IPCF Shared Memory Driver User Manual

NXP

May 20, 2024

CONTENTS

CHAPTER

ONE

INTER-PLATFORM COMMUNICATION FRAMEWORK (IPCF)

1.1 Software Product Overview

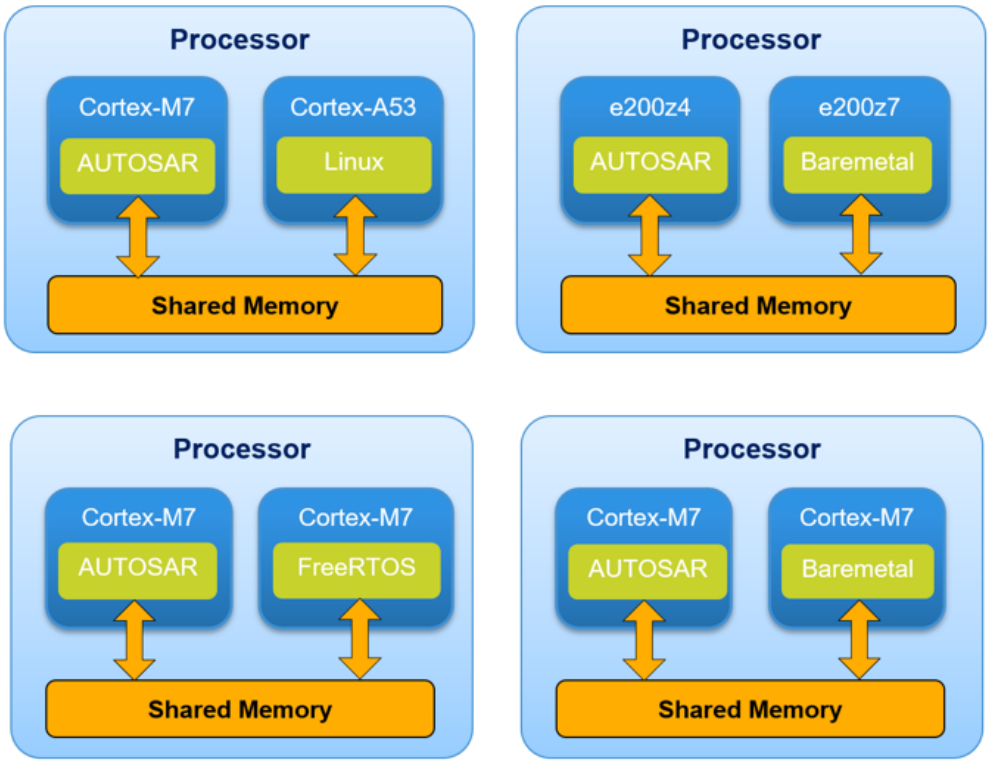
Inter-Platform Communication Framework (IPCF) is a subsystem which enables applications, running on multiple homogenous or heterogenous processing cores, located on the same chip or different chips, running on different operating systems (AUTOSAR, Linux, FreeRTOS, Zephyr, etc.), to communicate over various transport interfaces (Shared Memory, etc.).

IPCF is designed for NXP embedded systems and features low-latency and a tiny-footprint. It exposes a zero-copy API that can be directly used by customers for maximum performance, minimum overhead and low CPU load. The driver ensures freedom from interference between local and remote shared memory by executing all writing operations only in the local memory domain. Customers can enforce memory protection for their software with XRDC/SMPU peripherals.

