MICROSAR Classic I-PDU Multiplexer

Technical Reference

Version 6.01.00

Authors	visbms, visms, visfrm
Status	Released

Document Information

History

Author	Date	Version	Remarks
visssa	2011-12-06	1.00.00	Initial CFG5 version derived from TechnicalReference_ASR_IpduM.pdf
visbms	2012-08-23	2.00.00	ESCAN00058313 AR4-160: Support AUTOSAR 4.0.3
visbms	2013-01-29	2.01.00	ESCAN00063294 AR4-197: Support BIG_ENDIAN Copy Segments in IpduM
visms	2013-04-04	2.02.00	ESCAN00064368 AR4-325: Post-Build Loadable
visbms	2014-11-05	2.03.00	AR4-698: Post-Build Selectable (Identity Manager)
visbms	2014-12-01	2.04.00	FEAT-229: Support 16bit selector in IpduM [AR4-927]
visbms	2015-08-11	2.05.00	FEAT-1315: IPDUM for CAN-FD supporting nPdu2Frame-Mapping
visms	2016-02-25	2.06.00	FEAT-1631: Trigger Transmit API with SduLength In/Out according to ASR4.2.2
visbms	2017-12-21	2.07.00	STORYC-3561: made CalculateSizeOfContainer an external API
visbms	2018-03-06	2.08.00	FEAT-2072: Introduce separate main functions for reception and transmission for IPDUM
visbms	2018-05-04	3.00.00	STORYC-5229: Add Container PDU Configuration to Doc_TechRef
visbms	2019-01-18	3.01.00	STORYC-6469: Improve documentation and validation for Container PDUs and FlexRay
visbms	2019-11-19	4.00.00	COM-941: Finalize priority for Tx ContainedIPdus with LastIsBest collection semantics
visbms	2021-01-18	4.00.01	Expanded the chapters “Configuration” and “Limitations”
visbms	2021-05-19	5.00.00	Removed DirectComInvocation
visfrm	2021-06-30	5.01.00	COM-1961: Update Doc_TechRef with the new template COM-1836: [MC] ASR 4.5 Multi-Partition Support in IpduM
visbms	2023-02-08	6.00.00	Product name updated to Classic
visms	2024-03-20	6.01.00	COM-5300: Compile Code as Unity Build

Reference Documents

No.	Source	Title	Version
[1]	AUTOSAR	Specification of I-PDU Multiplexer	R4.3.0
[2]	AUTOSAR	List of Basic Software Modules	R4.3.0
[3]	Vector	Technical Reference MICROSAR Post-Build Loadable	See delivery

Caution
We have configured the programs in accordance with your specifications in the questionnaire. Whereas the programs do support other configurations than the one specified in your questionnaire, Vector´s release of the programs delivered to your company is expressly restricted to the configuration you have specified in the questionnaire.

Contents

Illustrations

Tables

1 Introduction

This document describes the functionality, API and configuration of the AUTOSAR BSW module IPDUM as specified in [1].

Supported Configuration Variants:	PRE-COMPILE [SELECTABLE] POST-BUILD-LOADABLE [SELECTABLE]	
Vendor ID:	IPDUM_VENDOR_ID	30 decimal (= Vector-Informatik, according to HIS)
Module ID:	IPDUM_MODULE_ID	52 decimal (according to ref. [2])

Multiplexing is a concept generally used to save CAN identifiers where an I-PDU with a CAN ID is used to carry different I-PDUs thereby saving the CAN IDs for the I-PDUs it carries. The IpduM can be used also with Flexray and LIN.

I-PDU multiplexing means using the same PCI of a PDU with more than one unique layouts of its SDU. The SDU in a multiplexed I-PDU contains a selector field which is used to determine the layout of the SDU being multiplexed or de-multiplexed.
The AUTOSAR IpduM multiplexes a dynamic part along with or without a static part.
Each dynamic and static part is an independent signal-I-PDU in AUTOSAR BSW module Com.

1.1 Architecture Overview

The following figure shows where the IPDUM is located in the AUTOSAR architecture.

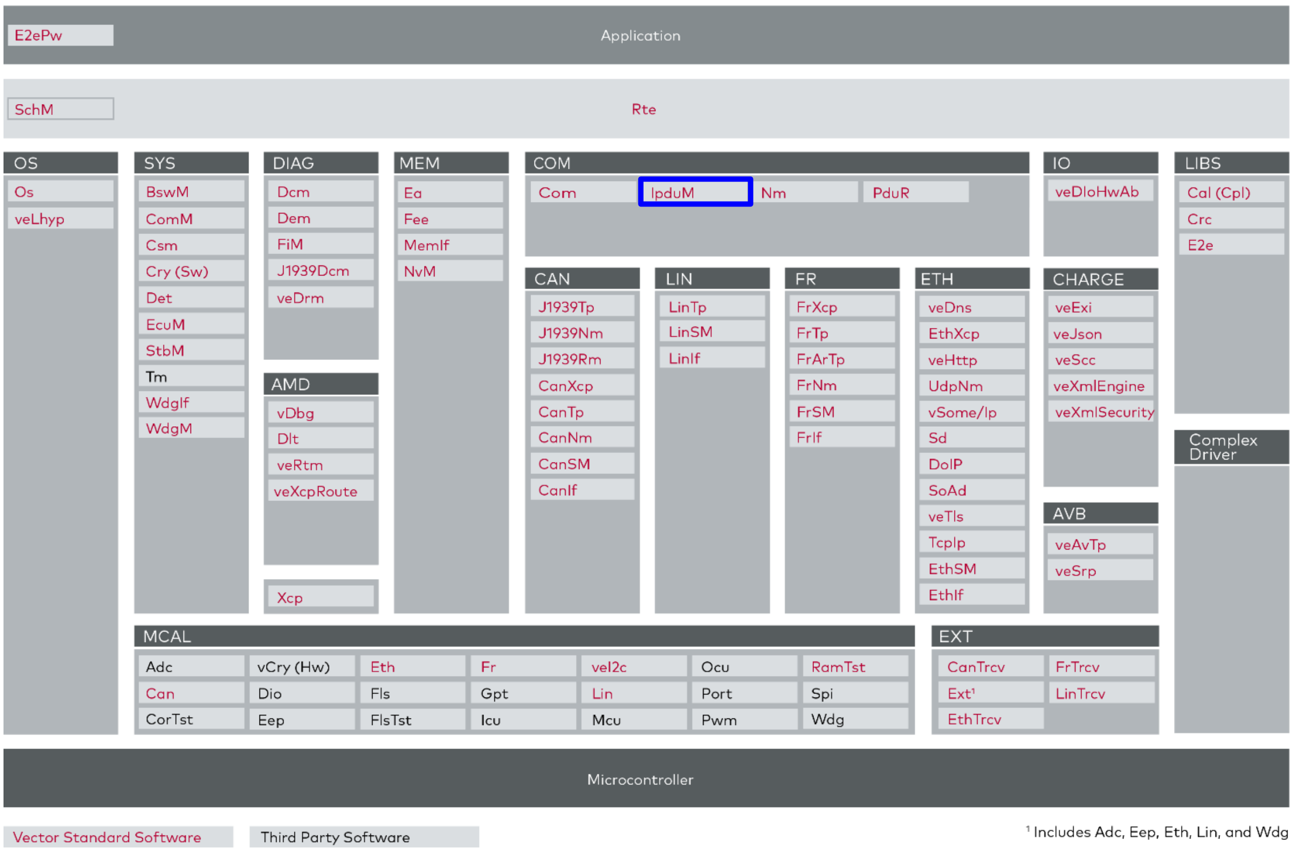

Figure 1-1 AUTOSAR 4.1 Architecture Overview

Figure 1-2 AUTOSAR architecture

The next figure shows the interfaces to adjacent modules of the IPDUM. These interfaces are described in chapter 3.2.

