppërly polled. */
void some_app_function(void)
```

```
{
    FlagsNative msg_avlbl_flags; /* variable for containing msg-available flags
*/

    /* poll for the reception of message "RX_MESSAGE" */
    /* ---------------------------------------------- */
    /* Get msg-available flags */
    msg_avlbl_flags = CAN_NWID_NODEID_get_msg_avlbl_flags_RX_MESSAGE();
    if(msg_avlbl_flags.all[0]) /* Checking against the first byte of the 'all'
element */
    {
        /* Message has been received! */
        /* Consume (clear) all message-available flags */
        CAN_NWID_NODEID_clr_msg_avlbl_flags_RX_MESSAGE();

        /* Code for getting the received data and doing something
         * with it goes here... Refer to section 'Accessing Data of Messages'
         * for more information. */
    }
    /* else --> Message has not been received. Do nothing */
}
```

This other example shows the usage of the message-available flags as individual flags.

```
/* Call this some_app_function in a periodic tasks so that the message reception
 * is propperly polled. */
void some_app_function(void)
{
    FlagsNative msg_avlbl_flags; /* variable for containing msg-available flags
*/

    /* poll for the reception of message "RX_MESSAGE" */
    /* ---------------------------------------------- */
    /* Get msg-available flags */
    msg_avlbl_flags = CAN_NWID_NODEID_get_msg_avlbl_flags_RX_MESSAGE();
    /* This function uses only one of the flags (flag0) so the other ones can be

     * used by other 'clients' of the same message. */
    if(msg_avlbl_flags.flags.flag0) /* Checking for flag0 */
    {
        /* Message has been received! */
        /* Consume (clear) only flag0 message-available flag */
        msg_avlbl_flags.flags.flag0 = 0;

        /* Code for getting the received data and doing something
         * with it goes here... */
    }
    /* else --> Message has not been received. Do nothing */
}
```

Signal-Level available flags

In addition to the message-available flags, a set of signal-available flags is generated per each subscribed message. In this case one signal-available flag will be generated per each signal defined for the message (if message has 3 signals, then 3 signal-available flags will be generated). Those signal-available flags can be read/cleared by a pair of generated macros:

`CAN_NWID_NODEID_get_avlbl_SIGNALNAME` and `CAN_NWID_NODEID_clr_avlbl_SIGNALNAME` respectively.

Example of usage for signal-available flags.

```
/* Call this some_app_function in a periodic tasks so that the message reception
 * is propperly polled. */
void some_app_function(void)
{
    uint8_t signal_avlbl_flag; /* variable for containing a signal-available
flag */

    /* poll for the reception of signal "Signal_RX" of message "RX_MESSAGE" */
    /* ---------------------------------------------------------------- */
    /* Get signal-available flag */
    signal_avlbl_flag = CAN_NWID_NODEID_get_avlbl_Signal_RX();
    if(signal_avlbl_flag) /* Checking for arrival of signal "Signal_RX" */
    {
        /* Signal "Signal_RX" has been received! */
        /* Consume (clear) the signal-available flag */
        CAN_NWID_NODEID_clr_avlbl_Signal_RX();

        /* Code for getting the received data of signal "Signal_RX"
         * and doing something with it goes here... */
    }
    /* else --> Signal has not been received. Do nothing */
}
```

Accessing Data of Messages

Unified Data Buffers

Calvos CAN IL generates two data buffers, one used for the received data and another one for the data to be transmitted. Since a single buffer is used for all received data and another one for all transmitted data they will be referred to as the "unified reception (RX) buffer" and the "unified transmission (TX) buffer".

The RX unified buffer is generated with the name `can_NWID_NODEID_RxDataBuffer` and its length is equal to the sum of the individual lengths of all the subscribed messages of *NODEID*. This in order to be able to contain all the subscribed data of interest.

Data received by the HAL is automatically copied to the RX unified buffer on its corresponding place. As such this buffer is very likely to be written within ISR context so that special care shall be taken when accessing this buffer directly by the app (access needs to be protected by critical sections to avoid data corruptions). Suggested procedure to read data from this RX buffer is explained in following sections.

The TX unified buffer is generated with the name `can_NWID_NODEID_TxDataBuffer`. Its length is equal to the sum of the individual lengths of all the published messages of *NODEID*.

When a message is to be transmitted the data to transmit is directly taken from this TX unified buffer. Hence, the data needs to be firstly updated in the TX unified buffer. Suggested procedure to read/write data from/to this TX buffer is explained in following sections.

Accessing Received Data

Data received by the CAN HAL is automatically copied to the RX unified buffer. As such, this buffer is very likely to be written within ISR context and, hence, special care shall be taken by the application when accessing this buffer.

The suggested procedure for accessing the received data is as follows:

1. Confirm the reception of the message of interest. If reception callback is used then this is given as a fact. If polling method is used then confirm that the available flag(s) is set. Refer to section "Receiving Messages" for more details on this.
2. *Safely* copy the data of the received message from the RX unified buffer into a temporal local CAN data structure from application side. This can be accomplished by using the generated "get message" macros that are named `CAN_NWID_NODEID_get_msg_MESSAGENAME`.

   A pointer to the CAN Data Structure for the given message shall be provided as argument to the "get message" macro. This CAN Data Structure can be defined locally using a generated data type with name `S_MESSAGENAME`. Refer to section "CAN Data Structures" for more information about the generated CAN data structures.

   The "get message" macro will enter a critical section, copy the data from the RX unified buffer to the provided CAN data structure and then exit the critical section. **Note:** Macros `CALVOS_CRITICAL_ENTER` and `CALVOS_CRITICAL_EXIT` defined in file `USER_general_defs.h` shall be properly instrumented by the user for this to work.
3. Use the data from the **local** CAN data structure for the application purposes. Refer to below sections "CAN Data Structures" and "CAN Access Macros" for details and examples on how to read the individual signal's data from the CAN data structure.

An example of accessing received data by *polling* for message reception is shown below:

```
/* Call this some_app_function in a periodic tasks so that the message reception
 * is proppерly polled. */
void some_app_function(void)
{
    FlagsNative msg_avlbl_flags; /* variable for containing msg-available flags
*/
    S_RX_MESSAGE local_msg_data; /* Local CAN data structure for message
RX_MESSAGE */

    /* poll for the reception of message "RX_MESSAGE" */
    /* ---------------------------------------------- */
    /* Get msg-available flags */
    msg_avlbl_flags = CAN_NWID_NODEID_get_msg_avlbl_flags_RX_MESSAGE();
    /* This function uses only one of the flags (flag0) so the other ones can be

     * used by other 'clients' of the same message. */
    if(msg_avlbl_flags.flags.flag0) /* Checking for flag0 */
    {
        /* Message has been received! */
        /* Consume (clear) only flag0 message-available flag */
        msg_avlbl_flags.flags.flag0 = 0;

        /* Get the data from the RX unified buffer */
        /* Data will be copied from RX unified buffer into the local
         * CAN data structure. */
        CAN_NWID_NODEID_get_msg_RX_MESSAGE(&local_msg_data);
        /* Now individual signals can be accessed from the
         * local CAN data structure. */
        if(local_msg_data.s.Signal_RX == 0xAA)
        {
            /* Do something if received data for Signal_RX is
             * 0xAA .... */
        }

        /* More code for doing something with received data
         * goes in here... */
    }
    /* else --> Message has not been received. Do nothing */
}
```

An example of accessing received data by using the reception *callback* is shown below:

```
/* ===========================================================================*/
/** Callback for RX_MESSAGE reception.
 *
 * Invoked within ISR context whenever RX_MESSAGE is received by node NODEID
 *  of network NWID.
 * ===========================================================================*/
void can_NWID_NODEID_RX_MESSAGE_rx_callback(void)
{
    S_RX_MESSAGE local_msg_data; /* Local CAN data structure for message
RX_MESSAGE */

    /* There is no need to check for available flags since this
     * callback invocation means that there is already newly
     * received data available. */

    /* Get the data from the RX unified buffer */
    /* Data will be copied from RX unified buffer into the local
     * CAN data structure. */
    CAN_NWID_NODEID_get_msg_RX_MESSAGE(&local_msg_data);

    /* Now individual signals can be accessed from the
     * local CAN data structure. */
    if(local_msg_data.s.Signal_RX == 0xAA)
    {
        /* Do something if received data for Signal_RX is
         * 0xAA .... */
    }

    /* More code for doing something with received data
     * goes in here but keep it short since this is called
     * within ISR context... */
}
```

If for some reason, the application needs to directly read from the RX unified buffer "get direct" macros are also generated for each signal. Those macros follow the go by the name CAN_NWID_get_direct_SIGNALNAME.

The "get direct" macros do operate directly over the RX unified buffer and do not have any protection related to *critical sections*, etc. Hence, is responsibility of the application to properly make use of these macros in order to avoid data corruption. More details and examples for these macros can be found in section "Signals Direct Access Macros".

Accessing Transmission Data

When a message is to be transmitted the data to transmit is directly taken from the TX unified buffer. Hence the data needs to be firstly updated in the TX unified buffer.

The suggested procedure for updating the data to be transmitted is as follows:

1. *Safely* get a "local copy" of the data currently present in the TX unified buffer for the message that needs to be transmitted. This can be accomplished by using the generated "get message" macros that are named `CAN_NWID_NODEID_get_msg_MESSAGENAME`.

   A pointer to the CAN Data Structure for the given message shall be provided as argument to the "get message" macro. This CAN Data Structure can be defined locally using a generated data type with name `S_MESSAGENAME`. Refer to section "CAN Data Structures" for more information about the generated CAN data structures.

   The "get message" macro will enter a critical section, copy the data from the TX unified buffer to the provided CAN data structure and then exit the critical section.
2. Update the required data to be transmitted in the *local* CAN data structure. Do any required update to the data that needs to be transmitted operating over the local CAN data structure. Refer to below sections "CAN Data Structures" and "CAN Access Macros" for details and examples on how to write the individual signal's in the CAN data structure.
3. Synchronize the updated transmission data in the local CAN data structure with the TX unified buffer. This is done by using the generated "update" macro named `CAN_NWID_NODEID_update_msg_MESSAGENAME`.

   The argument of macro `CAN_NWID_NODEID_update_msg_MESSAGENAME` is the pointer to the local CAN Data Structure with the data to be transmitted.

   The macro will enter a critical section, copy the data from the local CAN data structure to the TX unified buffer and then exit the critical section.
4. If the conveyor message is cyclic, then the message transmission will be triggered automatically sending the data available in the TX unified buffer. If an spontaneous transmission is required of the newly updated data then trigger such transmission by calling function `can_NWID_NODEID_transmitMsg`.
5. Check if TX message transmission is confirmed by the CAN HAL. For example, if TX message state is `kCANtxState_transmitted`. Refer to section "Getting Transmission State" for more information about transmission states, or if the return value of `can_NWID_NODEID_transmitMsg` is `kNoError`.

See following example of updating some signals for transmission based on above's procedure.

- Message:
  - Name: TX_MESSAGE
  - Length: 4 bytes
  - Transmission type: *spontan*
- TX_MESSAGE signals:
  - Signal_TX_1: (start bit = 0, start byte = 0, length = 8)
  - Signal_TX_2: (start bit = 0, start byte = 1, length = 8)

- Signal_TX_3: (start bit = 0, start byte = 2, length = 8)
- Signal_TX_4: (start bit = 0, start byte = 3, length = 8)

