User Manual

For S32K1XX MCAL Sample Application

Document Number: UMSAASR4.3R1.0.0
Rev. 1.2

Contents

Section number | **Title** | **Page**

Chapter 1
Revision History

Chapter 2
About this Manual

Chapter 3
Installation Steps

Chapter 4
Sample Application Example Description

Chapter 5
Building the Sample Application Example

Chapter 1
Revision History

Table 1-0. Revision History

Revision	Date	Author	Description
1.0	9.11.2018	Stefan Tataru	1.0.0 Release
1.1	15.12.2018	Stefan Tataru	1.0.1 Release

Chapter 2
About this Manual

This User Manual describes utilization of the sample application for S32K2XX microcontroller with Autosar MCAL 4.3 version EAR 0.8.1

2.1 Acronyms and Definitions

Table 2-1. Acronyms and Definitions

Abbreviation / Acronym	Description
DIO	Digital Input Output Driver
PORT	Port Driver
BSW	Basic Software
ADC	Analog Digital Converter
FEE	Flash EEPROM Emulation
DEM	Diagnostic Event Manager
DET	Development Error Tracer
ECU	Electronic Control Unit
ISR	Interrupt Service Routine
OS	Operating System
GUI	Graphical User Interface
API	Application Programming Interface
EcuM	ECU state Manager
WDG	Watchdog Driver
PLL	Phase Lock Loop
LED	Light Emitting Diode
PB Variant	Post Build Variant
LT Variant	Link Time Variant
PC Variant	Pre Compile Variant

Reference List

2.2 Reference List

Table 2-2. Reference List

#	Items	Version
1	S32K2xx Reference Manual	S32K2xx_RM_Rev1DraftJ

Chapter 3
Installation Steps

3.1 Hardware Installation

The hardware installation describes setup of the Evaluation Board.

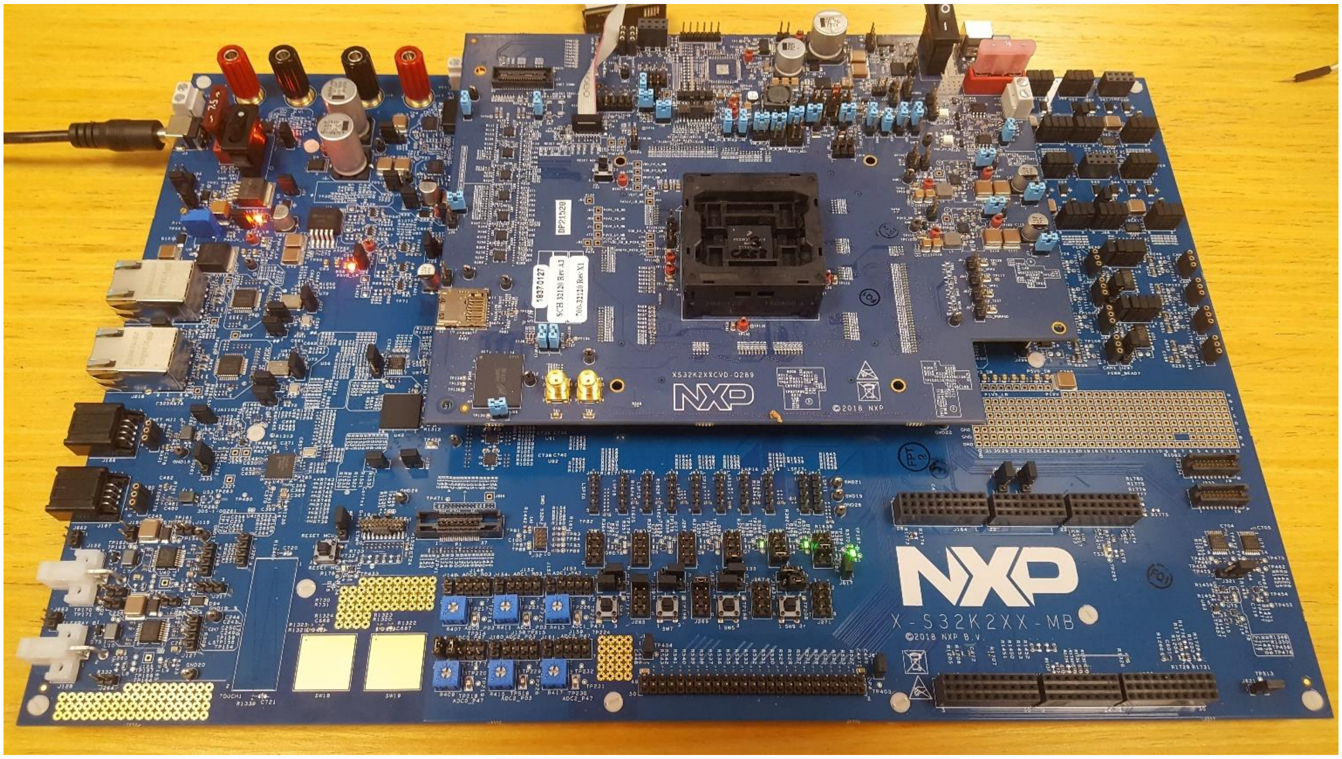


Figure 3-1. SCH-31431 Rev B Mother Board

SCH -32120 Rev A3 Daughter Card

3.2 Software Installation

Please install the MCAL package on your computer. The Integration Framework Sample Application package is delivered as a MCAL-type plugin: *IntegrationFramework_TS_T40D17M8I1R0*

The Application plugin has the following folder structure:

Folder or file	Description
-autosar	contains the IntegrationFramework.epd file
-auxiliary	contains files and folders required to build and start the framework application
-build	Contains the cmm folder and the batch files required to start build system
-bin subfolder	generated object files and linker output files are stored into this folder
-cmm subfolder	contains Lauterbach T32 cmm script files
-toolchains subfolder	contains linker-scrips folder, make folder and startup folder
-linkfiles subfolder	contains linker-scrips files
-make folder	contains make-files needed to build the application for all available CPU cores and compilers
-startup folder	contains source files and headers needed to start the application (startup code and interrupt vector definitions for each CPU type and available compiler)
-config folder	contains the configuration XDM template file for EB tresos.
-generate_PC	contains the configuration generation templates
- src folder	contains the source code files for all components of the framework application
- include folder	contains header files for all components of the framework application
- makefile file	the framework application makefile
- make.bat file	launches the make command
- launch.bat file	contains path to the Tresos Studio installation and launches the make.bat file
- Tresos folder/workspace	contains the Tresos project with the application configuration

Note: Since the application framework is NOT production code it is not delivered in the same plugins folder as the rest of the MCAL drivers. In order to build and run the application the user must copy the application plugin folder *"IntegrationFramework_TS_T40D17M8I1R0"* in the same folder where the rest of the MCAL plugins are located.

3.2.1 Tresos Project Installation

The following procedure requires that the user has EB Tresos Studio installed.

Procedure:

1. Make sure that all MCAL plugins are already installed in the Tresos Studio pluginsdirectory
2. Open Tresos Studio
3. Import Sample application project

a. Click on "File" and select "Import"

b. Select "Existing Projects into Workspace" and click on "Next" button as shown in Figure 4-3. Import Window - the First View

c. Next steps are depicted in Figure 4-4. Import Window - the Second View

- Select "Select root directory" and click on "Browse"
- Select the location of the [project] folder in the installed Sample application packagefolder (Tresos/workspace/[project])
- Select "Copy projects into workspace"
- Click on "Finish" button