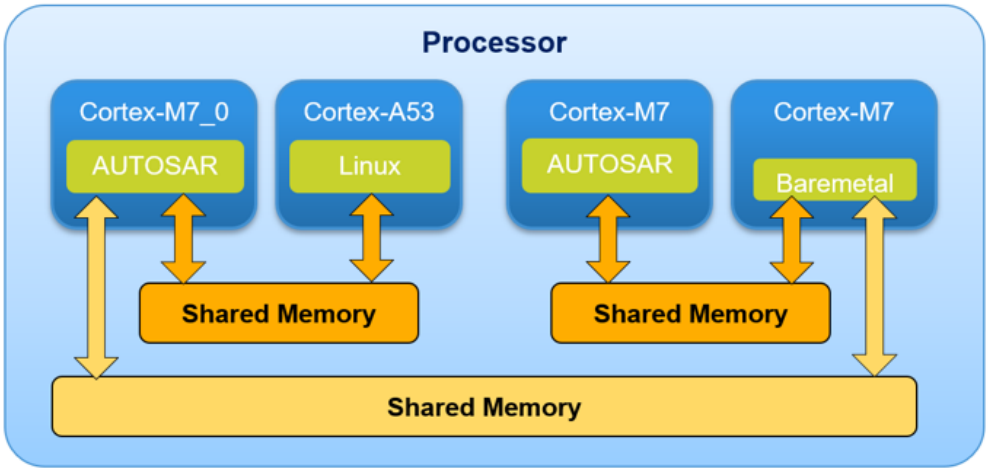
Customers can choose to build exactly what they need in terms of HW, OS and transport interface.

1.2 Use cases

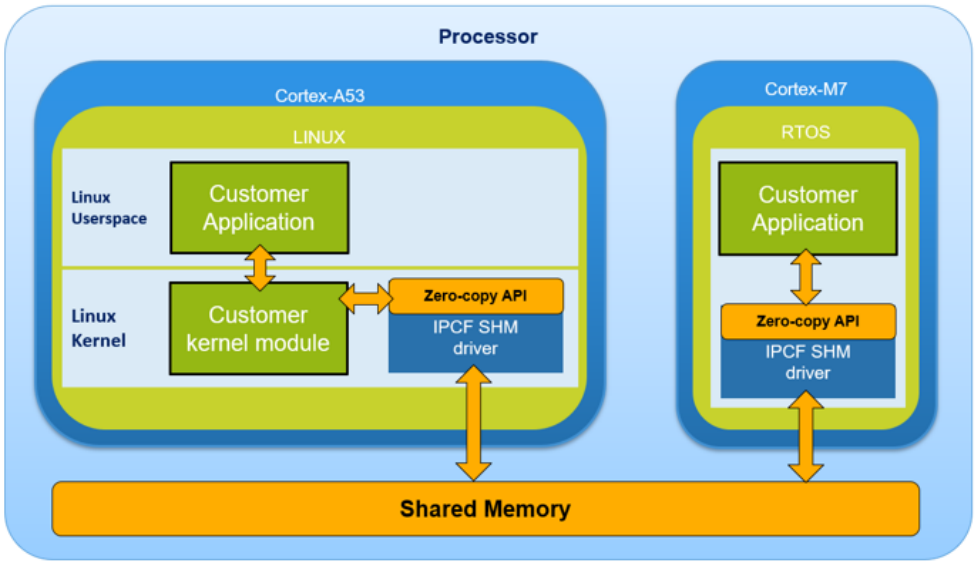
The following diagram illustrates some use-cases addressed by IPCF.



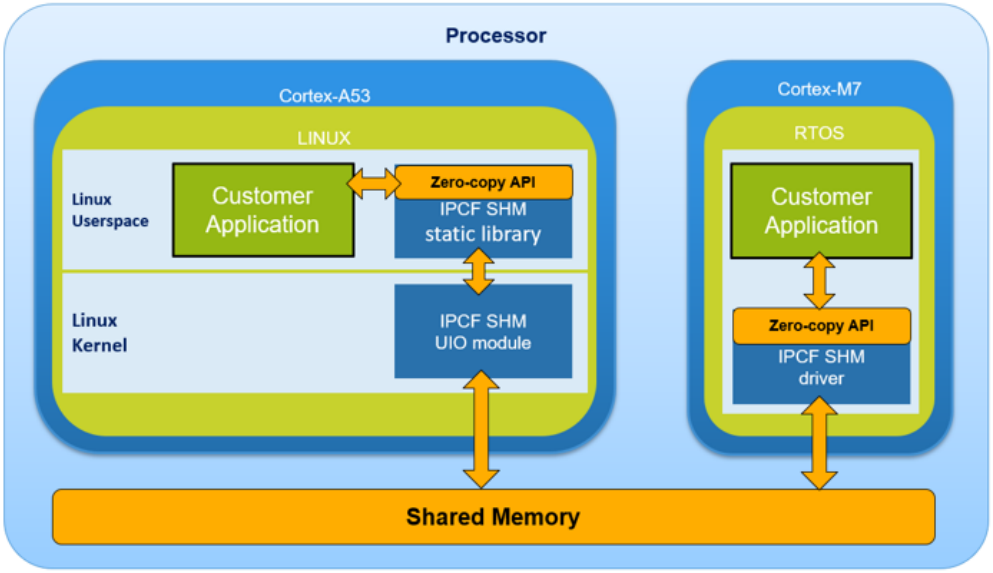
IPCF use cases on multiple homogenous or heterogenous processing cores.