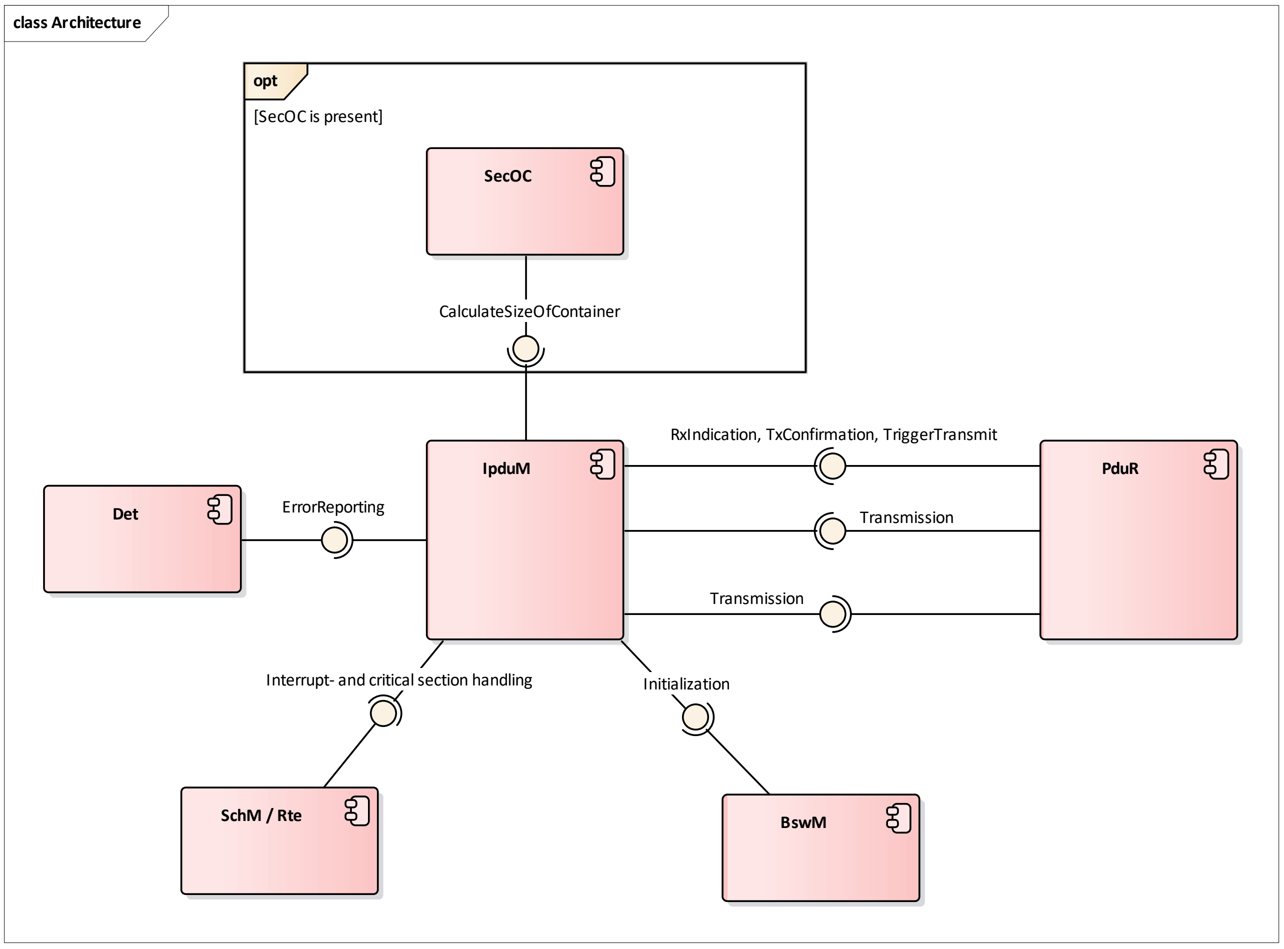

Figure 1-3 Interfaces to adjacent modules of the IPDUM

The IpduM only interacts with the PduR and offers an API used by SecOC.

The IpduM also uses the interfaces to the BSW Scheduler and Det.

2 Functional Description

2.1 Features

The features listed in the following tables cover the complete functionality specified for the IPDUM.

The AUTOSAR standard functionality is specified in [1], the corresponding features are listed in the tables

- > Table 2-1 Supported AUTOSAR standard conform features
- > Table 2-2 Not supported AUTOSAR standard conform features
- > Table 2-3 Features provided beyond the AUTOSAR standard

For further information of not supported features see also chapter 7.

The following features specified in [1] are supported:

Supported Feature
Configuration
> Multiplexing of the static and dynamic parts of multiplexed I-PDUs.
> Event based transmission triggering of the multiplexed I-PDU(s).
> Forwarding confirmations of all transmitted multiplexed I-PDUs to the BSW module AUTOSAR Com.
> Timeout handling of transmission confirmations.
> De-multiplexing and indicating receptions of the de-multiplexed dynamic parts and static part to the BSW module AUTOSAR Com.
> Static part configuration support in multiplexed I-PDU(s).
> AUTOSAR EcuC format.
> Initialization of transmission multiplex I-PDU buffers. This is required by e.g. FrIf in case the transmission of a multiplex I-PDU is triggered by interface layer BSW module before the transmission of their mapped dynamic and or static parts is requested.
> Just-in-Time Update of dynamic and / or static parts.
> DET error reporting.

Table 2-1 Supported AUTOSAR standard conform features

The following features specified in [1] are not supported:

Not Supported AUTOSAR Standard Conform Features
> Configuring of transmission confirmations of transmitted multiplexed I-PDUs AUTOSAR Com.

Table 2-2 Not supported AUTOSAR standard conform features

The following features are provided beyond the AUTOSAR standard:

Features Provided Beyond The AUTOSAR Standard
> nPdu-to-Frame mapping for CAN-FD
> Priority handling for last-is-best Container PDUs

Table 2-3 Features provided beyond the AUTOSAR standard

2.2 Initialization

The IpduM is initialized by calling the API `IpduM_Init()`. This is done by the AUTOSAR BSW module EcuM. If this is not the case, an adequate and suitable solution must be provided.

The API initializes all the transmission multiplexed I-PDU(s) buffer(s) and all runtime relevant module variables. The successful execution of the API initializes the IpduM.

Caution
In case of a multi-partition configuration, the API IpduM_Init() must only be called once on a designated master-partition. Communication on any partition must only start after the initialization was done.