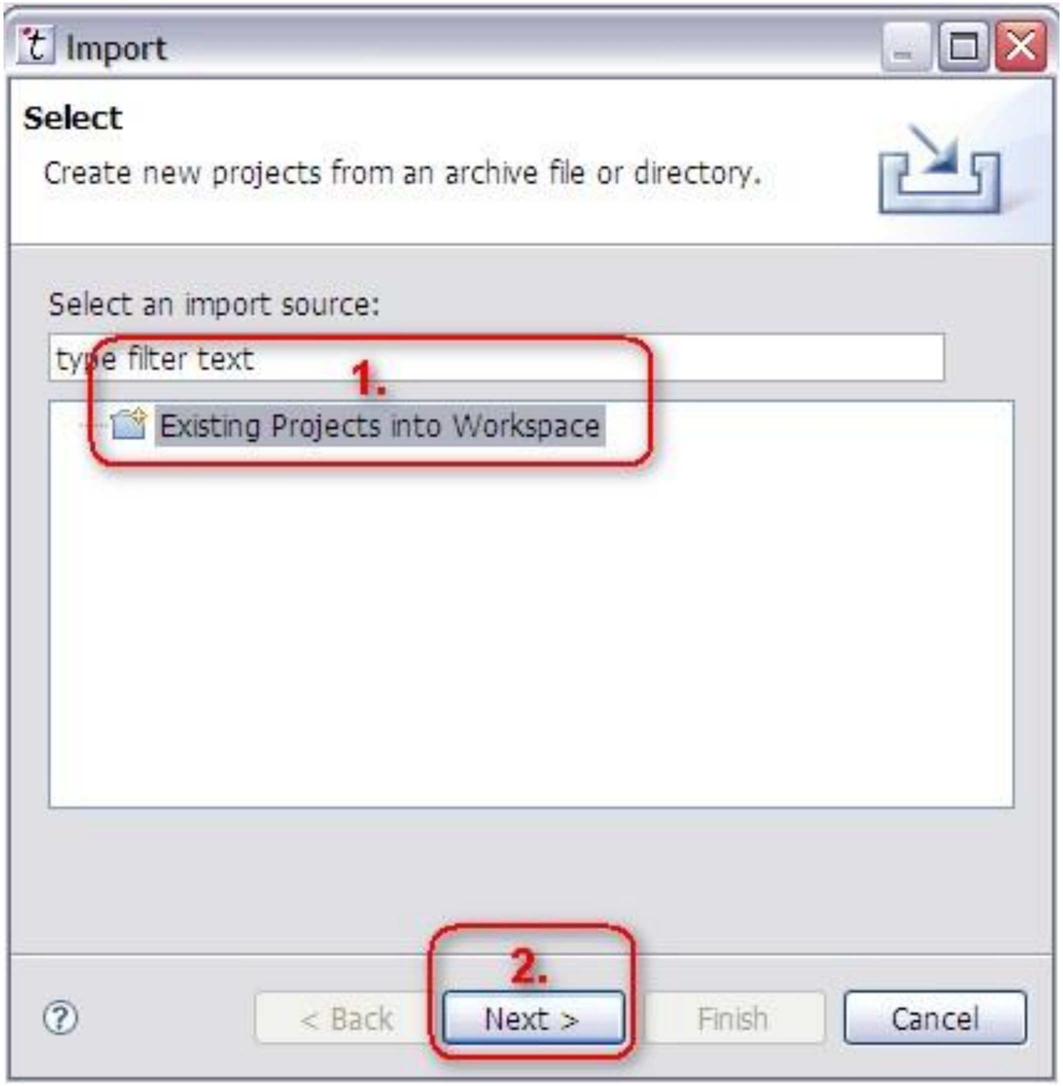


Figure 3-3. Import Window - the First View

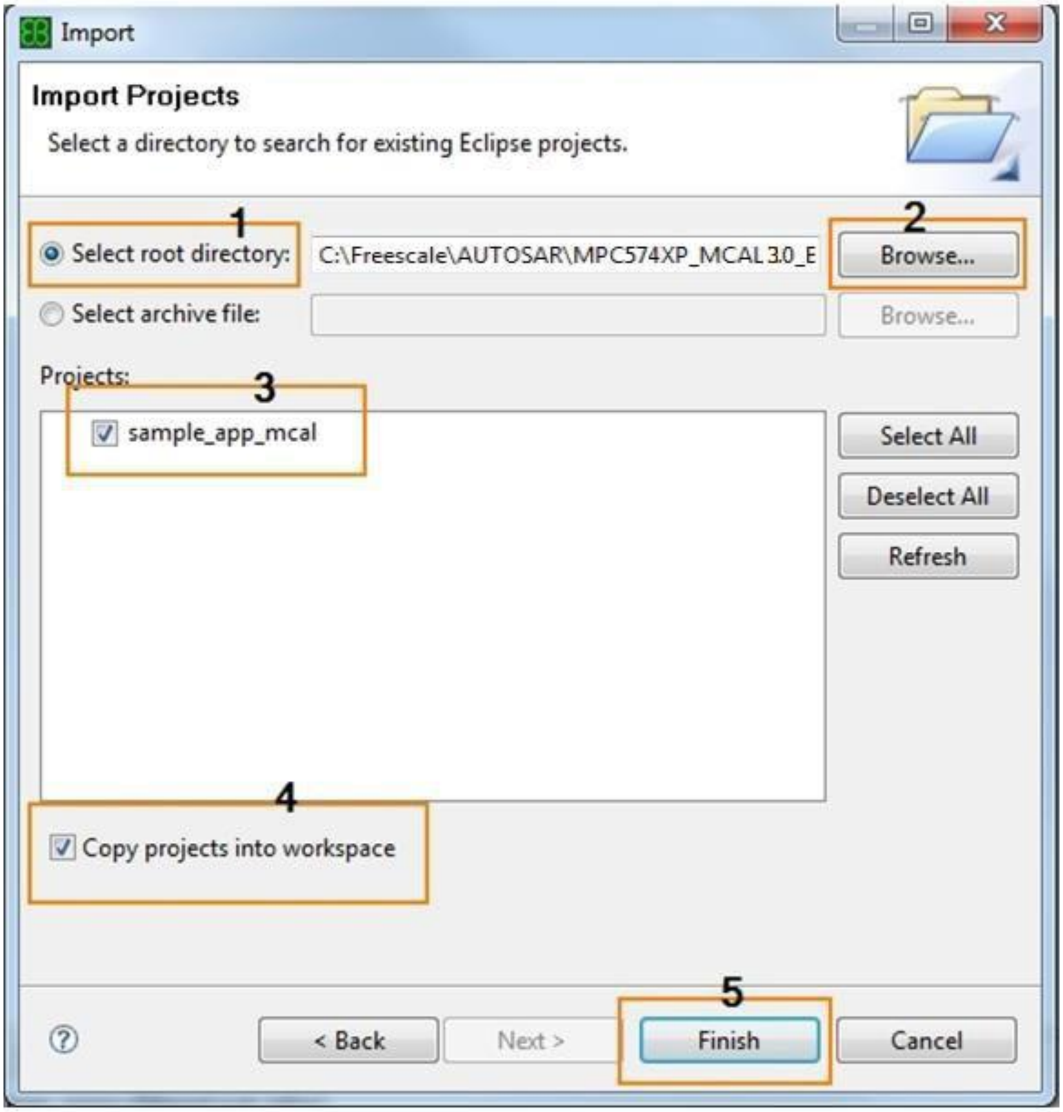


Figure 3-4. Import Window - the Second View

3.2.2 MCAL Application Configuration

The following procedure requires that the user has EB Tresos Studio installed and the toolchains versions specified in the MCAL Release Notes.

The toolchain that will be used needs to be installed for correct operation and the path to the installation location shall be added into the system environment variable(s):

- GHS_DIR

  Ex: SET GHS_DIR= C:/tools/ghs/ARM_MULTI_7.1.4COMPILER_2017.1.4

- GCC_DIR

  Ex: SET LINARO_DIR= C:/tools/GCC/gcc-6.3-arm32-eabi

- IAR_DIR

  Ex: SET IAR_DIR= C:/tools/IARSystem/EmbeddedWorkbench8.0/arm

- TRESOS_DIR for setting up the path to installed EB Tresos folder

  Ex: TRESOS_DIR=C:/Tools/EB/v24.0.1

- PLUGINS_DIR for defining the path to all source file to be build

  Ex: SET PLUGINS_DIR=C:/Tools/EB/v24.0.1/plugins

- TRESOS_WORKSPACE_DIR for defining the path to all generated configuration files

  Ex: TRESOS_WORKSPACE_DIR=C:/Tools/EB/v25.0.0/workspace/

  lighting_S32K2XX_4.3_EAR 0.8.1 /output

Note The path to the toolchain must not contain spaces. In case the compiler is installed into a path with spaces, the variable must be set with the "short" folder name (8.3 version of the file name that can be displayed with dir /X in command prompt)

Procedure:

1. Open launch.bat file in a text editor and specify the EB Tresos Studio location in the TRESOS_DIR parameter as shown in Figure 4-5. Configuration of the Tresos Studio Location.