IPCF use case with multiple instances.



IPCF use case for Linux kernel.



IPCF use case for Linux user-space.

1.3 Software Content

The IPCF software package contains the communication driver over shared memory supporting AUTOSAR, FreeRTOS, Zephyr and baremetal. For Linux support the IPCF software package is published on GitHub.

The driver is accompanied by sample applications which demonstrate a ping-pong message communication (for more details see the samples readme file).

The IPCF software package contains a Configuration Tool component used in NXP S32 Design Studio that is quick and easy to use. This component is installed over NXP S32 Real Time Drivers releases and can be used with all Real Time Drivers components.

The IPCF Shared Memory Driver for Linux and sample application are integrated as out-of-tree kernel modules in NXP Auto Linux BSP. The IPCF driver is provided in Yocto images from NXP Auto Linux BSP (fsl-image-auto) but can also be built manually.

The IPCF Shared Memory Driver for Linux also provides an user-space static library in case the customer application requires IPCF usage from user-space.

The source code and samples of IPCF implementation for Linux drivers are published on:

```
https://github.com/nxp-auto-linux/ipc-shm
```

The IPCF software package also contains:

Release Notes (information about release):

1. Supported platforms
2. Software dependencies
3. Validated compilers
4. Instructions about installation steps
5. New features
6. Known limitations
7. Licensing and support

IPCF Driver User Manual:

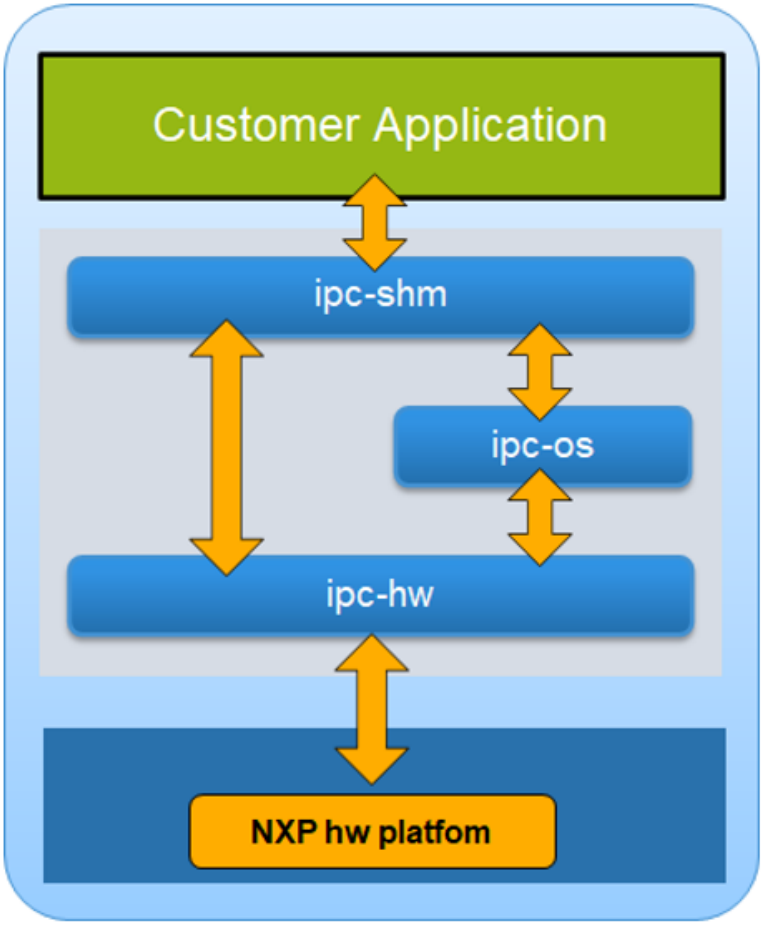
1. Installation instruction for different integration
2. Driver usage, compile and configuration for different OSes and platforms
3. Instructions to compile, build and run the sample application
4. Description of the driver's APIs