2.3 States

The IpduM has the states: uninitialized and initialized. By calling IpduM_Init(), the state is changed from uninitialized to initialized.

2.4 Main Functions

If the IpduM is used without Multi-Partition support:

> IpduM provides `IpduM_MainFunctionTx()` and `IpduM_MainFunctionRx()`. Both should be called cyclically by the Basic Software Scheduler or a similar component.

If the IpduM is used with Multi-Partition support:

> IpduM provides postfixed MainFunction APIs which must be called cyclically on the actual partition:

  > `IpduM_MainFunctionTx_<EcucPartitionShortname>()`

  > `IpduM_MainFunctionRx_<EcucPartitionShortname>()`

The `IpduM_MainFunctionTx()` takes care of timeout handling of transmission confirmations for multiplexed and container PDUs, send timeout handling of container PDUs, transmission of queued container PDUs.

The `IpduM_MainFunctionRx()` takes care of reception of queued container PDUs.

2.5 Error Handling

2.5.1 Development Error Reporting

By default, development errors are reported to the DET using the service `Det_ReportError()` or `Det_ReportRuntimeError()` as specified in [1], if development error reporting is enabled (i.e. pre-compile parameter `IPDUM_DEV_ERROR_DETECT==STD_ON`). The DET reporting can be enabled during pre-compile time.

If another module is used for development error reporting, the function prototype for reporting the error can be configured by the integrator, but must have the same signature as the service `Det_ReportError()` or `Det_ReportRuntimeError()`.

The reported IPDUM ID is 52.

The reported services IDs identify the IPDUM services as specified by [1]. For the service IDs and their related service names see IpduM.h Doxygen DefGroup “IpduMApiIds”.

The errors reported to DET are described in section 7.6 of [1].

2.6 nPdu-to-Frame mapping for CAN-FD

This feature was backported from AUTOSAR 4.2.1. It can be used to collect multiple PDUs and transfer them in one CAN-FD frame for saving bandwidth or tunneling several classic CAN busses over one CAN-FD bus. The IpduM module handles packing and unpacking of those frames similar to the multiplexing feature.

Several “contained” PDUs can be put into a single “container” PDU in arbitrary order and unpacked in the same order on the receiver side. Transmission is triggered by one of these conditions:

> Contained PDU send timeout runs out.

> Container PDU send timeout runs out.

> Size threshold of container PDU is reached.

> Contained PDU to be added does not fit into the container PDU anymore.

> Contained PDU is configured for immediate transmission.

> Container PDU is configured for immediate transmission.

On the receiver side, the contained PDUs are retrieved from the container PDU again and RxIndications are created for every single contained PDU in the same order they were put into the container PDU. For even greater flexibility, a container PDU on the receiver side can be configured to accept any contained PDU instead of just the ones that are configured.

3 Integration

This chapter provides information necessary for the integration of the MICROSAR Classic IPDUM into an application environment of an ECU.

3.1 Scope of Delivery

The delivery of the IPDUM contains the files which are described in the chapters 3.1.1 and 3.1.2:

3.1.1 Static Files

File Name	Description
IpduM_Unity.c	This is the source file of the IPDUM to be used by the compiler. Unit specific c files are included here.
IpduM.c	This is the source file of the IPDUM. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM.h	This is the header file of the IPDUM.
IpduM_TxConfirmation.c	This is the source file of the IPDUM Tx Confirmation Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_TxConfirmation.h	This is the header file of the IPDUM Tx Confirmation Unit.
IpduM_ContainerRx.c	This is the source file of the IPDUM Container Rx Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ContainerRx.h	This is the header file of the IPDUM Container Rx Unit.
IpduM_ ContainerRxProcessing.c	This is the source file of the IPDUM Container Rx Processing Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerRxProcessing.h	This is the header file of the IPDUM Container Rx Processing Unit.
IpduM_ ContainerRxQueue.c	This is the source file of the IPDUM Container Rx Queue Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerRxQueue.h	This is the header file of the IPDUM Container Rx Queue Unit.
IpduM_ ContainerTx.c	This is the source file of the IPDUM Container Tx Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTx.h	This is the header file of the IPDUM Container Tx Unit.
IpduM_ ContainerTxDataQueue.c	This is the source file of the IPDUM Container Tx DataQueue Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxDataQueue.h	This is the header file of the IPDUM Container Tx DataQueue Unit.
IpduM_	This is the source file of the IPDUM Container Tx

File Name	Description
ContainerTxDataQueueHandling.c	DataQueue Handling Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxDataQueueHandling.h	This is the header file of the IPDUM Container Tx DataQueue Handling Unit.
IpduM_ ContainerTxDataQueueInstance.c	This is the source file of the IPDUM Container Tx DataQueue Instance Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxDataQueueInstance.h	This is the header file of the IPDUM Container Tx DataQueue Instance Unit.
IpduM_ ContainerTxDataQueueUtil.c	This is the source file of the IPDUM Container Tx DataQueue Util Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxDataQueueUtil.h	This is the header file of the IPDUM Container Tx DataQueue Util Unit.
IpduM_ ContainerTxRequestQueue.c	This is the source file of the IPDUM Container Tx Request Queue Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxRequestQueue.h	This is the header file of the IPDUM Container Tx Request Queue Unit.
IpduM_ ContainerTxRequestQueueHandlin g.c	This is the source file of the IPDUM Container Tx Request Queue Handling Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxRequestQueueHandlin g.h	This is the header file of the IPDUM Container Tx Request Queue Handling Unit.
IpduM_ ContainerTxRequestQueuePriorityH andling.c	This is the source file of the IPDUM Container Tx Request Queue Priority Handling Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ ContainerTxRequestQueuePriorityH andling.h	This is the header file of the IPDUM Container Tx Request Queue Priority Handling Unit.
IpduM_ContainerTxRequestQueue Util.c	This is the source file of the IPDUM Container Tx Request Queue Util Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ContainerTxRequestQueue Util.h	This is the header file of the IPDUM Container Tx Request Queue Util Unit.
IpduM_ContainerTxSendTimeout.c	This is the source file of the IPDUM Container Tx Send Timeout Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ContainerTxSendTimeout.h	This is the header file of the IPDUM Container Tx Send Timeout Unit.
IpduM_ContainerTxTrigger.c	This is the source file of the IPDUM Container Tx Trigger Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ContainerTxTrigger.h	This is the header file of the IPDUM Container Tx Trigger Unit.

File Name	Description
IpduM_ContainerUtil.c	This is the source file of the IPDUM Container Util Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_ContainerUtil.h	This is the header file of the IPDUM Container Util Unit.
IpduM_MuxRx.c	This is the source file of the IPDUM Mux Rx Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_MuxRx.h	This is the header file of the IPDUM Mux Rx Unit.
IpduM_MuxTx.c	This is the source file of the IPDUM Mux Tx Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_MuxTx.h	This is the header file of the IPDUM Mux Tx Unit.
IpduM_MuxTxJit.c	This is the source file of the IPDUM Mux Tx Jit Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_MuxTxJit.h	This is the header file of the IPDUM Mux Tx Jit Unit.
IpduM_MuxUtil.c	This is the source file of the IPDUM Mux Util Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_MuxUtil.h	This is the header file of the IPDUM Mux Util Unit.
IpduM_Reporting.c	This is the source file of the IPDUM Reporting Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_Reporting.h	This is the header file of the IPDUM Reporting Util Unit.
IpduM_TxConfBuffer.c	This is the source file of the IPDUM Tx Conf Buffer Unit. Exclude this file from your build system. The file is build via IpduM_Unity.c.
IpduM_TxConfBuffer.h	This is the header file of the IPDUM Tx Conf Buffer Unit.
IpduM_Cbk.h	This is the callback header file of IPDUM.