2. Make sure that installation location of the compiler is added in the system environment variable (GHS, GCC, IAR)

3. Setup the plugins folder location if the plugins are not installed in the Tresos plugins folder (PLUGINS_DIR)
4. Setup the workspace folder location plugins folder (TRESOS_WORKSPACE_DIR).

```
:: uncomment line below if you do not set TRESOS_DIR over environment
:: SET TRESOS_DIR=
:: SET GHS_DIR=
:: SET IAR_DIR=
:: SET LINARO_DIR=
:: SET PLUGINS_DIR=
```

Chapter 4
Sample Application Example Description

This application demonstrates an example of usage for the MCAL modules. It is not part of the production code deliverables

4.1 The application software functionality

Initializes MCU module

- Initializes PLL and configures it to 80MHz.
- Checks whether PLL is locked
- Activates the PLL clock to the MCU clock distribution

Initializes PORT module. Pins configuration is show in section PORT and DIO Modules - Pin Configuration and DioChannel Assignment for keys and leds, and in PORT Configuration excluding Leds and Keys

Initialize ADC driver.

Initialize the GPT driver. Gpt channel 0 (PIT_0_CH_RTI) is used to trigger the internal round-robin scheduler.

Initializes the PWM driver and Sample application specific data for this driver.

Initializes the OCU driver. In the configuration provided by the sample application framework the OCU driver is used to trigger ADC at user-configured moments of time.

Initializes Integration Framework components and starts internal round-robin scheduler. The internal scheduler

Once the scheduler is started the following tasks are executed cyclically:

- IO Driver Abstraction (IoDal) task is called every 10 ms.
- System Driver Abstraction (SysDal) task is called every 10 ms.
- Run Time Environment (Rte) task is called every 20 ms.
- Run Time Environment Initialization (Rte_Init) is called just once.

a. Basic Software - IO Driver Abstraction task

IODAL component main purpose is to handle and control all I/O capable drivers: ADC, DIO, OCU, ICU, PWM.

IODAL handles Digital I/O's and PWM channels by calling the underlaying drivers directly.

ADC Channels are handled by using SW trigger conversion which is executed at precise moments of time define by using a Time-trigger table in OCU. For each ADC group two time-events are required:

- Start conversion event: defined in configuration as a Time Trigger scheduling point
- Read results event: define in configuration as "conversion time"

Note. IoDal configuration is based around IO channel descriptors. For analog channels each descriptor is defined as a unique combination of an ADC group and an ADC channel; for digital/pwm channels each descriptor is define by a single dio/pwm channel.

b. Basic Software - System Driver Abstraction task

SYSDAL component main purpose is to implement core system functionalities (i.e. the internal round-robin scheduler, user interrupt enablement) and to implement the power-up and power-down sequences (similarly to ECUM from AUTOSAR).

User Interrupts and Power Modes are also, defined and configured in SysDal.

The round-robin scheduler is based around a GPT timer and is used to execute application functionality at predefined moments of time. (i.e. every 10 ms call IoDal_Main()…)

c. Basic Software - Communication Driver Abstraction task

COMDAL component main purpose is to handle and control all external communication capable drivers: CAN, LIN, UART (etc.)

For each driver the handling of the communication is done either in polling mode (CAN) or interrupt driven (CAN, UART) depending on the driver capabilities.

In case of CAN, the COMDAL also implements the CanIf functions (CanIf_TxConfirmation and CanIf_RxIndication)

d. **Lighting Application task**

Application task is used to process command inputs signals and calculate outputs values for the all output channels of each lighting instance using the following logic.

- If Output signal is PWM and If Input command signal is ANALOG_INPUT, the duty cycle value for all outputs for that application instance is proportional to the value read from the analog input (i.e. Analog value read from a Potentiometer)
- If Output signal is PWM and If Input command signal is DIGITAL_INPUT, the duty cycle value for all outputs for that application instance is increased by 12.5% every-time the digital input is set from LOW to HIGH. When the duty-cycle value exceeds 100% the duty-cycle is reset to 0% and the cycle will start from that. Basically every-time a button is pressed the duty is increased by 12.5% until it reaches 100%.
- If Output signal is DIGITAL_OUTPUT and If Input command signal is DIGITAL_INPUT, the output signal shall be toggled between LOW and HIGH every time the digital input is set from LOW to HIGH. Basically every-time a button is pressed the output is toggled between LOW and HIGH.

e. **Using the Virtual Data Router**

The Virtual Data Router (VDR) is a SW components (SWC) that has the main purpose to handle data messages between different SWCs. In a sense, VDR's scope is to act as a network manager and route data from either from host ECU to a remote location (i.e. other ECU or Host PC) and vice-versa.

❖ **VDR Communication protocol – RX handling.**

When used to exchange data over communication lines (UART, CAN etc), the VDR works by handling RX messages with a fixed format. Each RX Message is composed of 3 or more bytes of data, and each byte has a fixed meaning as described below.

Byte 0	***Byte 1***	***Byte 2***	***Byte 3***	***Byte 4***	***Byte 5***	***Byte 6***	***…***	***Byte n***
Message ID	*Message Type*	*Message Response ID*	*Instance Id/Data 0*	*Channel Id/Data 1*	*Data 2*	*Data 3*		*Data n-3*

Table 3-1. RX Message format

Message ID – Represent the unique ID given to each handled message for a given direction.

Note: It is possible for a RX and TX message to have ID with the same value.

Message Type – Represent the scope associated with that message. Basically, the Message type instructs how that message with be processed and what type of response will be given to that message (if a response is expected).

- REQUEST_STATUS (*Byte_1* = 1U): **VDR** has receive a request to send the status of a given application instance (indicated by *Byte_3*). Data shall be packed and routed to TX_Reply message with the id given in *Byte_2*.
- REQUEST_OUTPUT_UPDATE (*Byte_1* = 2U): **VDR** has receive a request to update output values of a given application instance (indicated by *Byte_3*). This message does not require a Reply.
- REQUEST_SYS_GO_SLEEP (*Byte_1* = 4U): **VDR** has receive a request to change the run-mode of the host ECU or print the value of the current mode. The action required is indicated by *Byte_3* had has the following parameters:
  - 'R' – requests ECU to execute a SW Reset.
  - 'S' – requests ECU to change run-mode to the value indicated by *Byte_4*.
  - 'P' – requests ECU to 'print' the current value of the run-mode. This value will be packed into the TX_Replay message with the id given by: *Byte_2*.
- REQUEST_SYS_CHANGE_HW_VARIANT (*Byte_1* = 8U): **VDR** has receive a request to change the HW variant configured of the host ECU or print the current variant ID which is in use. The action required is indicated by *Byte_3* had has the following parameters:
  - 'V' – requests ECU to change HW variant to the value indicated by *Byte_4*.
  - 'P' – requests ECU to 'print' the current value of the running HW variant. This value will be packed into the TX_Replay message with the id given by: *Byte_2*.
- REQUEST_FORWARD_MSG (Byte_1 = 16U): **VDR** has receive a request to forward a given message to a different host ECU (or PC). The request will be packed into the TX_Replay. message with the id given by: *Byte_2*.
- REQUEST_ PRINT_RTM_MEAS (Byte_1 = 32U): **VDR** has receive a request to 'print' the result from a RUN-TIME Measurement.

Message ID – Represent the unique ID of the TX_Replay message associated with the current RX Message ID.

Instance ID/Communication Type – Represent the unique ID of the TX_Replay message associated with the current RX Message ID. This byte is a Bit-wise-OR operation between Instance ID (bit5 .. bit 0) and Communication type (bit7 and bit 6).

When REQUEST_OUTPUT_UPDATE was received for a “Lighting” instance that uses COM as method of changing the output value, the VDR will update the internal output buffers of the given instance as following:

- For a single output channel (with the id given by *Byte_4*) if *bit7* and *bit6* are ‘0’.
- For a set of channels (with the mask value given by *Byte_4*) if *bit7* = ‘0’ and *bit6* = ‘1’
- For all channels of the given instance if *bit7* = ‘1’ and *bit6* = ‘0’.

Channel ID Byte. – Only used in certain cases. See above for details

❖ VDR Communication protocol – TX data format.

VDR will send data from the host ECU either as a reply to a received message or based on an internal event.

Events type message are setup by VDR and sent whenever the given event was raised by other applications (i.e. Lighting generated event to inform user that it detected a critical error like OPEN LOAD or SHORT TO GROUND on a certain output channel).