Quality Package - delivered to customers for RTM releases

1.4 Architecture

The IPCF driver contains the next layers:

- Shared memory generic implementation that is HW and OS agnostic
- HW abstraction component: abstraction over various HW IP modules (MSCM, INTC ...)
- OS abstraction component: OS agnostic API for common OS services

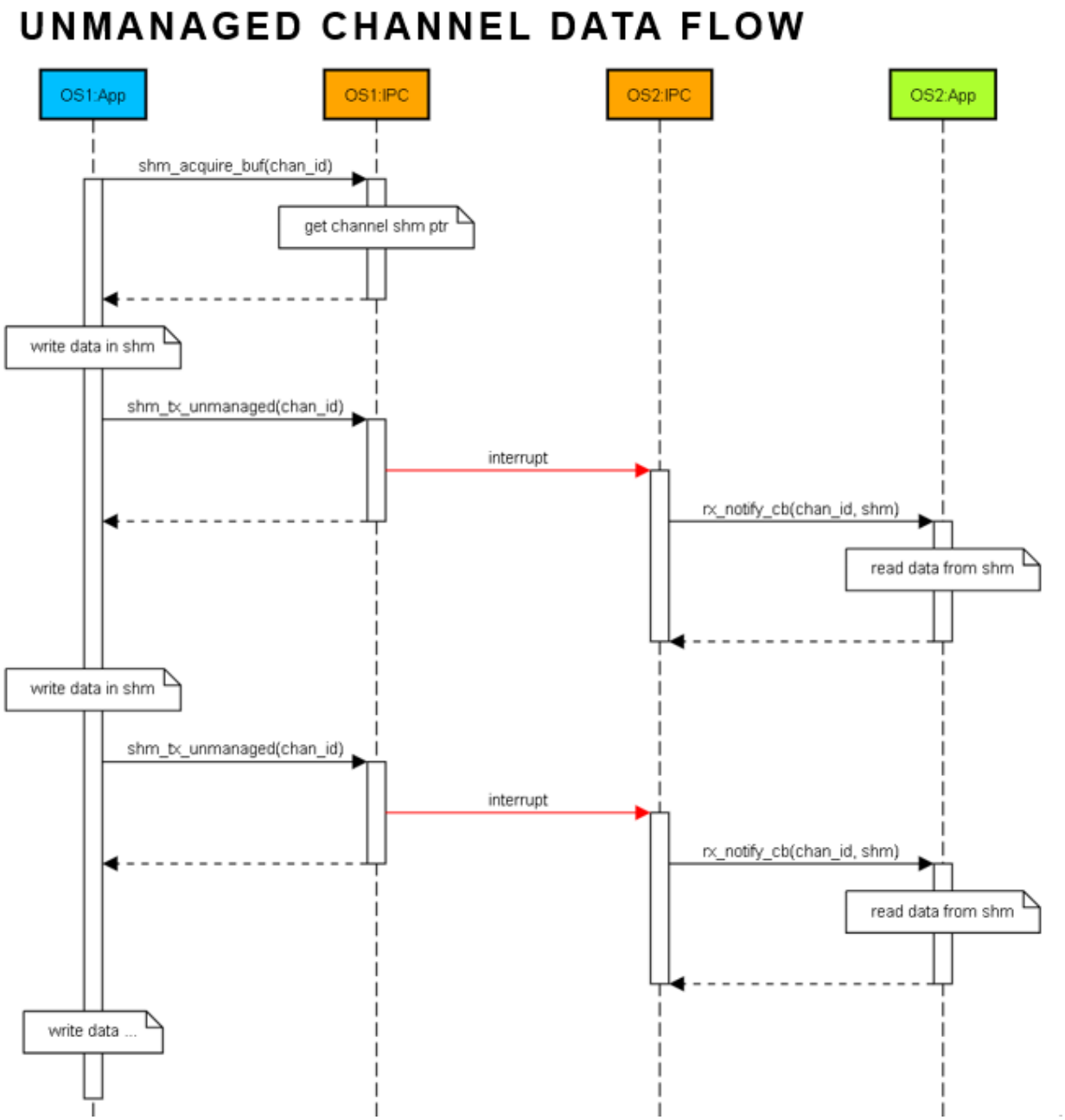


IPCF System Architecture.