```
void some_app_function(void)
{
    S_TX_MESSAGE local_msg_data; /* Local CAN data structure for message
TX_MESSAGE */
    CANtxState tx_msg_state; /* variable for containing the tx message state */
    CalvosError tx_return; /* Holds the return value of the TX request */

    /* Step 1: Get local copy from TX unified buffer */
    CAN_NWID_NODEID_get_msg_TX_MESSAGE(&local_msg_data);

    /* Step 2: Modify signals as required by application... */
    local_msg_data.s.Signal_TX_1 = 25;
    local_msg_data.s.Signal_TX_3 = 6;
    /* In this example signals "Signal_TX_2" and "Signal_TX_4" don't need to
     * be modified so then they will be transmited with their previous values.
*/

    /* Step 3: Synchronize local data with the TX unified buffer */
    CAN_NWID_NODEID_update_msg_TX_MESSAGE(&local_msg_data);

    /* Step 4: Request spontaneous transmission of TX_MESSAGE */
    /* If TX_MESSAGE was cyclic, then it will be automatically
     * transmitted in its next period. In this example assuming
     * that message is spontan */
    tx_return =
can_NWID_NODEID_transmitMsg(kCAN_NWID_NODEID_txMsgIdx_TX_MESSAGE);

    /* Step 5: Confirm that message was successfuly transmited */
    if(tx_return == kNoError)
    {
        /* Message transmission accepted */
    }
    else
    {
        /* Message transmission neither accepted by CAN HAL nor
         * queued. Do manual retransmission attempt some time
         * later...*/
    }
}
```

CAN Data Structures

A data structure is generated for each message within file `cog_comgen_CAN_NWID_network.h`. The purpose of this structure is to facilitate the access of the signals defined in such message.

IMPORTANT: For all examples here following assumptions are made:

- Target MCU is little endian (big endian is currently not supported).
- Compiler is set in a way that the structures are "packed", this means no memory alignment is performed between fields but rather the compiler reduces the structure size as much as possible. For example, this is achieved in `gcc` by using the compiler directive `__attribute__((packed))` after the `struct` keyword.

The data structure generated for each message is an `union` having on the one hand an `struct` (called `s`) defining the message's signals as fields of it according to their layout information and on the other hand a byte-array (named `all`) of length equal to the message's length in order to provide raw access to the message's data.

```
typedef union{
    struct __attribute__((packed)){
        /* signal(s) fields following layout */
    } s;
    uint8_t all[kCAN_NWID_msgLen_MESSAGENAME];
}S_MESSAGENAME;
```

Wildcard MESSAGENAME corresponds to the defined message name.

Canonical and Non-Canonical Signals/Messages

The generated structure `s` can be either a bitfield or a regular structure depending on the layout of the message's signals.

If all the signals of the message are *canonical* then a regular structure is generated (not a bitfield).

If at least one signal is *non-canonical* then a bitfield is generated within structure `s` to model the different signals.

A signal is defined as *canonical* if its starting bit is a multiple of 8 (meaning that the signal starts at an exact byte position) and if its length is also multiple of 8 (the signal length occupies full byte(s)). If this is not met, then the signal is considered *non-canonical*.

A *message* is canonical if all of its signals are canonical and is non-canonical if at least one of its signals is non-canonical.

Example of canonical message:

- Message:
  - Name: MESSAGE1
  - Length: 8 bytes
- MESSAGE1 signals:
  - Signal_11: (start bit = 0, start byte = 0, length = 8)
  - Signal_12: (start bit = 0, start byte = 1, length = 16)
  - Signal_13: (start bit = 0, start byte = 3, length = 8)
  - Signal_14: (start bit = 0, start byte = 4, length = 8)

The generated structure `s` for MESSAGE1 will look as follows:

```
    struct {
        uint8_t Signal_11;
        uint16_t Signal_12;
        uint8_t Signal_13;
        uint8_t Signal_14;
    } s;
```

Example of non-canonical message:

- Message:
  - Name: MESSAGE2

    - Length: 8 bytes
- MESSAGE2 signals:
    - Signal_21: (start bit = 0, start byte = 0, length = 8)
    - Signal_22: (start bit = 0, start byte = 1, **length = 7**)
    - Signal_23: (start bit = 0, start byte = 3, length = 8)
    - Signal_24: (start bit = 0, start byte = 4, length = 8)

Since signal Signal_22 is non-canonical, a bitfield will be generated. The resulting structure `s` for MESSAGE2 will look then as follows:

```
    struct {
        uint8_t Signal_21 : 8;
        uint8_t Signal_22 : 7;
        uint8_t reserved_0 : 1;
        uint8_t reserved_1 : 8;
        uint8_t Signal_23 : 8;
        uint8_t Signal_24 : 8;
        uint8_t reserved_2 : 8;
        uint8_t reserved_3 : 8;
        uint8_t reserved_4 : 8;
    } s;
```

Notice the inserted reserved fields which correspond to empty space in the message.

Signal Fragmentation

If a signal:

- Is non-canonical or,
- Is canonical but its length is greater than 8 bits **and** doesn't exactly match the size of the compiler's basic data types (8, 16, 32 or 64 bits for gcc compiler) then the signal will need to be **fragmented** within the structure *s*.

If a signal gets fragmented it can no longer be accessed by a single field within the structure `s` but the access will need to be performed individually per each generated fragment (or can be accessed as a whole via access *macros* explained in further sections).

Examples of non-canonical signals fragmentations:

- Message:
    - Name: MESSAGE3
    - Length: 8 bytes
- MESSAGE2 signals:
    - Signal_31: (**start bit = 1**, start byte = 0, length = 8)
    - Signal_32: (start bit = 0, start byte = 2, **length = 15**)

Signals Signal_31 and Signal_32 are non-canonical so they need to get fragmented. The resulting structure `s` for MESSAGE3 will look as follows:

```
    struct {
        uint8_t reserved_0 : 1;
        uint8_t Signal_31_0 : 7;
        uint8_t Signal_31_1 : 1;
        uint8_t reserved_1 : 7;
        uint8_t Signal_32_0 : 8;
        uint8_t Signal_32_1 : 7;
        uint8_t reserved_2 : 1;
        uint8_t reserved_3 : 8;
        uint8_t reserved_4 : 8;
        uint8_t reserved_5 : 8;
        uint8_t reserved_6 : 8;
    } s;
```

Notice that each signal fragment gets a suffix `_x` where x indicates the fragment number.

Example of canonical signals fragmentation:

- Message:
  - Name: MESSAGE4
  - Length: 8 bytes
- MESSAGE4 signals:
  - Signal_41: (start bit = 0, start byte = 0, **length = 24**)
  - Signal_42: (start bit = 0, start byte = 3, **length = 40**)

Signals Signal_41 and Signal_42 are canonical, however, their sizes do not match the size of a basic data type so they need to get fragmented. The resulting structure `s` for MESSAGE4 will look as follows:

```
    struct {
        uint16_t Signal_41_0;
        uint8_t Signal_41_1;
        uint32_t Signal_42_0;
        uint8_t Signal_42_1;
    } s;
```

Notice that each signal fragment gets a suffix `_x` where x indicates the fragment number and that no bit-field is generated since MESSAGE4 is a canonical message.
Note: the generator tries to reduce the number of fragments as possible trying to then chose the bigger possible fragment sizes.

If a signal is defined as an **array** (currently only byte-arrays are supported) and if its conveyor message is also canonical, then a regular array will be generated in structure `s` for such signal. Otherwise the array signal will get exploded into fragments corresponding to a byte each one.

Note: A signal of array type shall always be canonical, it is not allowed to define an array signal starting at a bit position not multiple of a byte or for it to have a length not multiple of 8.

Example of an array signal in a canonical message:

- Message:
  - Name: MESSAGE5
  - Length: 7 bytes
- MESSAGE5 signals:
  - Signal_51: (start bit = 0, start byte = 0, **length = 32**, **type = array**)

    - Signal_52: (start bit = 0, start byte = 4, length = 24, type = scalar)

Signals Signal_51 and Signal_52 are canonical, hence message is canonical. Signal Signal_52 needs to get fragmented. The resulting structure `s` for MESSAGE5 will look as follows:

```
struct {
    uint8_t Signal_51[4];
    uint16_t Signal_52_0;
    uint8_t Signal_52_1;
} s;
```

Example of an array signal in a non-canonical message:

- Message:
    - Name: MESSAGE6
    - Length: 7 bytes
- MESSAGE6 signals:
    - Signal_61: (start bit = 0, start byte = 0, **length = 32**, **type = array**)
    - Signal_62: (start bit = 0, start byte = 4, **length = 23**, type = scalar)

Signal_62 is non-canonical, hence message is non-canonical. A bitfield will be generated and therefore, array signal Signal_61 gets exploded into fragments. The resulting structure `s` for MESSAGE6 will look as follows:

```
struct {
    uint8_t Signal_61_0 : 8;
    uint8_t Signal_61_1 : 8;
    uint8_t Signal_61_2 : 8;
    uint8_t Signal_61_3 : 8;
    uint8_t Signal_62_0 : 8;
    uint8_t Signal_62_1 : 8;
    uint8_t Signal_62_2 : 8;
} s;
```

The other element of the message's `union` is just a simple array of bytes named `all`. This is generated so that the user can modify the signals in a raw manner if desired.

CAN Access Macros

Signals Access Macros

A set of access macros are generated for each signal in the network within file `cog_comgen_CAN_NWID_network.h`. These macros provide alternative means of accessing the signals (reading and writing) of the messages directly from their raw data instead of using the `s` structures defined in previous section.

If for some reason, the generated `s` structure fields don't get properly packed then the macros can be used as alternatives since they do not depend on compiler's struct logic. Instead, the macros use type casting, masking and shifting's in order to access the signals.

Another advantage of using these macros is that they do not face signal fragmentation situation. Meaning that regardless of the type of signal (except for array signals) a single macro can fully access it.

At the end is users decision to choose which means of accessing the signals fits better its needs either by using the `s` structure, the `all` array or the access macros.

"Extract" signal macros (read from any provided array)

Extract macros are generated for all non-array signals. These macros have the following naming convention:

`#define CAN_NWID_extract_SIGNALNAME(msg_buffer)`

The macro argument `msg_buffer` is a pointer to an arbitrary array of bytes. The intention, however, is that this array has the same length as the signal's conveyor message and that the provided msg_buffer points always to the byte 'zero' of the array.

Example:

- Message:
    - Name: MESSAGE7
    - Length: 3 bytes
- MESSAGE7 signals:
    - Signal_71: (start bit = 0, start byte = 0, length = 8)
    - Signal_72: (start bit = 0, start byte = 1, length = 7)
    - Signal_73: (start bit = 0, start byte = 2, length = 1)

Three extract macros will be generated:

`#define CAN_NWID_extract_Signal_71(msg_buffer)`

`#define CAN_NWID_extract_Signal_72(msg_buffer)`

`#define CAN_NWID_extract_Signal_73(msg_buffer)`

A typical usage will imply defining a byte array representing the received MESSAGE7 raw data:

`uint8_t MESSAGE7_data[3];`

This array `MESSAGE7_data[3]` is expected to be updated with the raw received data. Once this is done, the extract macros can be used to read the individual signals from this array.

```
void some_app_function(void)
{
  uint8_t MESSAGE7_data[3]; /* Local array for raw data of MESSAGE7 */
  uint8_t my_Signal_71; /* Local variable for signal Signal_71 */
  uint8_t my_Signal_72; /* Local variable for signal Signal_72 */
  uint8_t my_Signal_73; /* Local variable for signal Signal_73 */

  /* Code for updating MESSAGE7_data with the received data from CAN. */
  /* Refer to further sections for information about this operation. */
  /* ... */

  /* Extracting signals from MESSAGE7_data array */
  my_Signal_71 = CAN_NWID_extract_Signal_71(MESSAGE7_data);
  my_Signal_72 = CAN_NWID_extract_Signal_72(MESSAGE7_data);
  // Also possible this way...
  my_Signal_73 = CAN_NWID_extract_Signal_73(&MESSAGE7_data[0]);

  /* Application code doing something with signals goes here */
  /* ... */
}
```

"Get" signal macros (read from message's `union`)

Get macros are generated for all non-array signals. These "get" macros are similar to the "extract" macros but instead of providing an arbitrary array, they expect the message's union as argument and they operate over the `all` array.

These macros have the following naming convention:

`#define CAN_NWID_get_SIGNALNAME(msg_buffer)`

The macro argument `msg_buffer` is a pointer to the conveyor's message union `S_MESSAGENAME`. These macros indeed use the extract macros but they pass the `msg_buffer.all` pointer to them:

`#define CAN_NWID_get_SIGNALNAME(msg_buffer)` `(CAN_NWID_extract_SIGNALNAME(msg_buffer.all))`

Usage Example:

- Message (same message 7 as used before):
  - Name: MESSAGE7
  - Length: 3 bytes
- MESSAGE7 signals:
  - Signal_71: (start bit = 0, start byte = 0, length = 8)
  - Signal_72: (start bit = 0, start byte = 1, length = 7)
  - Signal_73: (start bit = 0, start byte = 2, length = 1)

Three get macros will be generated:

`#define CAN_NWID_get_Signal_71(msg_buffer)`

`#define CAN_NWID_get_Signal_72(msg_buffer)`

`#define CAN_NWID_get_Signal_73(msg_buffer)`

A message's `S_MESSAGENAME` `union` is expected to be defined in this case. The raw received data needs to get updated into the `union` and then the get macros can be used over it.