VDR will also sent cyclic data messages if such messages were configured (TX messages configured as APP_EVENT_FORWARD_MSG)

Byte 0	***Byte 1***	***Byte 2***	***Byte 3***	***Byte 4***	***Byte 5***	***Byte 6***	**…**	***Byte n***
Message ID	*Message Type*	*Data 0*	*Data 1*	*Data 2*	*Data 3*	*Data 4*		*Data n-2*

❖ Table 3-2. TX Message format

Message Type – Represent the scope associated with that message. In case of TX messages the Type has the following values:

- TX_SEND_MSG – if the given message is of type: APP_EVENT_FORWARD_MSG
- RX_FORWARD_MSG – if the given message is a reply to a received RX_FORWARD_MSG
- TX_ON_REQUEST – if the given message is a reply to any other RX REQUEST messages.

- TX_ON_EVENT– if the given message is generated by an event. This message will send the status of the associated instance ID.

Note:

TX_ON_EVENT and TX_ON_REQUEST (when is a reply to REQ_STATUS message) will requires a considerable data payload. Basically, each message will require a minimum of: 5bytes + (17byte * N_channels) (where N_channels = number of output channels configured on the given instance). The user must make sure that the associated physical channel for that message, is capable of handling messages with the required data length.

The 'STATUS' messages are constructed using ASCII encoding and have the following format:

Byte 0	Message Id
Byte 1	Tx Type (On_Event or On_Reqeust)
Byte 2	Instance type: 'L' - lighting
Byte 3	Instance Id (0, 1, 2 ..)
Byte 4	' ' (empty space for formatting)
Byte 5 + repeating_offest*	'C'
Byte 6 + repeating_offest*	*Channel Id* (0, 1, 2…)
Byte 7 + repeating_offest*	' ' (empty space for formatting)
Byte 8 + repeating_offest*	'S'
Byte 9 + repeating_offest*	*Status Info* read from Lighting App. for that channel (Idle = 0, Active = 1, Stopped = 2, S2G = 4, OL = 8)
Byte 10 + repeating_offest*	' ' (empty space for formatting)
Byte 11 + repeating_offest*	'A'

Byte 12..13 + repeating_offest*	Active Command: Expected output command given on that channel (Percentage value – i.e. 50%)
Byte 14 + repeating_offest*	'%'
Byte 15 + repeating_offest*	' ' (empty space for formatting)
Byte 16 + repeating_offest*	' F'
Byte 17..18 + repeating_offest*	Feedback Voltage value: Current analog voltage value if feedback was configured on that channel.
Byte 19 + repeating_offest*	'm'
Byte 20 + repeating_offest*	'V'
Byte 21 + repeating_offest *	' ' (empty space for formatting)

* **repeating_offest** is calculated considering that the status is requested for all channels of a lighting instance with more than one channel. **repeating_offest = channel_id * 17** (channel_id = 1, 2, 3);

4.2 Description of the LEDs and Buttons functionality

The detailed description of the LEDs and Buttons functionality is depicted in the following table:

PortPin Name	Pin ID (PCR ID)	Pin Mode	Pin Direction	Pin Level	Connected HW	Channel Assignment
PortPin_Pwm1	48	EMIOS_0_eMIOS_0_CH_4_G_OUT	Out	Low	RGB_BLUE	-
PortPin_Pwm2	49	EMIOS_0_eMIOS_0_CH_5_G_OUT	Out	Low	RGB_RED	-
PortPin_DigOut1	12	GPIO	Out	Low	RGB_GREEN	Dio_out1
PortPin_DigKey1	43	GPIO	In	Low	SW3	Dio_DigKey1
PortPin_PwmKey2	62	GPIO	In	Low	SW2	Dio_DigKey2
PortPin_AnalogPot	144	ADC0_ADC0_P4	In	-	POT	-

Table 4-1. PORT and DIO Modules - Pin Configuration and DioChannel Assignment for S32K2XX

LEDs and Buttons	Functionality
D76(led)	Each time SW7 is pressed the duty cycle controlling this led increase with 12% until it reach 100% than starts again from 0%
D81(led)	When SW7 is pressed the led turns on, when SW3 is pressed again led turn off
D79 (led)	It’s duty cycle is control by the Analog Potentiometer (R409)
SW3 (button)	Input for controlling the duty cycle of led D76
SW7 (button)	Input for controlling led D81(On/Off)
R409 (button)	Input for controlling the duty cycle of led D79

Table 4-2. LEDs and Buttons Functionality for S32K2XX

Chapter 5
Building the Sample Application Example

This section describes the build procedure.

5.1 Building the Sample Application example

Procedure:

1. Open the Windows command prompt window
2. Change the current directory to the sample application folder
3. To build the sample, execute the following command to run launch.bat: launch.bat
4. The object files and linker output file (sample_app_mcal.elf) shall be generated inthe /bin subdirectory
5. To execute the sample application, load the executable file placed in the /binsubdirectory to the evaluation board using the Lauterbach debugger and run.cmm script.

Note

The launch.bat file calls the make.bat file and then the GNU make utility is called from the Tresos Studio bin directory.

5.2 Building with different compilers

To build the sample application with a different compiler, use the following parameter for the launch command:

launch.bat TOOLCHAIN=[toolchain]

where [toolchain] can have the values:

* ghs - default - use the GreenHills Multi compiler
* linaro - use the Gcc compiler
* iar - use the Iar compiler

5.3 Building for different run-modes

To build the sample application for a different run-mode, use the following parameter for the launch command:

launch.bat MODE=[run_mode] where

[run_mode] can have the values: * SUPR

- default - run in Supervisor mode

* USER - un in User mode

Note

In order to run in USER mode, all drivers that need to be executed in this mode should have the "Enable User Mode Support" parameter set to 'true' and their configuration files regenerated from Tresos.

Note

In order to run in USER mode, AUTOSAR OS should not be used since it does not allow other run-modes except Supervisor

5.4 Building for different controller derivatives

To build the sample application for a either 3M or 6M derivative, use the following parameter for the launch command:

launch.bat DERIV=[derivative] where [derivative] can have the values:
* 118 – build for S32K118 variant
* 144 – build for S32K144 variant

Note

In order to run for either of the variants the user should make sure that the correct resource was selected in Tresos and configurations were correctly generated for that derivative.

Note

It is possible to build with different option by using a command line with more than one parameter.

For instance: running “*launch.bat* TOOLCHAIN=ghs MODE=SUPR CORE=m7 “ will build the code for GHS compiler, Supervisor Mode running on core m7

5.5 Clean Object and Linker Output Files

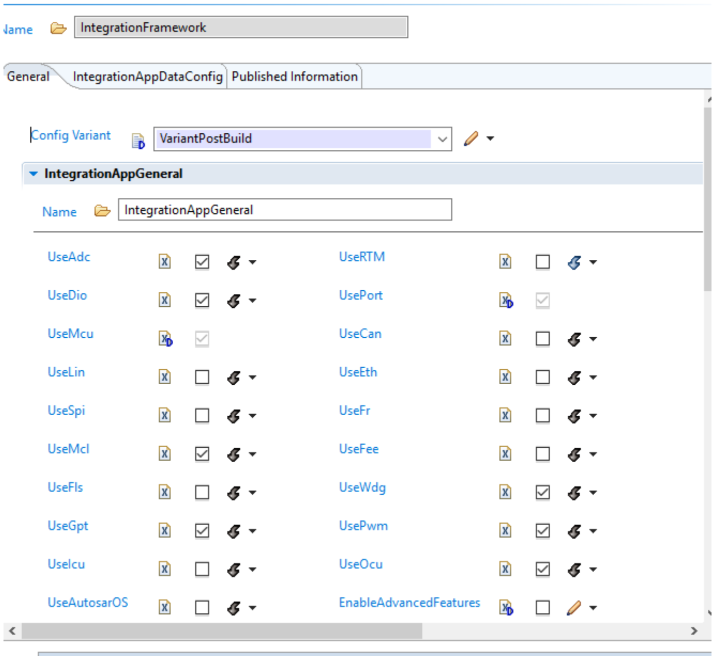
To clean the object and linker output files from the folder /bin, execute the following steps

Procedure:

1. Open the Windows command prompt window
2. Change the current directory to sample application folder
3. Execute the following command launch.bat clean
4. The object files and linker output files shall be cleared from the /bin and from the sample application root folders.