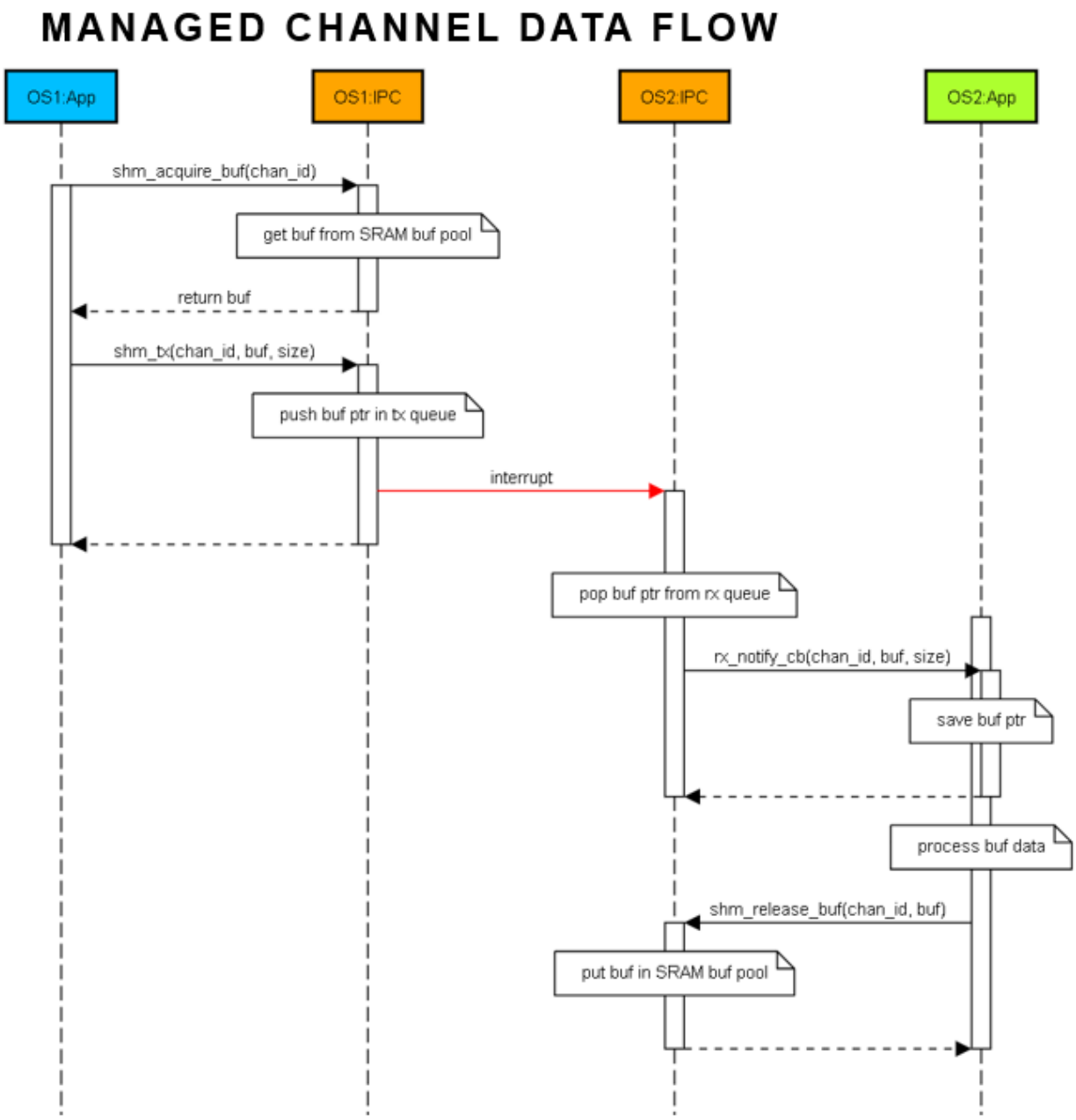
1.5 Driver details

The IPCF driver uses for buffer management:

- **unmanaged channel data flow**: (Similar to POSIX ShM) buffer management is disabled, and application owns the entire channel memory; use-case example: video streaming or non-critical data exchange. Each App owns half of the channel memory.
- **managed channel data flow**: memory is split in buffer pools and buffer management is controlled by driver; use-case example: CAN forwarding or flash update. Can handle multiple data flows simultaneous.



IPCF unmanaged channel data flow.



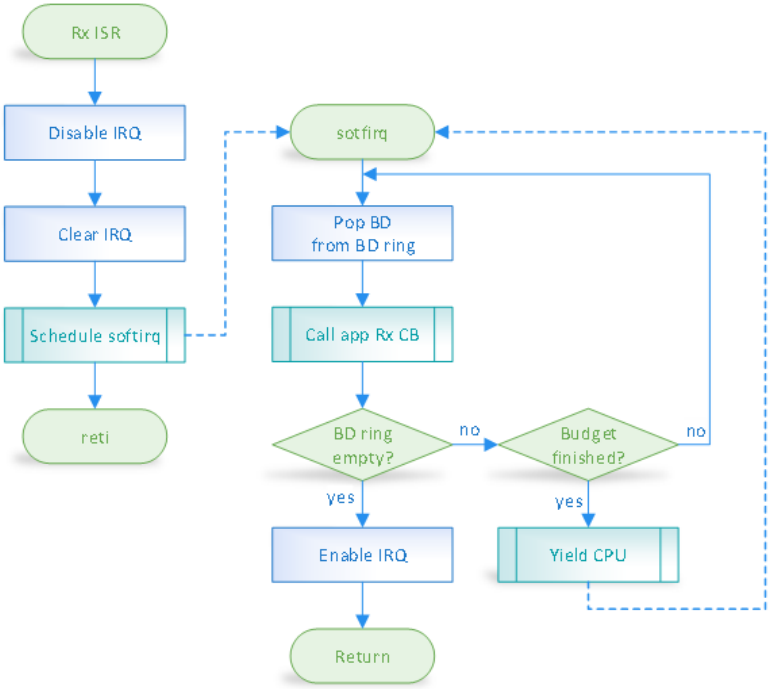
IPCF managed channel data flow.

The IPCF driver supports the following inter-core notification methods:

- intercore interrupts (hardware notification)
- polling method (sending and receiving is managed by user)

The IPCF driver reduces the receive interrupt overhead using the interrupt coalescing technique thus avoiding storming interrupts.

When a receive interrupt is triggered the code disables the interrupt, processes all buffer descriptors from the receive FIFO and then reenables the interrupt.