Table 3-1 Static files

3.1.2 Dynamic Files

The dynamic files are generated by the configuration tool DaVinci Configurator.

File Name	Description
IpduM_Cfg.h	This file contains: > global constant macros > global function macros > global data types and structures > global data prototypes > global function prototypes of CONFIG-CLASS PRE-COMPILE data.

File Name	Description
IpduM_Lcfg.h	This file contains: > global constant macros > global function macros > global data types and structures > global data prototypes > global function prototypes of CONFIG-CLASS LINK data.
IpduM_Lcfg.c	This file contains: > local constant macros > local function macros > local data types and structures > local data prototypes > local data > global data of CONFIG-CLASS LINK and PRE-COMPILE data.
IpduM_PBcfg.h	This file contains: > global constant macros > global function macros > global data types and structures > global data prototypes > global function prototypes of CONFIG-CLASS POST-BUILD data.
IpduM_PBcfg.c	This file contains: > local constant macros > local function macros > local data types and structures > local data prototypes > local data > global data of CONFIG-CLASS POST-BUILD data.

Table 3-2 Generated files

3.2 Critical Sections

The critical section IPDUM_EXCLUSIVE_AREA_0 has to lock global interrupts to protect common critical sections.

4 Understanding IpduM

This chapter explains the modes the IpduM can operate in and the paths a PDU can take through the IpduM.

4.1 Multiplexing

Multiplexing in IpduM means choosing between different, fixed data layouts for a single PDU for every transmission. With multiplexing, there is one selector value that defines which of the predefined data layouts is used by the PDU. There is also an optional static part that is the same for all data layouts.

4.1.1 Configuration

Multiplexed PDUs are configured in the IpduMRxPathway and IpduMTxPathway containers. Each of the fixed data layouts is configured as a separate Com PDU, and the optional static part is an additional, separate Com PDU.

Caution
IpduMTxPathway parts (both static and dynamic parts) may only link via PduR routing paths to MICROSAR Classic Com Tx PDUs. Other communication partners are not supported here.

4.1.1.1 Tx Pathway

Each TxPathway consists of at least one dynamic part and zero or one static part. Within all dynamic parts of the same TxPathway, there must be one signal in the same position that contains the selector value. Its position is configured in the IpduMSelectorFieldPosition. The other signals of the dynamic parts don't have to overlap. The value of this selector value is configured in Com as the default value of that PDU. The static part signals must not overlap with any of the dynamic part signals.

4.1.1.1.1 Triggering a multiplexed PDU

There are four trigger modes:

- Static part trigger
- Dynamic part trigger
- Static or dynamic part trigger
- None

When the according PDU is received from Com, the multiplexed PDU is transmitted to the bus. The "none" mode is only useful for a time-triggered bus like FlexRay where immediate transmission is not needed.

4.1.1.1.2 Just-in-Time Update (JIT Update)

If the Com module uses update bits, special configuration in IpduM is necessary. The Com update bits can either be cleared on transmission or on triggerTransmit, not both. So in order for this to work, the IpduM must always get the PDU in the same mode: either always on transmit, or always on triggerTransmit.

For a time-triggered bus like FlexRay, the “none” trigger is typically used, and the data received from Com does not reach the bus but is buffered in IpduM instead. If the update bit clear context is configured to reset on transmission, these repeated buffered transmissions can lead to overwriting the update bit value. So changes to the update bit won’t be reflected on the bus. To fix this, configure the update bit to be reset on triggerTransmit and configure all static and dynamic parts of the according TxPathway to use the JIT update. This gets data via triggerTransmit from Com, essentially forwarding the triggerTransmit call from the bus and ensuring that this PDU with the update bits reaches the bus.

For other buses like Can, one of the three other triggers is usually used and the data from Com reaches the bus when it is triggered. If used with a static part, however, maintaining the same mode for transmission from Com for clearing the update bits requires special configuration. The easiest way to go here is choosing the dynamic part trigger, setting the dynamic part Com PDUs to clear the update bit on transmit, the static part Com PDU to clear on triggerTransmit, and the IpduM static part to use the JIT update. This way, both parts are always up to date on the bus and both get their update bits cleared correctly.

If a part – static or dynamic – does not have update bits, it usually does not need JIT update at all. If getting the latest data is critical though, the JIT update can be used for that as well.

4.1.1.1.3 Unused Areas Default Value

The unused areas default value in IpduM should match the value configured in the COM module, because during initialization, the unused areas default value from IpduM is used with the initial values from COM.

4.1.1.2 Rx Pathway

Each RxPathway consists of at least one dynamic part and zero or one static part. Within all dynamic parts of the same RxPathway, there must be one signal in the same position that contains the selector value. Its position is configured in the IpduMSelectorFieldPosition. The other signals of the dynamic parts don’t have to overlap. The value of this selector value is configured in the RxSelectorValue in each dynamic part. The static part signals must not overlap with any of the dynamic part signals.

4.1.1.3 Meta Data

Meta Data is transparently forwarded through IpduM multiplexed PDUs. It is appended to the end of multiplexed and dynamic part PDUs. The length of the dynamic part PDUs must be identical to the length of the multiplexed PDUs for this to work.

Caution
Meta Data is not supported for static part PDUs.

Caution
Meta Data is only supported in combination with the MICROSAR Classic CanIf.

4.2 Container Handling

Container handling means packing multiple PDUs into a single, larger “container PDU” on transmission, and unpacking it into the individual PDUs again on reception. This can be a dynamic process, comparable to a FIFO, or it can be a static process with fixed positions within a given container PDU. Unlike multiplexing, there are no multiple static layouts, however.

4.2.1 Configuration

Container handling is configured in the IpduMContainedRxPdu, IpduMContainedTxPdu, IpduMContainerRxPdu and IpduMContainerTxPdu containers.

There are two different modes of operation: dynamic (using short or long headers) or static (using no headers, but optionally using update bits). These can be configured per container PDU. Usually, it is one setting for the entire project, however. Most of the configuration applies to both modes.

4.2.1.1 Tx Container / Tx Contained

The container defines the large PDU that is used to collect several contained PDUs and transport them as one package on the bus.

4.2.1.1.1 Data provision: Direct vs. Trigger Transmit

The parameter IpduMContainerTxTriggerMode is used to determine when to remove a Container PDU from the IpduM’s internal buffer.

> IPDUM_DIRECT: the PDU is removed on triggering

> IPDUM_TRIGGERTRANSMIT: the PDU is removed when it is fetched via a call to IpduM_TriggerTransmit

So, the value should be IPDUM_DIRECT for every bus that does not fetch data via Trigger Transmit.