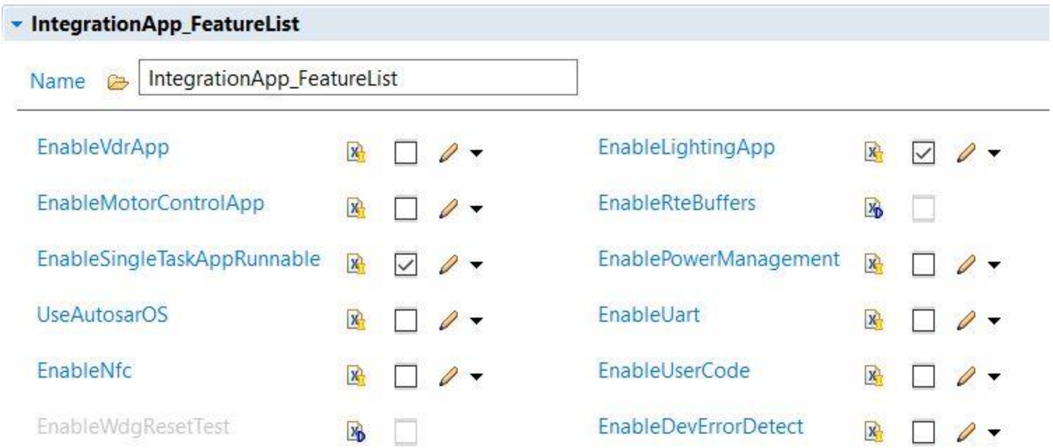
5.6 Framework Configuration Parameters

1. **IntegrationAppGeneral** Container – holds driver enablement parameters.



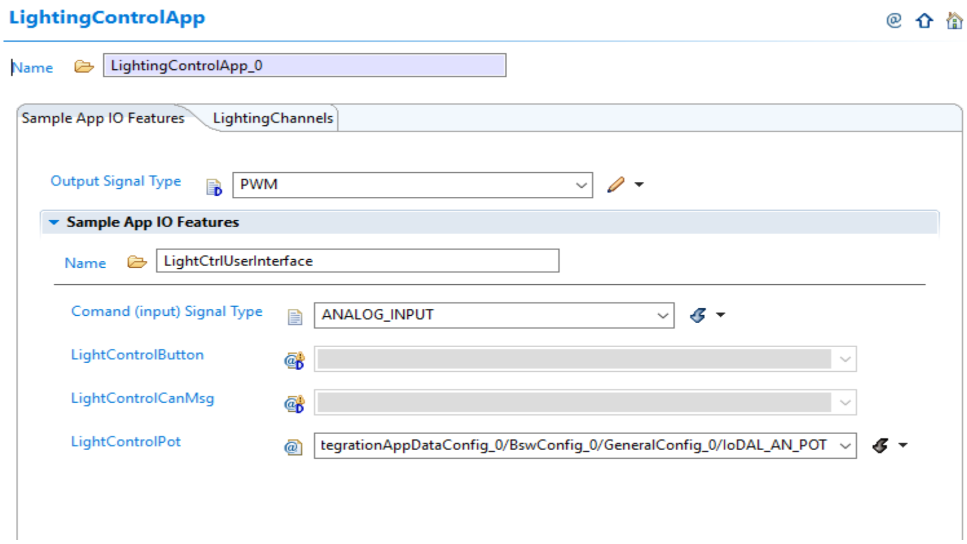
Note: For this version of application only Adc, Dio, Mcu, Gpt, Port, Pwm and Ocu drivers are supported.

2. **IntegrationAppFeaturesConfig** Container – holds application enablement parameters.



Note: For this version of application only Lighting App is supported.

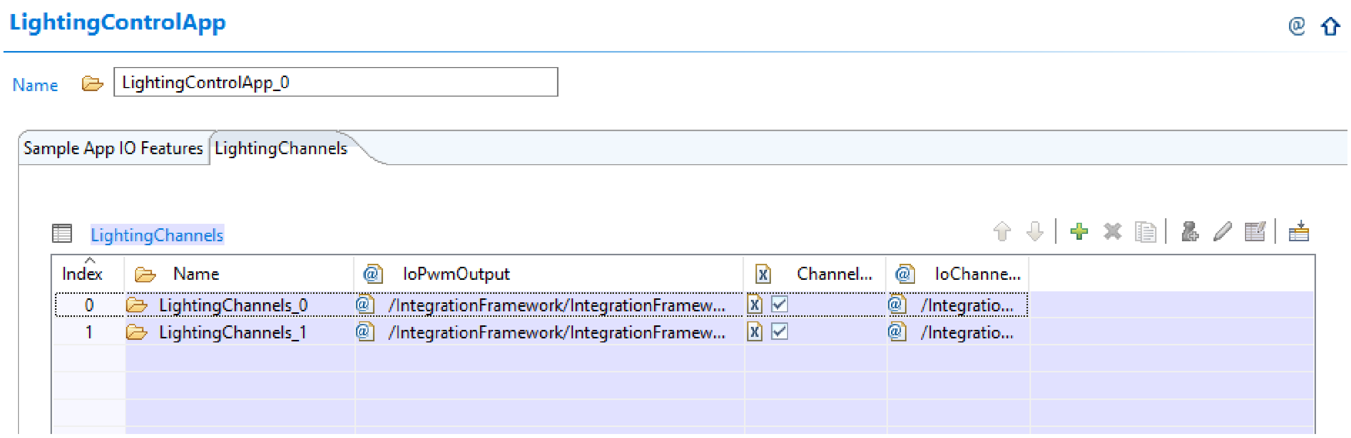
3. **AppConfig/LightingControlApp** Container – holds parameters for configuring each lighting instance.



- LightCtrlOutputSignalType: specifies what type of output current instance shall use (supported values: PWM, DIGITAL_OUTPUT)

- LightCtrlInputSignalType: specifies what type of input current instance shall use (supported values: ANALOG, DIGITAL_INPUT)

- **LightingChannels** Container – holds parameters for configuring each output channel for the current instance (and their corresponding feedback channels)



- **VDR Message configuration logic.**

Each VDR Message has a unique ID based on the given “direction” as following:

- To store data to current ECU, the VDR requires a RX_MESSAGE to be configured.
- To send data from current ECU, the VDR requires either a TX_ MESSAGE or a TX_REPLY to be configured.

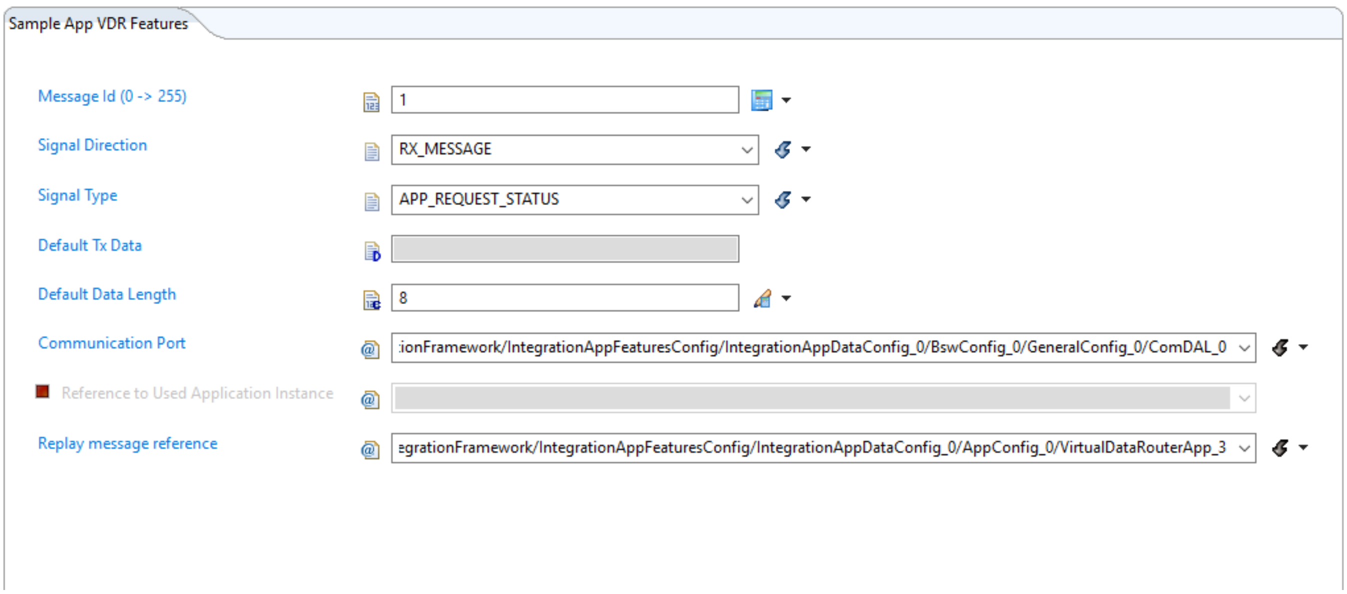
Each direction type requires a certain “signal type” to be configured:

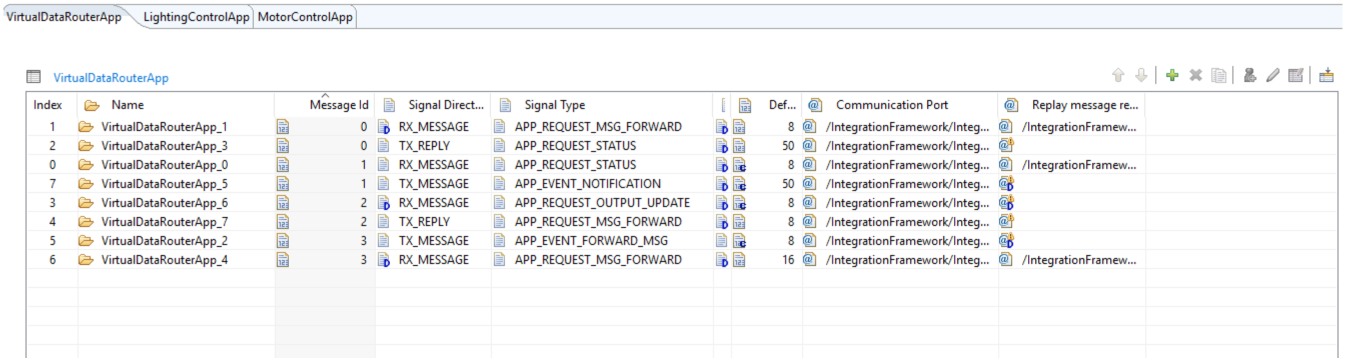
- All RX Messages require a signal of type: APP_REQUEST_[action], where [action] may be: STATUS, MSG_FORWARD, OUTPUT_ UPDATE etc.
  If [action] is different from ‘OUTPUT_ UPDATE’ the given RX Message will also require a TX_REPLY message to be configured.
- All TX Messages require a signal of type: APP_EVENT_[action], where [action] may be: NOTIFICATION, FORWARD_MSG.

Each message requires to have a COMDAL channel configured (see ComDal)

The default data length for each message must be equal or less then the underlaying ComDal channel data length. (i.e. if the message is expected to be routed on the CAN interface, than the data length for than message is given by the CAN Payload configuration attribute (‘MBDSR’)

Default Tx Data is only configurable if the message is of type: APP_EVENT_FORWARD_MSG



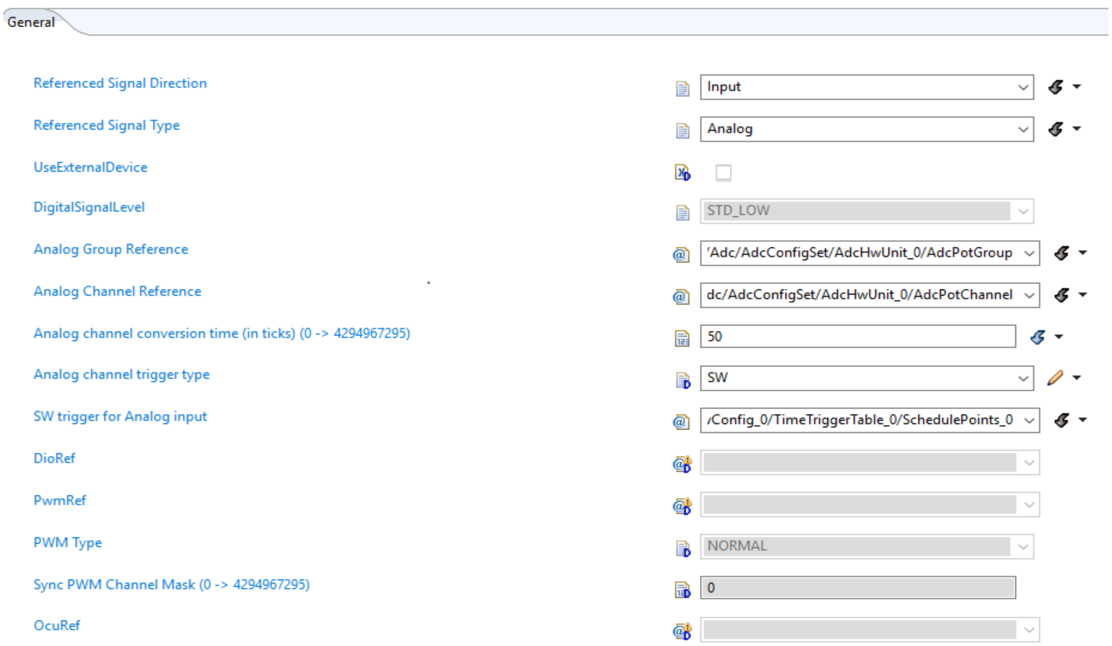


VirtualDataRouterApp | LightingControlApp | MotorControlApp

VirtualDataRouterApp

Index	Name	Message Id	Signal Direct...	Signal Type	Def...	Communication Port	Replay message re...
1	VirtualDataRouterApp_1	0	RX_MESSAGE	APP_REQUEST_MSG_FORWARD	8	/IntegrationFramework/Integ...	/IntegrationFramew...
2	VirtualDataRouterApp_3	0	TX_REPLY	APP_REQUEST_STATUS	50	/IntegrationFramework/Integ...	
0	VirtualDataRouterApp_0	1	RX_MESSAGE	APP_REQUEST_STATUS	8	/IntegrationFramework/Integ...	/IntegrationFramew...
7	VirtualDataRouterApp_5	1	TX_MESSAGE	APP_EVENT_NOTIFICATION	50	/IntegrationFramework/Integ...	
3	VirtualDataRouterApp_6	2	RX_MESSAGE	APP_REQUEST_OUTPUT_UPDATE	8	/IntegrationFramework/Integ...	
4	VirtualDataRouterApp_7	2	TX_REPLY	APP_REQUEST_MSG_FORWARD	8	/IntegrationFramework/Integ...	
5	VirtualDataRouterApp_2	3	TX_MESSAGE	APP_EVENT_FORWARD_MSG	8	/IntegrationFramework/Integ...	
6	VirtualDataRouterApp_4	3	RX_MESSAGE	APP_REQUEST_MSG_FORWARD	16	/IntegrationFramework/Integ...	/IntegrationFramew...

4. **BswConfig/GeneralConfig/IoDAL** Container holds IoDal channel descriptor configuration for each BSW IO signal.

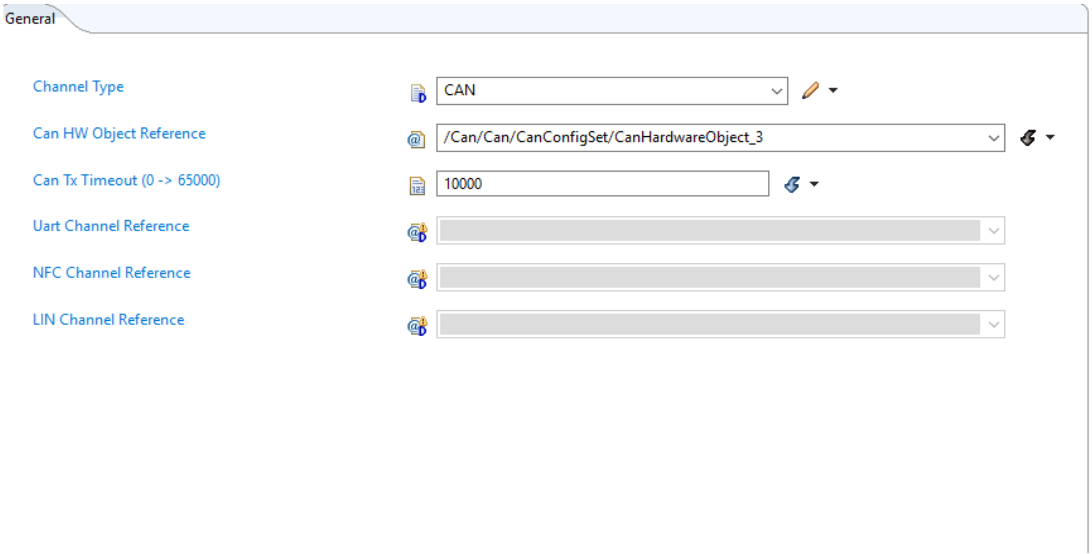


- ReferenceDirection: determines I/O channel direction (input/output)
- ReferenceType: determines signal type (analog, digital, pwm)