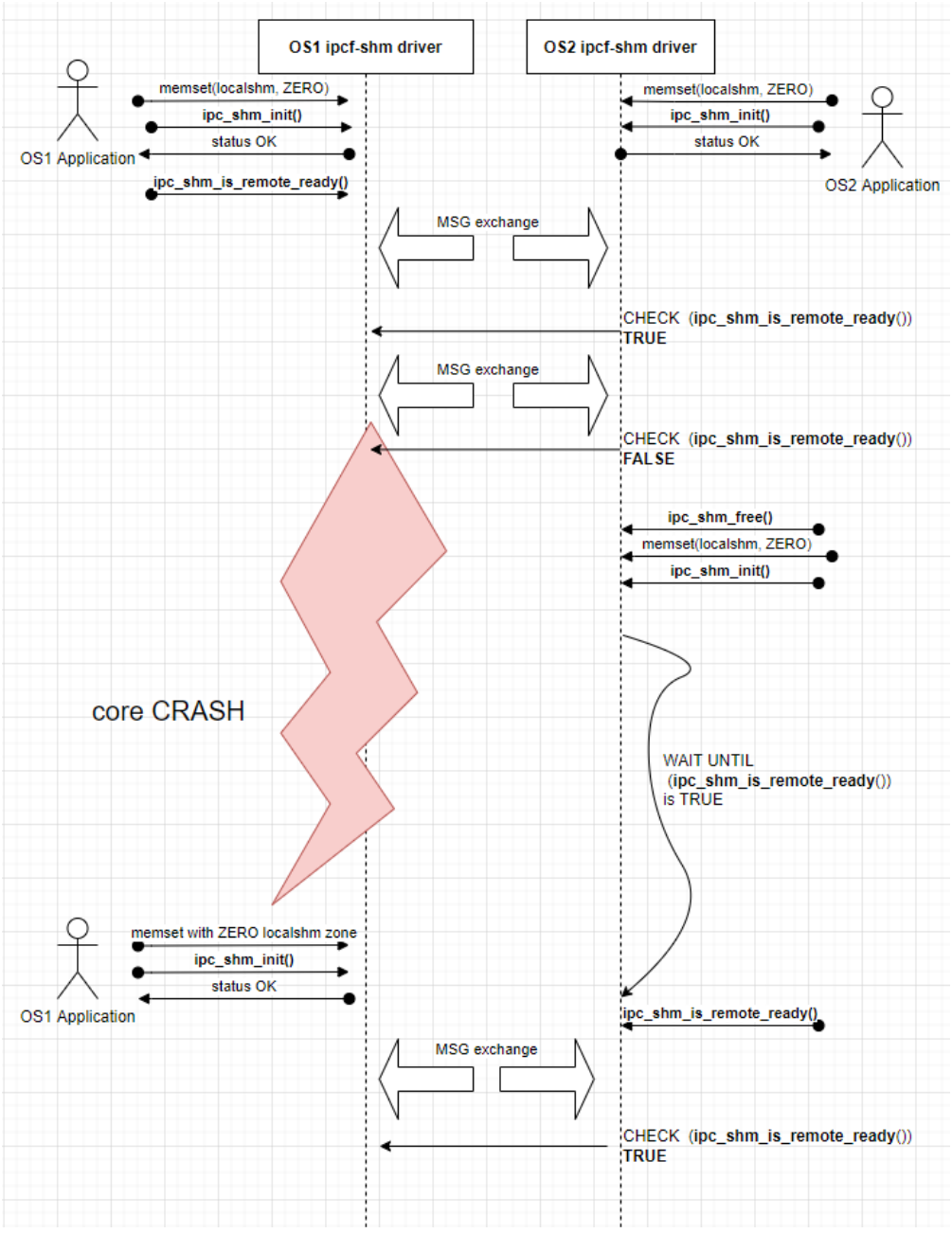


IPCF mitigation techniques.

1.6 IPCF sample application

This is the IPCF Shared Memory driver sample application. It demonstrates a ping-pong message communication using the shared memory driver.

One application initializes the driver, waits until the remote application is ready and does an echo reply for every message received. The other application sends a number of messages and waits for the reply. If a core crashes or the application crashes the other application detects this and re-initializes all instances.



IPCF sample diagram

CHAPTER

TWO

IPCF SHARED MEMORY DRIVER FOR LINUX

2.1 Overview

The Linux IPCF Shared Memory Driver enables communication over shared memory with an RTOS application running on a different core of the same processor.

The driver is accompanied by a sample application which demonstrates a ping-pong message communication with a RTOS application (for more details see the readme from the sample directory).

The driver is integrated as an out-of-tree kernel module in NXP Auto Linux BSP.

The source code of this Linux driver is published on github.com.

2.2 HW platforms

The supported processors are listed in the sample application documentation.

2.3 Configuration notes

There are five hardware-related parameters that can be configured at the driver API level: TX and RX inter-core interrupt IDs, local core ID, remote core ID and trusted cores.

The interrupt IDs are MSCM core-to-core directed interrupt IDs. Valid values are in range [0..2] for S32xx, [0..11] for S32G3xx and [0..3] for S32V234. The TX and RX interrupt IDs must be different.

Note: the interrupt ID expected in the driver configuration is different from the corresponding processor exception number (used to register the interrupt handler). See the Reference Manual of each platform for specific information.

The TX interrupt can be disabled by setting its ID to IPC_IRQ_NONE. When disabled the remote application must check for incoming messages by invoking the function ipc_shm_poll_channels().