```
void some_app_function(void)
{
  S_MESSAGE7 MESSAGE7_union; /* Local union for the data of MESSAGE7 */
  uint8_t my_Signal_71; /* Local variable for signal Signal_71 */
  uint8_t my_Signal_72; /* Local variable for signal Signal_72 */
  uint8_t my_Signal_73; /* Local variable for signal Signal_73 */

  /* Code for updating MESSAGE7_union with the received data from CAN. */
  /* Refer to further sections for information about this operation. */
  /* ... */

  /* Getting signals from MESSAGE7_union array */
  my_Signal_71 = CAN_NWID_get_Signal_71(MESSAGE7_union);
  my_Signal_72 = CAN_NWID_get_Signal_72(MESSAGE7_union);
  my_Signal_73 = CAN_NWID_get_Signal_73(MESSAGE7_union);

  /* Application code doing something with signals goes here */
  /* ... */
}
```

Regardless of the signal type, a single macro is generated which will provide all the signal's data. Hence, the local data getting the signal's data shall be long enough to accomodate the full signal's data.

Gcc compiler provides basic data up-to 64-bits (`long long int`) so then any CAN signal (which can't never exceed 64-bits) can be accessed with a single macro statement.

"Get Pointer" macros for array-signals (read)

For array-signals, "get pointer" macros are generated. These macros have the following naming convention:

`#define CAN_NWID_get_ptr_SIGNALNAME(msg_buffer)`

The macro argument `msg_buffer` is a pointer to the conveyor's message union `S_MESSAGENAME`. These macros simply return the pointer to the corresponding starting byte of the signal in the `all` array of the union (taking into consideration the signal's defined start byte).

The user can then operate over the signal as if it was a normal byte-array.

Example:

- Message:
  - Name: MESSAGE8
  - Length: 5 bytes
- MESSAGE8 signals:
  - Signal_81: (start bit = 0, start byte = 0, length = 16, type = scalar)
  - Signal_82: (start bit = 0, start byte = 2, length = 24, **type = array**)

Following get pointer macro will be generated for the array-signal Signal_82:

`#define CAN_NWID_get_ptr_Signal_81(msg_buffer)`

A message's `S_MESSAGENAME` `union` is expected to be defined in this case. The raw received data needs to get updated into the `union` and then the get pointer macro can be used over it.

```
void some_app_function(void)
{
  S_MESSAGE8 MESSAGE8_union; /* Local union for the data of MESSAGE8 */
  uint8_t my_Signal_81; /* Local variable for signal Signal_81 */
  uint8_t * my_Signal_82; /* Local pointer variable for signal Signal_82 */

  /* Code for updating MESSAGE8_union with the received data from CAN. */
  /* Refer to further sections for information about this operation. */
  /* ... */

  /* Getting signal pointer for Signal_82 from MESSAGE8_union */
  my_Signal_82 = CAN_NWID_get_ptr_Signal_82(MESSAGE8_union);

  /* Application code doing something with signal goes here */
  if(my_Signal_82[1] == 0xFF)
  {
      /* Do something if the second byte of the array Signal_81 is 0xFF */
    /* ... */
  }
}
```

"Write" signal macros (write to any provided array)

Write macros are generated for all non-array signals. These macros have the following naming convention:

`#define CAN_NWID_write_SIGNALNAME(msg_buffer,data)`

The macro argument `msg_buffer` is a pointer to an arbitrary array of bytes. The intention, however, is that this array corresponds to the message data containing the signal to be written. Hence, the array is expected to have the same length as the conveyor message and the provided macro argument `msg_buffer` shall always point to the byte 'zero' of such array regardless of the signal's starting byte (the macro will accommodate for this).

The macro argument `data` shall be the value of the signal to be written. The variable or constant provided in the `data` argument shall be long enough to contain all the desired value to be written.

Usage Example:

- Message:
  - Name: MESSAGE9
  - Length: 8 bytes
- MESSAGE9 signals:
  - Signal_91: (start bit = 0, start byte = 0, length = 1)
  - Signal_92: (start bit = 1, start byte = 0, length = 4)
  - Signal_93: (start bit = 0, start byte = 1, length = 40)

Three write macros will be generated:

`#define CAN_NWID_write_Signal_91(msg_buffer, data)`

`#define CAN_NWID_write_Signal_92(msg_buffer, data)`

`#define CAN_NWID_write_Signal_93(msg_buffer, data)`

A typical usage will imply defining a byte array representing the MESSAGE9 raw data to be written (and transmitted since writing signals is only allowed for transmitted data):

`uint8_t MESSAGE9_data[8];`

This array `MESSAGE9_data[8]` is expected to be updated with the user's desired signal values and then this data needs to be synchronized to the transmission buffer so that it can be transmitted in a CAN message.

```
void some_app_function(void)
{
  uint8_t MESSAGE9_data; /* Local array for the data of MESSAGE9 */
  uint8_t my_Signal_91; /* Local variable for signal Signal_91 */
  /* Signal Signal_92 will be updated with a constant. */
  uint64_t my_Signal_93; /* Local variable for signal Signal_93 */
  /* Variable my_signal_94 is of 64-bit so that it can contain the 40-bits
   * required for writting signal Signal_94. */

  /* Setting a temporal value for Signal_91 */
  my_Signal_91 = 1u;
  /* Writting Signal_91 in the local message data */
  CAN_NWID_write_Signal_91(MESSAGE9_data, Signal_91);
  /* Writting Signal_92 directly with constant value 0x03 */
  CAN_NWID_write_Signal_92(MESSAGE9_data, 0x03);

  /* Setting a temporal value for Signal_93 */
  my_Signal_93 = 0x000000FFFFFFF123ull;
  /* Writting Signal_93 in the local message data */
  CAN_NWID_write_Signal_93(MESSAGE9_data, my_Signal_93);
  /* Bits 41 to 64 will be ignored since Signal_93 is of 40-bits length */