When a Container PDU is triggered, it is transmitted to the lower layer module. The detailed behavior depends on the data provision set in the configuration:

> IPDUM_DIRECT:

  > The Transmit call contains the data of the container Pdu.

  > The container Pdu will be closed and no new contained Pdus can be added.

  - The container Pdu is removed from the queue after a successful transmit. It will be re-transmitted in the next MainFunction if the transmission failed.
- IPDUM_TRIGGERTRANSMIT:
  - The Transmit call contains no data (valid pointer, but Pdu length of 0). It is only an informative transmit call.
  - The container Pdu will stay 'open' and new contained Pdus can be added until the lower layer module fetches the data via the TriggerTransmit API.

The container Pdu will be closed and queued immediately if the informative transmit was unsuccessful. It will be re-transmitted in the next MainFunction.

4.2.1.1.2 Collection Semantics / Queueing

There are two ways the IpduM can collect PDUs into a container: either as a transparent FIFO (maintaining the order they came in, including all duplicate PDUs) or with a "last is best" approach (only keeping one instance of each PDU – the most recent one – and maintaining the order they were first added into the container). This is configured with the Collection Semantics switch in the contained PDU.

Depending on this configuration, the queue behaves differently: for "last is best", there is exactly one buffer for each PDU as duplicates cannot exist. Therefore, the queue size configured in the container is ignored. For "queued", the queue size defines the number of containers that can be held in the queue. Contrary to "last is best", there is no inherent limit here. Finding the right queue size requires analyzing the "worst case" burst transmission the bus would not handle right away, i.e. the number of containers the IpduM has to hold and retransmit at a later point in time. See chapter Queue size considerations.

4.2.1.1.3 Understanding Triggers

There are several triggers that can be used for initiating a transmission to the bus. These define how many PDUs are collected into a container and when it is transmitted to the bus. All triggers "close" the container and create a new, empty container to be filled with PDUs – with one exception: First Contained PDU Trigger.

- SendTimeout (container or contained PDU): when the first PDU is received from the upper layer and put into the container, the send timeout timer starts. If a newly added PDU has a lower send timeout than the currently running timer, it is reduced to fit the new time. Once the send timeout elapses, the container is triggered from the Main Function.
- Size Threshold: when the PDUs including their headers exceed this threshold (reaching it is not enough), the container is triggered from the context of the transmission of the last PDU that was added.
- Trigger Always: when this PDU is added to the container, the container is triggered from the context of this PDU's transmission.
- First Contained PDU Trigger: this is mainly for time-triggered busses. When the first PDU is added to the container, an empty transmission is sent to the bus informing it that new data is available. The container is not triggered and remains open for more PDUs to be added while it is waiting.

> Overfilling: when more PDUs are added to the container than it can hold, the container is triggered from the context of the transmission of the PDU that led to the overflow.

4.2.1.1.4 Priorities

Contained PDUs can be assigned a priority. If no priorities are assigned, the IpduM handles them in a regular FIFO pattern. If priorities are assigned, however, the PDUs with higher priority are added to a container first while it is assembled for transmission. 0 is the highest priority, 255 the lowest priority. PDUs of the same priority are handled in a FIFO manner.

4.2.1.1.5 Static vs. Dynamic Container PDUs

The above applies for dynamic container PDUs. For static PDUs there are some differences.

> Each PDU has a static location within its container which is defined by the contained PDU offset.

> Collection semantics “last is best” is the only available option. With static positions there can be no FIFO behavior.

> Either Update Bits or checking for the unused areas default pattern can be used to determine whether a static position contains data or is empty in a received container. Update bits have to be outside of any PDU’s static location within the container. So this takes 1 bit of extra space in the container per PDU.

4.2.1.2 Rx Container / Rx Contained

4.2.1.2.1 Accepting PDUs

A container PDU can be configured to either accept any PDU in it that is known to the IpduM, or only the ones that are configured in it with the ContainedRxInContainerPduRef. If configured to only accept configured PDUs, any unconfigured PDU will be dropped silently.

4.2.1.2.2 Processing Immediate/Deferred

A container PDU can either be configured to immediately unpack a received container and forward the PDUs to the upper layer, or to use the deferred processing in which it uses a queue and unpacks them in the context of the Main Function.

Note
When using deferred processing, the allowed size of the container is limited to the value configured. Immediate processing allows for processing larger PDUs because they are not stored in any buffer.

4.2.1.2.3 Static vs. Dynamic Container PDUs

The above applies for dynamic container PDUs. For static PDUs, there are some differences.

> Each PDU has a static location within its container defined by the contained PDU offset.

> Either Update Bits or checking for the unused areas default pattern can be used to determine whether a static position contains data or is empty in a received container.

Caution
When using static container layout (header size “none”) without update bits, the unused areas default pattern must be checked by the receiver’s application to make sure the PDU contains data. This is not done by IpduM. PDUs are forwarded as received.

4.2.1.3 Meta Data

Meta Data is transparently forwarded through IpduM container PDUs. It is appended to the end of container and contained PDUs. This is only supported for dynamic container PDUs.

Caution
Container PDUs with Meta Data may only contain exactly one PDU at any given time. The transmitting ECU has to ensure this. Reception of a container PDU with more than one PDU in it will lead to wrong Meta Data. It is recommended to use the “trigger always” trigger.

Caution
Meta Data is only supported in combination with the MICROSAR Classic CanIf.

5 Troubleshooting and configuration considerations

This chapter provides information for configuration of common use-cases and hints on common configuration challenges of the MICROSAR Classic IPDUM.

5.1 Container PDUs

5.1.1 Triggers and their effect on timing behavior

There are several ways of triggering a Container PDU:

> Contained PDU Send Timeout
> Container PDU Send Timeout
> Size Threshold
> Contained PDU Trigger Always
> Container PDU First Contained PDU Trigger
> Overfilling a Container PDU, i.e. trying to add more PDUs than what fits

All send timeout triggers are evaluated in the next main function cycle, except for a send timeout value of 0. All other triggers including send timeouts with a send timeout value of 0 are evaluated in the context of the function IpduM_Transmit.

The Container PDU First Contained PDU Trigger triggers the Container Pdu as soon as the first contained Pdu is put into it. This shall only be used with TriggerTransmit. Other contained Pdus can be added until the TriggerTransmit API was called for this Container.

The Size Threshold Trigger triggers in the context of the function IpduM_Transmit if the threshold is exceeded. Meeting the threshold is not enough – the length has to be larger than the threshold.

The Overfilling trigger is a non-configurable trigger and depends on runtime behavior. Whenever a PDU is about to be added to a Container PDU but does not fit anymore, the Container PDU is triggered immediately. This can lead to unplanned transmissions and usually occurs if too many PDUs are sent to the IpduM at a time.

5.1.2 Conditions for delaying transmission

There are several conditions that can lead to delayed transmissions.

Configured TxConfirmations block transmission until the previous transmission of the same PDU was either received or the TxConfirmationTimeout has elapsed. If a second transmission is triggered while the TxConfirmationTimeout is still running, the second Container PDU is put in the queue and transmission is retried in the next Main Function.