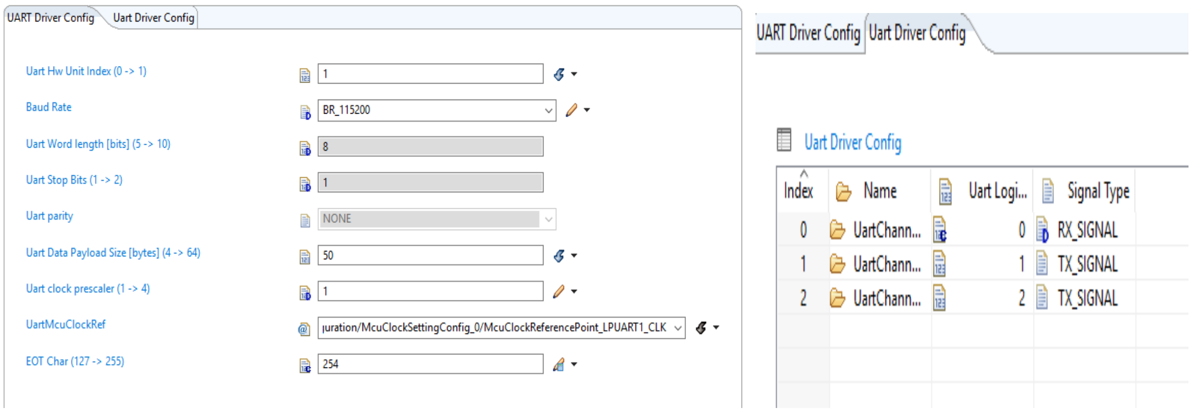
For analog signals, each descriptor is determined by unique combination of references to an Analog Channel and an Analog Group. Also for analog signals conversion time, trigger type and time are required.

For pwm and digital signals only references to underlaying drivers are needed.

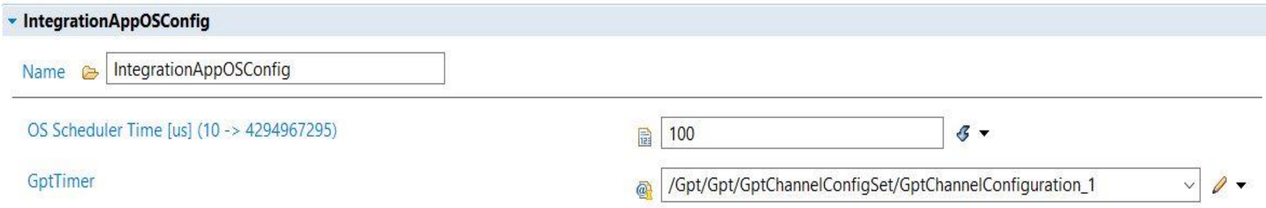
5. **BswConfig/GeneralConfig/ComDAL** container holds ComDal channel descriptor configuration for each BSW Communication signal.



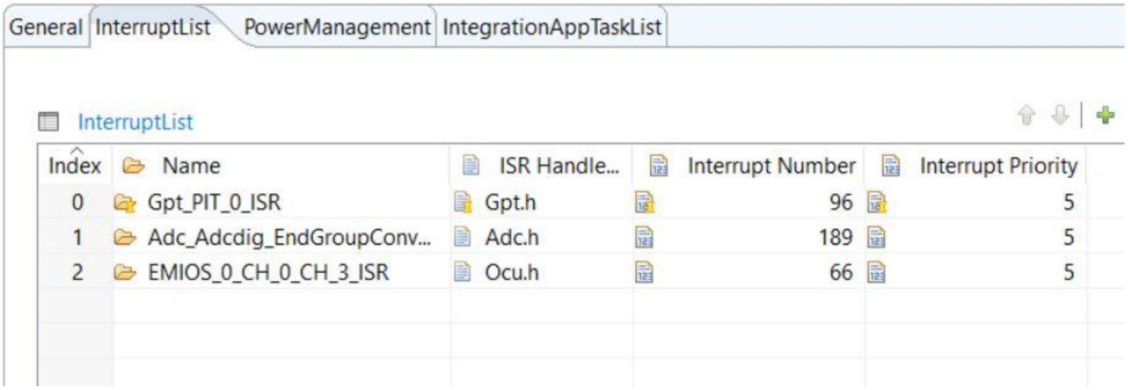
6. **BswConfig/UartDriverConfig** Container holds the configuration for the internal UART driver.



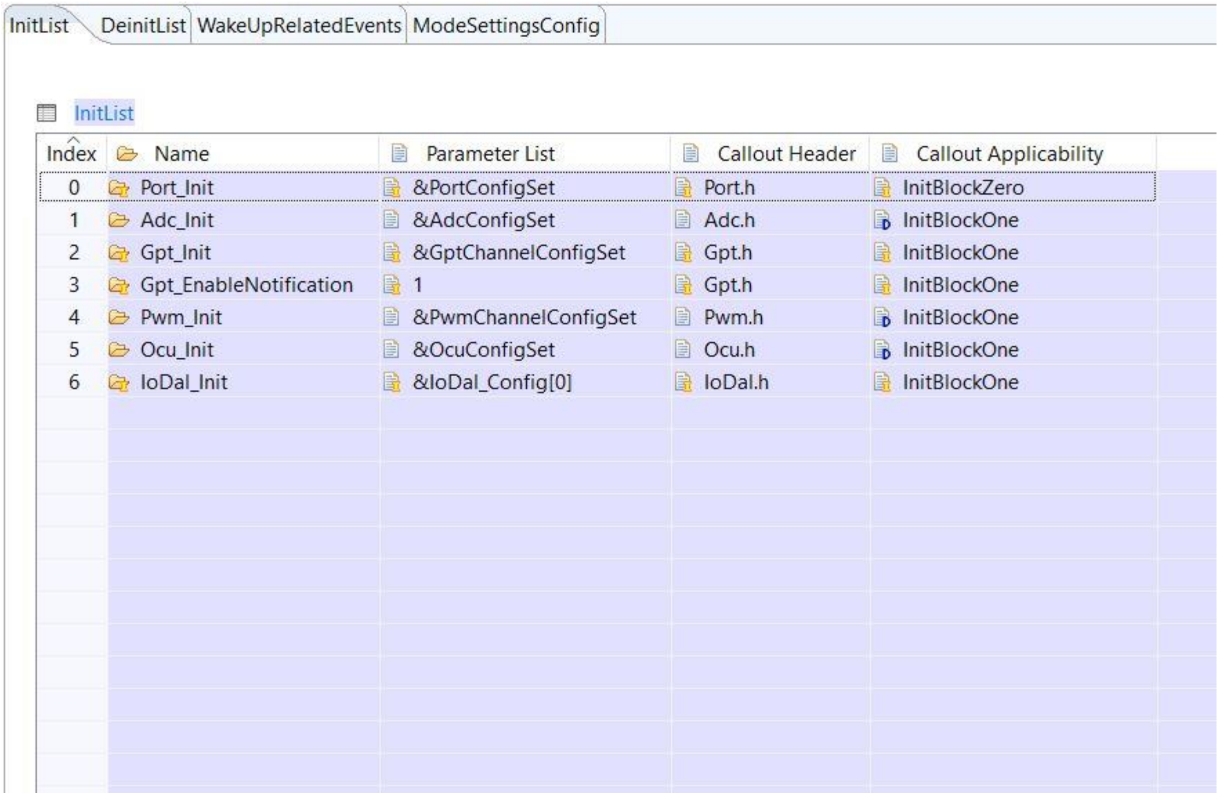
7. **BswConfig/GeneralConfig/SysDAL/General** Container holds SysDal global configuration, including the configuration on the internal scheduler.