  /* Code for updating transmission buffer for MESSAGE9 based on MESSAGE9_data.
*/
  /* Refer to further sections for information about this operation. */
  /* ... */
}
```

"Update" signal macros (write to a message's `union`)

Update macros are generated for all non-array signals. These "update" macros are similar to the "write" macros but instead of writing to an arbitrary array, they will write into a message's union `all` array.

These macros have the following naming convention:

`#define CAN_NWID_update_SIGNALNAME(msg_buffer, data)`

The macro argument `msg_buffer` is a pointer to the conveyor's message union `S_MESSAGENAME`.

The macro argument `data` shall be the value of the signal to be written.

These macros indeed use the write macros but they pass the `msg_buffer.all` pointer to them (and the `data` macro argument):

`#define CAN_NWID_update_SIGNALNAME(msg_buffer, data)`
`CAN_NWID_write_SIGNALNAME(msg_buffer.all, data)`

Usage Example:

- Message:
  - Name: MESSAGE9
  - Length: 8 bytes
- MESSAGE9 signals:
  - Signal_91: (start bit = 0, start byte = 0, length = 1)
  - Signal_92: (start bit = 1, start byte = 0, length = 4)
  - Signal_93: (start bit = 0, start byte = 1, length = 40)

Three update macros will be generated:

`#define CAN_NWID_update_Signal_91(msg_buffer, data)`

`#define CAN_NWID_update_Signal_92(msg_buffer, data)`

`#define CAN_NWID_update_Signal_93(msg_buffer, data)`

A typical usage will imply defining a local S_MESSAGE9 union for containing the temporal raw data to be written:

`S_MESSAGE9 MESSAGE9_union;`

This union `MESSAGE9_union` is expected to be updated with the user's desired signal values and then this data needs to be synchronized to the transmission buffer so that it can be transmitted in a CAN message.

```
void some_app_function(void)
{
  S_MESSAGE9 MESSAGE9_union; /* Local array for the data of MESSAGE9 */
  uint8_t my_Signal_91; /* Local variable for signal Signal_91 */
  /* Signal Signal_92 will be updated with a constant. */
  uint64_t my_Signal_93; /* Local variable for signal Signal_93 */
  /* Variable my_signal_94 is of 64-bit so that it can contain the 40-bits
   * required for writting signal Signal_94. */

  /* Setting a temporal value for Signal_91 */
  my_Signal_91 = 1u;
  /* Writting Signal_91 in the local message data */
  CAN_NWID_update_Signal_91(MESSAGE9_union, Signal_91);

  /* Writting Signal_92 directly with constant value 0x03 */
  CAN_NWID_update_Signal_92(MESSAGE9_union, 0x03);

  /* Setting a temporal value for Signal_93 */
  my_Signal_93 = 0x000000FFFFFFF123ull;
  /* Writting Signal_93 in the local message data */
  CAN_NWID_update_Signal_93(MESSAGE9_union, my_Signal_93);
  /* Bits 41 to 64 will be ignored since Signal_93 is of 40-bits length */

  /* Code for updating transmission buffer for MESSAGE9 based on MESSAGE9_union.
*/
  /* Refer to further sections for information about this operation. */
  /* ... */
}
```

"Update Pointer" macros for array-signals (write)

For array-signals, "update pointer" macros are generated. These macros have the following naming convention:

`#define CAN_NWID_update_ptr_SIGNALNAME(msg_buffer, data)`

The macro argument `msg_buffer` is a pointer to the message union `S_MESSAGENAME`.

The macro argument `data` shall be the pointer to the array containing the data to be written.

The "update pointer" macro does copy from the `data` array to the `S_MESSAGENAME.all` array.

Example:

- Message:
    - Name: MESSAGE10
    - Length: 6 bytes
- MESSAGE8 signals:
    - Signal_101: (start bit = 0, start byte = 0, length = 8, type = scalar)
    - Signal_102: (start bit = 0, start byte = 0, length = 40, **type = array**)

Following update pointer macro will be generated for the array-signal Signal_102:

`#define CAN_NWID_update_ptr_Signal_102(msg_buffer, data)`

A message's `S_MESSAGENAME` `union` is expected to be defined in this case. The raw received data needs to get updated into the `union` and then the get pointer macro can be used over it.

```
void some_app_function(void)
{
  S_MESSAGE10 MESSAGE10_union; /* Local union for the data of MESSAGE10 */
  uint8_t my_Signal_101[5]; /* Local variable for array signal Signal_101 */

  /* Code for updating MESSAGE8_union with the received data from CAN. */
  /* Refer to further sections for information about this operation. */
  /* ... */

  /* Assign some application values to the local signal */
  my_Signal_101[0] = 0x00;
  my_Signal_101[1] = 0x01;
  my_Signal_101[2] = 0x02;
  my_Signal_101[3] = 0x03;
  my_Signal_101[4] = 0x04;