Lower layer returning E_NOT_OK blocks transmissions. If the lower layer returns E_NOT_OK for any reason – this can for example be an unavailable bus – the Container PDU is put in the queue and transmission is retried in the next Main Function. This can lead to excessive queue usage during a bus off event. Note there is no way to flush this queue. There will be retransmissions once the bus is available again. Everything that didn’t fit in the queue will be missing. The queue must be planned accordingly.

5.1.3 DET error 49 (IPDUM_E_QUEUEOVFL)

DET reporting should be enabled for IpduM during development if Container PDUs are used. If DET error 49 (IPDUM_E_QUEUEOVFL) is observed, PDUs are being lost due to queue overflows. Usually, increasing the queue size does not solve this problem. It just makes the time until the problem occurs longer. There are two common reasons for queue overflows in IpduM.

If the amount of PDUs configured to be transmitted in a Container PDU exceeds the amount of Container PDUs the bus can accept, either because of bus load or because of a fixed transmission schedule (e.g. on FlexRay), the queue will overflow no matter how large the queue size.

If the timings of the PDUs and the according Container PDU are carefully matched for a near-100% filled Container PDU every time and still queue overflows occur, it is usually because of hardware timer jitters or late Tx confirmations or events that temporarily block transmission of a Container PDU, like a bus-off or a negative response to a transmit call. The paragraph Timing behavior with high bus load discusses this case in more detail.

5.1.4 Timing behavior with high bus load

If every available timeslot of the bus must be used (in a time-triggered bus like FlexRay) or if the general bus load is near 100% (in any other bus), the contents of the Container PDUs must be well planned. A single over-filled Container PDU can lead to a chain of PDUs overflowing into the next Container PDU and mess up the timing.

First, it is planned which PDUs are going to be in which transmitted instance of a given Container PDU. Next, it is planned in which order they are put into the Container PDU. This ensures the trigger by Container overflow is predictable if it does occur.

Finally, it must be planned how the Container PDU is going to be triggered exactly when all the planned PDUs are in the Container PDU. In a scenario where every Container PDU is nearly full, a size threshold on the Container PDU or a “trigger always” trigger on the last PDU being added are good choices. The overfilling trigger should be avoided.

One thing to keep an eye on are cycle times and offsets of periodically transmitted PDUs. They often stack up. If for example there is a PDU that is transmitted every 40ms and another one that is transmitted every 100ms, they will be triggered at the same time every 200ms. This is where an overfilling trigger can occur. By using offsets, this problem can be solved. In this case an offset of 10ms would work while an offset of 20ms would not.

Finally, a periodic timeslot with enough free space in a Container PDU should always be kept to catch up on overfilling.

5.1.5 SecOC PDUs in IpduM Container PDUs

If the IpduM has Container Tx PDUs that contain SecOC PDUs, the collection semantics “last is best” must not be used. The current IpduM Implementation issues two trigger transmit calls to the upper layer for “last is best” PDUs. The SecOC however requires a reauthentication between consecutive trigger transmit calls to the same PDU, which usually does not happen fast enough because it depends on the SecOC main function being called between those two trigger transmit calls.

In summary: if “last is best” is used here, the SecOC will return a negative result for the second trigger transmit call and the IpduM will delete the PDU from the Container Tx PDU. Use the “queued” semantics.

5.1.6 When to use Tx PDU collection semantics “last is best”

Using the “last is best” collection semantics is useful in these cases:

> Some PDUs have new data more often than the according Container PDU is transmitted, so the Container PDU might transmit old data.

> Receiving the same PDU twice in a very short time poses a problem on the receiving ECU.

The “last is best” collection semantics should not be used in these cases:

> All transmitted PDUs must be received (for example if a counter value is used for detecting missing messages for security reasons).

> The Container PDU is triggered before data in its PDUs can get old, requiring an update.

> The Container PDU is triggered fast enough in relation to the PDUs, so duplicate PDUs in the same container are not to be expected.

Configuring a PDU “last is best” will make the ECU slower. It adds at least two function calls to every contained PDU for every transmission. Only use it when it is necessary.

5.1.7 Send Timeouts and Time Bases

The send timeouts in IpduM are implemented as “maximum send timeouts” and not as “minimum send timeouts”. Send timeouts are decremented in the main function at intervals configured in the according Rx/Tx time base.

In the extreme case of identical values for send timeout and time base, this means that the send timeout is started and will elapse immediately with the next call to main function.

It is suggested to configure send timeouts to at least 2x the value of the according time base. This way, even if the main function is called immediately after starting the send timeout, there will be enough time for other processes to finish.

5.1.8 Queue size considerations

First of all, queues are only necessary for FIFO-style handling, i.e. CollectionSemantics “queued”. The “last is best” PDUs have a fixed size. To find the correct queue size, the “worst case” of a burst transmission needs to be found. To analyze this, all the periodic transmissions in Com module going towards IpduM are analyzed, and where they “pile up”. If, for example, there is one PDU with period=100ms, one with 200ms and one with 400ms, they will all be transmitted at the same time every 400ms. This is the worst case in this example – 3 PDUs at the same time. Offsets need to be considered, too, they can ease the load. But they can also lead to unexpected piling up of transmissions.

Next, the non-periodic transmissions need to be analyzed, triggered either by the application or incoming from a bus through a gateway routing. What is the “worst case” there?

Once the “worst case” transmission for all PDUs going into the same Container PDU is found, the order they will be coming in needs to be analyzed. By checking each of these PDU’s configured triggers in IpduM, it can be determined how often this container will be triggered. This is the number of Container PDU instances the queue needs to be able to hold – the minimum viable queue size.

Something that should also be considered: how is the ECU expected to behave in case of a bus-off event? PDUs might be piling up during the beginning of a bus-off event and re-transmitted after the bus becomes available again. Is it ok to lose some of these? If not, the queue size probably needs to be increased to hold the Container PDUs piling up there. The correct number can only be determined with a test on the real ECU with the full bus.

5.1.9 Considerations for a Time-Triggered Bus

The container PDU feature is very dynamic by design. When using it together with a time-triggered bus with a defined schedule like e.g. FlexRay, some things must be considered.

5.1.9.1 Where are the PDUs coming from?

If every single PDU is a periodic PDU originating from the same ECU the IpduM is running on, never changing the timings and offsets, then only effects like long-running interrupts may delay a PDU long enough to cause visible effects. In such a case, an occasional unused time slot every 10-20 times a PDU is scheduled may be enough.

If there are PDUs with a non-periodic transmission mode or coming from a different ECU, then a lot more free space should be planned to accommodate those effects. While a queue can spread the “extra” unplanned PDUs over a longer period, the free space in containers or empty time slots will be necessary to empty the queue again.

5.1.9.2 Which triggers should be used?

As time-triggered busses fetch data using triggertransmit, there is no use for a trigger in IpduM. The “ContainerFirstContainedPduTrigger”, however, is special here because it sends an informative transmit request to the lower layer while keeping the container open for more data. This is the only trigger that should be used with a time-triggered bus. Other triggers may interfere and lead to partially filled containers on the bus.

