# ComGen.CAN

# - Usage and Integration Manual -

## #00000000

| | |
|---|---|
| Primary Author(s): | Carlos Calvillo |
| Version: | 0.1.0 |
| Maturity: | draft |
| Project: | Project Name |
| Document ID: | #00000000 |
| Additional Information: | |

# Table of Contents

# 1. Document History

| Version | Date (dd/Mmm/yyyy) | Author(s) | Description |
|---|---|---|---|
| 0.1 | 08/Feb/2021 | Carlos Calvillo | Initial version. |

# 2. 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.

# 3. CAN Network definition

The CAN network to generate needs to be defined first using the "CAN Network Definition.ods" template. 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.

## 3.1 Network definitions

General network definitions need to be speciifed 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 speciifed 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 recomended 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 speciifed 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. 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 the message. It shall be one of the node names defined in tab "Network_and_Nodes".
- **Subscribers:** Node or nodes subscribing to the message. Timeout in milliseconds for each subscribing node can be specified within parentheses. Shall be a comma sepparated list of node names (within 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 transmissiont. 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 speciifed 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 a user-defined enumerated data type (see section below).
- **Length (bits):** length in bits of the signal. Can be betwenn 1 and upto the message size (e.g., 64 bits for an 8-byte length message). The signal must fit in the message available space (space not occipied by other message's 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 bitposition of the signal within the conveyor message.

  **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 will all zeros).
- **Fail Safe value:** value fo the signal to be set when the conveyor message timesout (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.

- **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-idenfitier syntax and shall be *unique* amognst 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 4 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 sepparated.

  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.

**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.

# 4. 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 "*project.xml*":

   *C:\my_calvos_project\project.xml*

   An empty project is an XML file with following contents:

   A codegen component "usr_in/Code_Generation_Setup.ods" Value "usr_in/CAN_Network_Definition.ods" "usr_in/Utilities_Definitions.ods" INFO calvOS Project 1.0.0

```
  <CalvosProject>
    <Name>Any user defined project name</Name>
    <Desc>Any user defined project description</Desc>
    <Version>Any user defined project version</Version>
    <Date>Any user defined project date</Date>
    <AutoDate></AutoDate>
    <AutoFile></AutoFile>
    <Components>
    </Components>
    <Params>
        <Param id="log_level">INFO</Param>
    </Params>
    <Metadata>
        <TemplateName>calvOS Project</TemplateName>
        <TemplateVersion>1.0.0</TemplateVersion>
        <TemplateDesc/>
    </Metadata>
</CalvosProject>
```

3. Create a sub-foler 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 *project.xml* and set its input file information:

   - Create the follwing XML `Component` node within the `Components` node in *project.xml*:
   - 
```
<Component type="comgen.CAN">
   <Name></Name>
   <Desc/>
   <Input type="ods">"usr_in/CAN_Network_Definition.ods"</Input>
   <Params>
   </Params>
</Component>
```

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

```
<CalvosProject>
   <Name>Any user defined project name</Name>
   <Desc>Any user defined project description</Desc>
   <Version>Any user defined project version</Version>
   <Date>Any user defined project date</Date>
   <AutoDate></AutoDate>
   <AutoFile></AutoFile>
   <Components>
       <Component type="comgen.CAN">
           <Name></Name>
           <Desc/>
           <Input type="ods">"usr_in/CAN_Network_Definition.ods"</Input>
           <Params>
           </Params>
       </Component>
   </Components>
   <Params>
       <Param id="log_level">INFO</Param>
   </Params>
   <Metadata>
       <TemplateName>calvOS Project</TemplateName>
       <TemplateVersion>1.0.0</TemplateVersion>
       <TemplateDesc/>
   </Metadata>
</CalvosProject>
```

Our project is now ready to be processed.

# 5. Processing Calvos Project

In order to generate the source code for our CAN network we need to invoke the calvos-engine which is a python package called "calvos".

Calvos-engine needs to be invoked from a console accepting following command-line arguments:

| Argument | Usage |
|---|---|
| -h, --help | show help message and exit |
| -p PROJECT, --project PROJECT | Mandatory. PROJECT: Full path with file name of the calvos project to be processed |
| -c CALVOS, --calvos CALVOS | CALVOS: Path where the calvos python package is located. If not provided, will look from installed python packages. |
| -l LOG_LEVEL, --log LOG_LEVEL | Logging level LOG_LEVEL: 0 - Debug, 1 - Info, 2 - Warning, 3 - Error. Default is 1 - Info. |
| -V, --version | show program's version number and exit |
| -v, --ver | show program's version number and exit |

Currently, `calvos` is not yet properly packaged so it needs to be set-up before use it.

## 5.1 Setting-up calvos-engine

1. Install and set-up python 3.7 or newer.

   Note: for simplicity we will assume that we are in a MS Windows environment and that the installed python 3.7 or newer can be invoked directly from the console, i.e., the python executable path is defined in the PATH system variable.

2. Get following dependencies (for example using `pip -m install`):

   - cogapp
   - lxml
   - pyexcel
   - pyexcel-ods

3. Get the `calvos` python source code from github https://github.com/calcore-io/calvos and put in *C:\calvos*

4. Open a command line and navigate into folder *C:\calvos\calvos-engine*

5. Run command `python -m calvos -v`

6. A legend similar to `calvos v0.1.0` shall be displayed.

# Generating the Source Code from project

In order to generate our project we need to run the following command from *C:\calvos\calvos-engine*:

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

If everythin wen't 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/erorrs 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 dependancies 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.

# 6. 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. |

# 7. 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.

## 7.1 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. |

## 7.2 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 flie referred as `comgen_CAN_NWID_network.h` corresponds to a generated file with name `comgen_CAN_C_network.h`.

## 7.3 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 flie referred as `comgen_CAN_NWID_NODEID_node_network.h` corresponds to a generated file with name `comgen_CAN_C_NODE_1_node_network.h`.

## 7.4 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 disntinguish 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 appliced (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 optmization 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.

# 8. Integration in 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).

## 8.1 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.

## 8.2 HAL Integration

For each node, a HAL integration is required for the reception and also for the transmission part of the code.

### 8.2.1 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. Function `can_NWID_NODEID_HALreceiveMsg()` needs to be invoked within the HAL function that signals the reception of a CAN message for the associated 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 suscribed 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 wihin 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_NODE_1_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. Invoke callback for message transmission confirmation from HAL. Function `can_CT_NODE_1_HALconfirmTxMsg()` needs to be called within the HAL function that signals the confirmation of the latest CAN message transmission. This function does't require any argument.

# Application Usage

## 8.3 Data Structures

### 8.3.1 Message's data structures

### 8.3.2 Accesing signals within a message

## 8.4 Receiving Messages

### 8.4.1 Using the reception callbacks

### 8.4.2 Using the available flags (polling for received messages)

### 8.4.3 Using the received data

## 8.5 Transmitting Messages

### 8.5.1 Setting data to be transmitted

### 8.5.2 Transmission of periodic messages

### 8.5.3 Transmission of spontaneous messages

# 9. Functions References