  /* Updating to the message union */
  CAN_NWID_update_ptr_Signal_101(&MESSAGE10_union, my_Signal_101);
}
```

Signals Direct Access Macros

Macros for having direct access to the data in the TX and RX unified buffer are also generated. As their name suggests, these macros do read/write data directly over the unified buffers without any critical section protection provided by default.

These are useful for example for reading data directly from within the reception callbacks (via "get direct" macros). Since these callbacks are typically called within ISR context, accessing data from the RX unified buffer also occurs in that ISR context and, hence, will be presumably done with interrupts disabled avoiding data corruption problems.

Application can freely use these direct access macros provided that the proper care is taken to avoid data corruption (for example by using the direct access macros within critical sections).

Another use case can be to quickly modify (via an "update direct" macro) a single signal to be transmitted without needing to perform all the suggested procedure in section "Accessing Transmission Data". Again, proper care shall be taken to perform this access within a critical section and also it shall be taken into account that directly writing to multiple signals in the same message could lead to the transmission of *partially* updated messages if those are *cyclic* or *cyclic_spontan*.

"Get Direct" macros (read directly from unified buffers)

Get Direct macros are generated for all non-array signals both for received and transmitted signals. These "get direct" macros retrieve the data for the given signal directly from the respective RX or TX unified buffer. These macros have the following naming convention:

`#define CAN_NWID_NODEID_get_direct_SIGNALNAME()`

The macro returns the value of the signal from the corresponding unified buffer. Hence, the variable data getting this returned signal's data shall be long enough to accommodate the full signal. Gcc compiler provides basic data up-to 64-bits (`long long int`) so then any CAN signal (which can't never exceed 64-bits in standard CAN) can be accessed with a single macro statement.

Usage Example:

- Message:
    - Name: MESSAGE11
    - Length: 5 bytes
- MESSAGE11 signals:
    - Signal_111: (start bit = 0, start byte = 0, length = 8)
    - Signal_112: (start bit = 0, start byte = 1, length = 32)

Three get macros will be generated:

`#define CAN_NWID_NODEID_get_direct_Signal_111()`

`#define CAN_NWID_NODEID_get_direct_Signal_112()`

```
void some_app_function(void)
{
  uint8_t my_Signal_111; /* Local variable for signal Signal_111 */
  uint32_t my_Signal_112; /* Local variable for signal Signal_112 */
  FlagsNative msg_avlbl_flags; /* variable for containing msg-available flags */

  /* poll for the reception of message "RX_MESSAGE" */
  /* ---------------------------------------------- */
  /* Get msg-available flags */
  msg_avlbl_flags = CAN_NWID_NODEID_get_msg_avlbl_flags_MESSAGE11();
  if(msg_avlbl_flags.all)
  {
      /* Message has been received! */
      /* Consume (clear) only message-available flags */
      CAN_NWID_NODEID_clr_msg_avlbl_flags_MESSAGE11();

      /* Read signals directly from RX unified buffer.
       * Access is done within critical section. */
      CALVOS_CRITICAL_ENTER();
      my_Signal_111 = CAN_NWID_NODEID_get_direct_Signal_111();
      my_Signal_112 = CAN_NWID_NODEID_get_direct_Signal_112();
      CALVOS_CRITICAL_EXIT();

      /* Do something with local signal's data */
      if(my_Signal_111 > 3)
      {
          /* Do something if received Signal_111 is greather than 3... */
      }

      /* More application code goes here... */
  }
}
```

"Update Direct" macros (write directly to TX unified buffer)

Update Direct macros are generated for all non-array transmission signals. These "update direct" macros write the data for the given signal directly to the TX unified buffer. "Update direct" macros are not applicable for reception signals since received signals shouldn't be updated by the user but directly by the CAN layer upon receiving them.

These "update direct" macros have the following naming convention:

`#define CAN_NWID_NODEID_update_direct_SIGNALNAME(data_in)`

The macro argument "data_in" is the data to be written to the TX unified buffer.

Usage Example:

- Message:
  - Name: MESSAGE12
  - Message direction: transmission
  - Message transmission type: spontan
  - Length: 5 bytes
- MESSAGE12 signals:
  - Signal_121: (start bit = 0, start byte = 0, length = 8)
  - Signal_122: (start bit = 0, start byte = 1, length = 32)

Three get macros will be generated:

`#define CAN_NWID_update_direct_Signal_121()`

`#define CAN_NWID_update_direct_Signal_122()`

```
void some_app_function(void)
{
  uint8_t my_Signal_121; /* Local variable for signal Signal_121 */
  uint32_t my_Signal_122; /* Local variable for signal Signal_122 */
  CalvosError tx_return; /* Hold the TX request returned value */

  /* Application code to determine local values for Signals Signal_121
   * and Signal_122 */
  my_Signal_121 = 0xDE;
  my_Signal_122 = 0xFE00AABB;

  /* Directly update TX unified buffer with local data */
  CALVOS_CRITICAL_ENTER();
  CAN_NWID_NODEID_update_direct_Signal_121(my_Signal_121);
  CAN_NWID_NODEID_update_direct_Signal_122(my_Signal_122);
  CALVOS_CRITICAL_EXIT();

  /* Request message 'MESSAGE12' transmission. If this message was of type
   * 'cyclic' then it will be automatically transmitted when its
   * period expires. */
  tx_return = can_NWID_NODEID_transmitMsg(kCAN_NWID_NODEID_txMsgIdx_MESSAGE12);
  if(tx_return == kError)
  {
      /* Message was not accepted for transmission by the CAN HAL and was
       * neither queued for automatic retransmission. Application needs
       * to manually trigger its transmission again. Probably some
       * milliseconds later...*/
  }
}
```