8. **BswConfig/GeneralConfig/SysDAL/InterruptList** Container holds the interrupt handlers that need to be activated at system level. (in our case only GPT, OCU and ADC interrupts are enabled)



9. **BswConfig/GeneralConfig/SysDAL/PowerManagement/InitList** Container hold the list of API's that need to be called by SysDal during initialization.

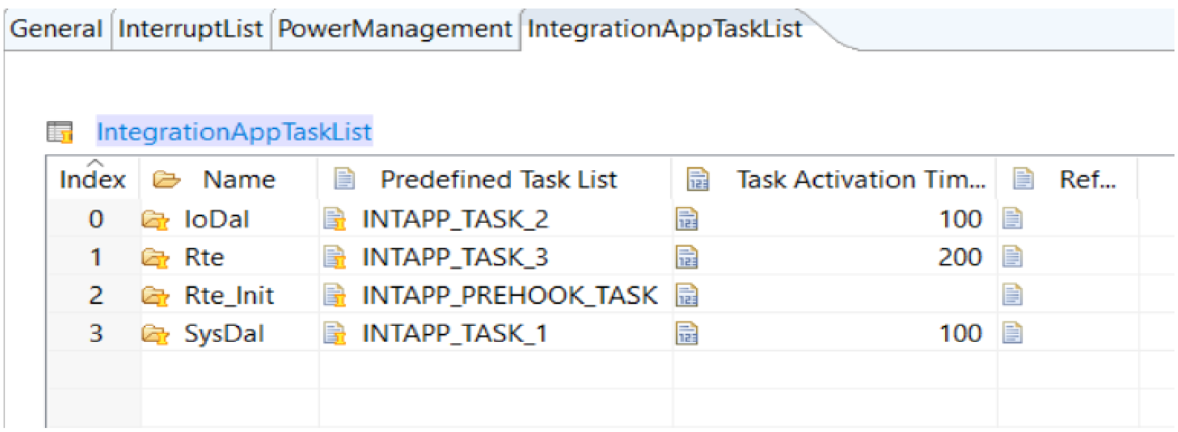


InitList | DeinitList | WakeUpRelatedEvents | ModeSettingsConfig

InitList

Index	Name	Parameter List	Callout Header	Callout Applicability
0	Port_Init	&PortConfigSet	Port.h	InitBlockZero
1	Adc_Init	&AdcConfigSet	Adc.h	InitBlockOne
2	Gpt_Init	&GptChannelConfigSet	Gpt.h	InitBlockOne
3	Gpt_EnableNotification	1	Gpt.h	InitBlockOne
4	Pwm_Init	&PwmChannelConfigSet	Pwm.h	InitBlockOne
5	Ocu_Init	&OcuConfigSet	Ocu.h	InitBlockOne
6	IoDal_Init	&IoDal_Config[0]	IoDal.h	InitBlockOne

10. **BswConfig/GeneralConfig/SysDAL/IntegrationAppTaskList** Container hold the configuration of the tasks for the internal round-robin scheduler. Up to 7 cyclic tasks and one sigle-shot pre-hook task can be enabled.



General | InterruptList | PowerManagement | IntegrationAppTaskList

IntegrationAppTaskList

Index	Name	Predefined Task List	Task Activation Tim...	Ref...
0	IoDal	INTAPP_TASK_2	100	
1	Rte	INTAPP_TASK_3	200	
2	Rte_Init	INTAPP_PREHOOK_TASK		
3	SysDal	INTAPP_TASK_1	100	

- For each defined task the user can configure a list of API's that will be called by it.

5.6 Modifying the Configuration in Tresos Studio

Users may change the application configuration according to their needs.

Procedure:

1. Open the EB Tresos Studio GUI
2. Open previously imported Sample Application project
3. Use the Tresos Studio GUI to modify configuration parameter values and save thechanges. The value of the External Crystal Frequency parameter can be changed as depicted in Figure 5-1: Modifying the External Crystal Frequency
4. Select the Sample Application project and click on "Generate" button to generate theconfiguration files.
5. Copy the generated configuration files from workspace/[project]/output/includedirectory into the Sample Application folder /cfg/include.

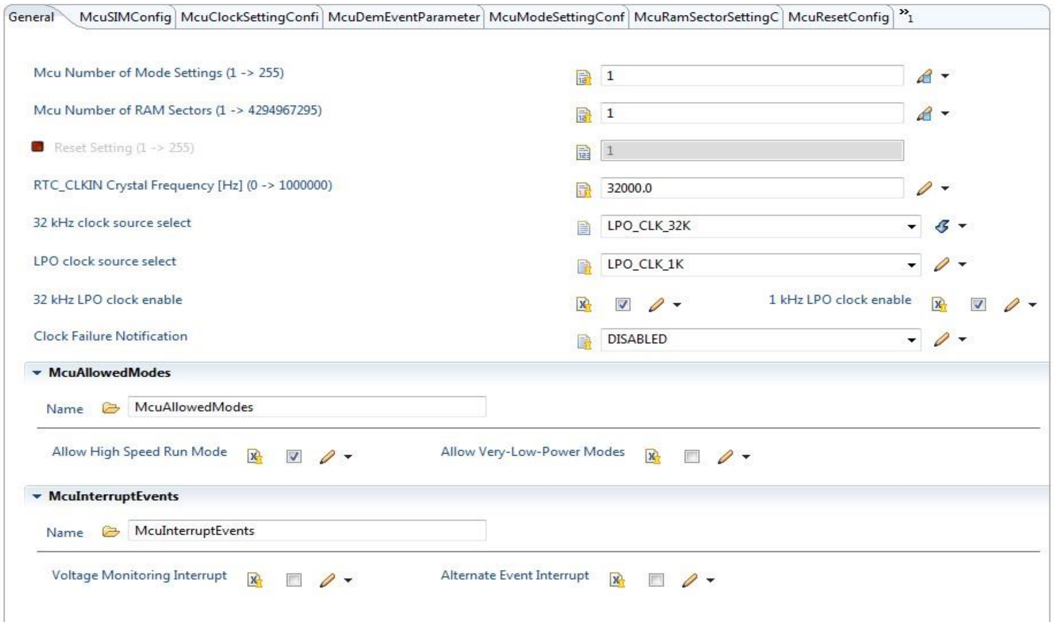


Figure 5-1. Modifying the External Crystal Frequency

5.7 Examples of Communication Data

The following messages are configured:

Message Id	*Direction*	*Type*	*Source*	*Reply Id*	*Payload*
0	RX_MESSAGE	APP_REQUEST_MSG_FORWARD	CAN0 – HO2	2	8
1	RX_MESSAGE	APP_REQUEST_STATUS	CAN0 – HO3	0	8
2	RX_MESSAGE	APP_REQUEST_OUTPUT_UPDATE	CAN1 – HO4	-	8
3	RX_MESSAGE	APP_REQUEST_MSG_FORWARD	UART Ch0	2	16
0	TX_REPLY	APP_REQUEST_STATUS	UART Ch1	-	50
1	TX_MESSAGE	APP_EVENT_NOTIFICATION	UART Ch2	-	50
2	TX_REPLY	APP_REQUEST_MSG_FORWARD	CAN1 – HO5	-	8
3	TX_MESSAGE	APP_EVENT_FORWARD_MSG	CAN0 – HO0	-	8

Figure 5-2. Example of a list of configured messages.

Having the message list configured as above, the following list of messages can be handled by the host ECU.

Source	*Byte*								*Comment*
	0	***1***	***2***	***3***	***4***	***5***	***6***	***7***	
CAN0	*0*	*16*	*2*	*15*	*10*	*11*	*10*	*15*	*Request Forward message with payload 0xF, 0xA, 0xB,0xA, 0xF on Tx Reply Id:2*
CAN0	*1*	*1*	*0*	*0*	*0*	*0*	*0*	*0*	*Request Status Info on Lighting Instance 0. Message will be packed in the Tx Reply with the id 0.*
CAN1	*2*	*2*	*0*	*2*	*1*	*100*	*0*	*0*	*Request Output Update on Lighting instance 2, Unicast - Channel 1, New Value: 100%*
UART	*3*	*16*	*2*	*50*	*16*	*15*	*16*		*Request Forward message with payload 0x32, 0x10, 0xF,0x10, on Tx reply Id:2. Message will be packed in the Tx Reply with the id 2.*

Figure 5-2. Example Rx Messages to be handled by host ECU

How to Reach Us:

Home Page:
nxp.com

Web Support:
nxp.com/support

Document Number UMSAASR4.3R1.0.0
Revision 1.2