5.1.10 Considerations for Container PDUs with close to 100% payload

5.1.10.1 When are the PDUs transmitted to IpduM?

If the PDUs have a defined order and always are transmitted in the same order, then this order can be used to plan which PDUs are expected to be in each Container. The last

PDU added to the container will then have to trigger its transmission. Here, “trigger never” for the normal PDUs and “trigger always” for the last PDU can be a good choice.

If the PDUs do not have a defined order, e.g. if they are received from an external bus, then enough free space has to be planned to accommodate transmission bursts. As the order of the PDUs is not always identical, a “size threshold” or “send timeout” trigger can be a good choice here to allow the PDU to fill up, but not delay transmission too much.

Without a defined order, having PDUs of similar size can improve container usage. They will always fill the container the same way no matter the order the PDUs came in.

5.1.10.2 Which PDUs have the highest priority?

If there are PDUs with high priority, there are two ways to handle them.

When using collection semantics “queued”, the high priority PDU should get the “trigger always” or a short “send timeout” trigger. This can lead to the container being only partially filled, but the important PDU is transmitted quickly.

When using collection semantics “last is best”, priorities can be added to PDUs. This will change the order of PDUs on transmission, making sure the high priority PDUs will go out first. But some free space needs to be reserved in the container to accommodate this more random filling. A good way to trigger such a container would be a “send timeout” trigger, and potentially “trigger always” for the most important PDUs.

5.1.11 IpduM PDUs inside IpduM PDUs

There are instances where IpduM PDUs need to be packed inside IpduM PDUs. This is possible, but there are some limitations.

> Multiplexed inside Multiplexed is not allowed
> Container inside Multiplexed is not allowed
> Multiplexed inside Container is allowed
> Container inside Container is allowed

This is configured by creating a routing path where the source and destination is the IpduM, going from IpduMTxRequest / IpduMContainerTxPdu to IpduMContainedTxPdu on TX, or from IpduMContainedRxPdu to IpduMRxIndication / IpduMContainerRxPdu on RX.

6 API Description

For an interfaces overview please see Figure 1-3.

6.1 Services provided by IPDUM

6.1.1 IpduM_InitMemory

Prototype	
`void **IpduM_InitMemory** (void)`	
Parameter	
void	none
Return code	
void	none
Functional Description	
The function initializes variables which cannot be initialized with the startup code.	
Particularities and Limitations	
> IpduM_Init() is not called yet. > The function is called by the Application.	
Expected Caller Context	
> The function must be called on task level.	

Table 6-1 IpduM_InitMemory

6.1.2 IpduM_Init

Prototype	
`void **IpduM_Init** (const IpduM_ConfigType* config)`	
Parameter	
`config`	> NULL_PTR in the IPDUM_CONFIGURATION_VARIANT_PRECOMPILE > Pointer to the IpduM configuration data in the IPDUM_CONFIGURATION_VARIANT_POSTBUILD_LOADABLE
Return code	
void	none
Functional Description	
This function initializes the IpduM and performs configuration consistency checks. If the initialization is performed successfully the IpduM is initialized.	
Particularities and Limitations	
> IpduM_InitMemory() has been executed, if the startup code does not initialize variables. The IpduM is in the state uninitialized. > No IpduM function shall not pre-empt IpduM_Init()	
Expected Caller Context	

> The function is used by the Ecu State Manager.

Table 6-2 IpduM_Init

6.1.3 IpduM_Transmit

Prototype	
`StdReturnType` **`IpduM_Transmit`** `(PduIdType PdumTxPduId, const PduInfoType* PduInfoPtr)`	
Parameter	
`PdumTxPduId`	ID of transmit request (static or dynamic part). Identifies the payload.
`PduInfoPtr`	Payload information of the I-PDU (pointer to data and data length).
Return code	
`StdReturnType`	E_OK: The request was accepted by the IpduM and by the destination layer. E_NOT_OK: The request was rejected because: the IpduM is not initialized or the provided PduInfoPtr is NULL or the SduDataPtr of the PduInfoPtr is NULL or the PdumTxPduId is not in the expected range or the destination layer did not accept the transmission request.
Functional Description	
The function serves to request the transmission of a static or dynamic part.	
Particularities and Limitations	
> The IpduM_Init() has been executed successfully. > The function is non-reentrant for the same PdumTxPduId.	
Expected Caller Context	
> The function is called by the PduR. > The function is called in an interrupt context and at the task level	

Table 6-3 IpduM_Transmit

6.1.4 IpduM_MainFunction(Rx|Tx)(_<EcucPartitionShortname>)

Prototype	
`void` **`IpduM_MainFunction(Rx|Tx)(_<EcucPartitionShortname>)`** `(void)`	
Parameter	
`void`	none
Return code	
`void`	none
Functional Description	
This function performs: > Processing of IpduM counters and timeouts > Deferred processing of queued container Pdus	
Particularities and Limitations	
> The IpduM_Init() has been executed successfully. > The function is called cyclically by the BSW scheduler	
Expected Caller Context	
> The function is called on task level	

Table 6-4 IpduM_MainFunction

6.1.5 IpduM_GetVersionInfo

Prototype	
`void` **`IpduM_GetVersionInfo`** `(Std_VersionInfoType* versioninfo)`	
Parameter	
`versioninfo`	Pointer to the structure to write the version info to.
Return code	
`void`	none
Functional Description	
Returns the version information of the IpduM.	
Particularities and Limitations	
> none	
Expected Caller Context	
> The function can be called on interrupt and task level. > The function is called by the Application.	

Table 6-5 IpduM_GetVersionInfo

6.1.6 IpduM_CalculateSizeOfContainer

Prototype	
`uint32` **`IpduM_CalculateSizeOfContainer`** `(PduLengthType headerSize, const PduInfoType* PduInfoPtr)`	
Parameter	
`headerSize`	Size of the header.
`PduInfoPtr`	Payload information of the I-PDU (pointer to data and data length).
Return code	
`uint32`	Size of the container PDU.
Functional Description	
Returns the size of the container PDU given in PduInfoPtr.	
Particularities and Limitations	
> none	
Expected Caller Context	
> The function can be called on interrupt and task level. > The function is called by Microsar Classic SecOC.	

Table 6-6 IpduM_CalculateSizeOfContainer

Caution
The function IpduM_CalculateSizeOfContainer may only be called by Microsar Classic SecOC. See safety manual.

6.2 Services used by IPDUM

In the following table, services provided by other components which are used by the IPDUM are listed. For details about prototype and functionality refer to the documentation of the providing component.

Component	API
Det	Det_ReportError
Det	Det_ReportRuntimeError
PduR	PduR_IpduMTransmit
PduR	PduR_IpduMTxConfirmation
PduR	PduR_IpduMRxIndication
PduR	PduR_IpduMTriggerTransmit
BSW Scheduler	SchM_Enter_IpduM
BSW Scheduler	SchM_Exit_IpduM

Table 6-7 Services used by the IPDUM

6.3 Callback Functions

This chapter describes the callback functions that are implemented by the IPDUM and can be invoked by other modules. The prototypes of the callback functions are provided in the header file `IpduM_Cbk.h` by the IPDUM.

6.3.1 IpduM_RxIndication

Prototype	
`void` **`IpduM_RxIndication`** `(PduIdType RxPduId, const PduInfoType* PduInfoPtr)`	
Parameter	
`RxPduId`	ID of received multiplexed I-PDU. Identifies the payload.
`PduInfoPtr`	Contains the length (SduLength) of the received I-PDU and a pointer to a buffer (SduDataPtr) containing the I-PDU.
Return code	
void	none
Functional Description	
This service indicates the complete reception of an IpduM I-PDUs. It de-multiplexes the received multiplexed I-PDU.	
Particularities and Limitations	
> The IpduM_Init() has been executed successfully > The function is non-reentrant for the same PdumRxPduId.	
Expected Caller Context	
> The function is called by the PduR. The function is called in an interrupt context or at the task level	

Table 6-8 IpduM_RxIndication

6.3.2 IpduM_TxConfirmation

Prototype	
`void` **`IpduM_TxConfirmation`** `(PduIdType TxPduId)`	
Parameter	
`TxPduId`	ID of transmitted multiplexed IPDU.
Return code	
`void`	none
Functional Description	
This service confirms the transmissions of the dynamic and static parts contained in the multiplexed IPDU whose transmission is confirmed.	
Particularities and Limitations	
> The IpduM_Init() has been executed successfully > The function is non-reentrant for the same PdumTxPduId.	

Expected Caller Context
> The function is called by the PduR. The function is called in an interrupt context or at the task level

Table 6-9 IpduM_TxConfirmation

6.3.3 IpduM_TriggerTransmit

Prototype	
`Std_ReturnType` **`IpduM_TriggerTransmit`** `(PduIdType TxPduId, PduInfoType* PduInfoPtr)`	
Parameter	
`TxPduId`	ID of transmission multiplexed IPDU.
`PduInfoPtr`	Contains a pointer to a buffer (SduDataPtr) to where the SDU data shall be copied, and the available buffer size in SduLengh. On return, the service will indicate the length of the copied SDU data in SduLength.
Return code	
`Std_ReturnType`	E_OK: SDU has been copied and SduLength indicates the number of copied bytes. E_NOT_OK: No data has been copied, because IpduM is not initialized or TxPduId is not valid or PduInfoPtr is NULL_PTR or SduDataPtr is NULL_PTR or SduLength is to small or the SDU data could not be updated because the just in time update via a trigger transmit call of the upper layer was not successful for multiplexed messages.
Functional Description	
This service copies the transmission multiplexed IPDU to the provided buffer.	
Particularities and Limitations	
> The IpduM_Init() has been executed successfully > The function is non-reentrant for the same TxPduId	
Expected Caller Context	
> The function is called by the PduR module. The function is called in an interrupt context or at the task level	

Table 6-10 IpduM_TriggerTransmit

7 Configuration

The IPDUM in the EcuC can be configured through the Vector configuration and generation tool CFG5.

7.1 Configuration of Post-Build

The configuration of post-build loadable is described in [3].

8 AUTOSAR Standard Compliance

8.1 Deviations

No deviations.

8.2 Additions/ Extensions

No additions/ extensions.

8.3 Limitations

See Table 2-2 Not supported AUTOSAR standard conform features.

During initialization of Multiplexed PDUs on ECU startup, the IpduM uses the IpduM UnusedAreasDefault instead of the Com UnusedAreasDefault. Configure them to identical values to avoid any issues.

Routing Paths with Multiplexed PDUs must have the Microsar Classic Com module as the upper layer. Other upper layer modules are not supported due to the way initial data for Multiplexed PDUs are gathered during code generator runtime.

9 Glossary and Abbreviations

9.1 Glossary

Term	Description
BSWMD	The BSWMD is a formal notation of all information belonging to a certain BSW artifact (BSW module or BSW cluster) in addition to the implementation of that artifact.
Buffer	A buffer in a memory area normally in the RAM. It is an area that the application has reserved for data storage.
CFG5	Configurator 5 (configuration and generation tool)
Component	CAN Driver, Network Management ... are software COMPONENTS in contrast to the expression module, which describes an ECU.
Confirmation	A service primitive defined in the ISO/OSI Reference Model (ISO 7498). With the service primitive 'confirmation' a service provider informs a service user about the result of a preceding service request of the service user. Notification by the CAN Driver on asynchronous successful transmission of a CAN message.
Electronic Control Unit	Also known as ECU. Small embedded computer system consisting of at least one CPU and corresponding periphery which is placed in one housing.
Event	An exclusive signal which is assigned to a certain extended task. An event can be used to send binary information to an extended task. The meaning of events is defined by the application. Any task can set an event for an extended task. The event can only be cleared by the task which is assigned to the event.
Interrupt	Processor-specific event which can interrupt the execution of a current program section.
Post-build	This type of configuration is possible after building the software module or the ECU software. The software may either receive parameters of its configuration during the download of the complete ECU software resulting from the linkage of the code, or it may receive its configuration file that can be downloaded to the ECU separately, avoiding a re-compilation and re-build of the ECU software modules. In order to make the post-build time re-configuration possible, the re-configurable parameters shall be stored at a known memory location of ECU storage area.
Scheduler	The algorithm which decides whether a task switch is to be affected and which triggers all necessary internal activities of the operating system is named scheduler. The scheduler decides whether a task switch is possible according to the implemented scheduling policy. The scheduler can be considered as a resource which can be occupied and released by tasks. Thus a task can reserve the scheduler to avoid task switch until the scheduler is released.
Signal	A signal is responsible for the logical transmission and reception of information depending on the restrictions of the physical layer. The definition of the signal contents is part of the database given by the vehicle manufacturer. Signals describe the significance of the individual data segments within a message. Typically bits, bytes or words are used

	for data segments but individual bit combinations are also possible. In the CAN data base, each data segment is assigned a symbolic name, a value range, a conversion formula and a physical unit, as well as a list of receiving nodes.

Table 9-1 Glossary

9.2 Abbreviations

Abbreviation	Description
API	Application Programming Interface
AUTOSAR	Automotive Open System Architecture
BSW	Basis Software
CAN	Controller Area Network protocol originally defined for use as a communication network for control applications in vehicles.
Com	AUTOSAR BSW module Com
COM	Communication
DEM	AUTOSAR BSW module Diagnostic Event Manager
DET	AUTOSAR BSW module Development Error Tracer
ECU	Electronic Control Unit
EcuM	AUTOSAR BSW module ECU Manager
HIS	Hersteller Initiative Software
ID	Identifier (e.g. Identifier of a CAN message)
I-PDU	Interactive layer Protocol Data Unit
IpduM	AUTOSAR BSW module I-PDU Multiplexer
IPDUM	I-PDU Multiplexer
MICROSAR	Microcontroller Open System Architecture (the Vector AUTOSAR flavor)
PCI	Protocol Control Information
PDU	Protocol Data Unit
PduR	AUTOSAR BSW module Pdu-Router
RAM	Random Access Memory
ROM	Read-Only Memory
SDU	Service Data Unit
SWS	Software Specification

Table 9-2 Abbreviations

10 Contact

Visit our website for more information on

- News
- Products
- Demo software
- Support
- Training data
- Addresses

www.vector.com