The local and remote core IDs configuration is divided into core type and core index. Supported values for core type and index are defined in ipc_shm_core_type and ipc_shm_core_index enum.

For ARM platforms, a default value can be assigned to the local and/or remote core IDs by choosing IPC_CORE_DEFAULT as the core type. When using this default value the core index is automatically chosen by the driver.

Note: see the Reference Manual of each platform for the core types and indexes supported.

The remote core ID specifies the remote core to be interrupted and the local core ID specifies the core targeted by the remote core interrupt. The local core ID configuration is only applicable to platforms where Linux may be running SMP and the interrupt could be serviced by a different core than the targeted (e.g. interrupt load balancing).

The trusted cores mask specifies which cores (of the same core type as local core) have access to the inter-core interrupt status register of the local core targeted by remote. The mask can be formed from valid values defined in ipc_shm_core_index enum.

Note: the local core ID and trusted cores configuration is only applicable to S32xx platforms running Linux. For other platforms or Operating Systems, the local core ID configuration is not used.

If using Linux IPCF Shared Memory User-space Driver, the user-space static library (libipc-shm) will automatically insert the IPCF UIO/CDEV kernel module at initialization. The path to the kernel module in the target board rootfs can be overwritten at compile time by setting IPC_UIO_MODULE_DIR (for using uio driver) or IPC_CDEV_MODULE_DIR (for using character driver) variable from the caller.

2.4 Cautions

This driver provides direct access to physical memory that is mapped as non-cachable in both kernel-space or user-space. Therefore, applications should make only aligned accesses in the shared memory buffers. Caution should be used when working with functions that may do unaligned accesses (e.g., string processing functions).

The driver ensures freedom from interference between local and remote memory domains by executing all write operations only in the local memory.

The driver is thread safe as long as only one thread is pushing and only one thread is popping: Single-Producer - Single-Consumer.

This thread safety is lock-free and needs one additional sentinel element in the ring buffers between the write and read indexes that is never written.

The driver is thread safe for different instances but not for same instance.

For technical support please go to:
https://www.nxp.com/support

CHAPTER

THREE

IPCF SHARED MEMORY DRIVER FOR REAL-TIME OPERATING SYSTEMS

3.1 Overview

The IPCF Shared Memory driver for RTOS enables communication over shared memory with another application running on a different core of the same processor. This driver is part of the Inter-Platform Communication Framework (IPCF).

The driver is accompanied by a sample application which demonstrates a ping-pong message communication with another sample application (for more details see the readme from the sample directory).

3.1.1 HW Platforms

The supported processors are listed in the release notes document.

3.1.2 SW Platforms

The supported SW platforms are listed in the release notes document.

The compilers used for driver validation are listed in the release notes document.

3.2 Integration with RTOS

To integrate this driver into a real-time application import ipc-shm-rtos.mk in the application makefile.

The following variables must be set by the caller makefile:

- SHM_PLATFORM - hardware platform to build the driver for
- SHM_OS_TARGET - target RTOS to build the driver for
- SHM_DRIVER_PATH - path to the driver directory

The ipc-shm-rtos.mk makefile will produce the next variables:

- SHM_DRIVER_SRC_DIR - source driver directories
- SHM_DRIVER_INCLUDES_DIRS - included driver directories
- SHM_DRIVER_INCLUDE_FILES - list driver include files
- SHM_DRIVER_SOURCE_FILES - list driver source files

- SHM_DRIVER_OUT_FILES - list driver object files

The compiler, assembler and linker flags used for building the IPCF driver are from NXP RTD. The driver doesn't need any additional flags.

The name of the interrupt handler must be **ipc_shm_hardirq** when MSCM is used and the callback function from RX channel **ipc_shm_mru_notification** when the MRU is used

3.2.1 Integration with NXP RTOS

The same steps apply for integration with any Autosar compliant OS:

- a category 2 ISR must be registered for each instance for the configured RX external IRQ, moreover the ISR attribute IsrFunction must be named **ipc_shm_hardirq** when MSMC is used
- an ISR must be registered for the configured MU RX IRQ with the following handler name: **ipc_shm_mu_notification** when MU is used
- an interrupt notification function must be set for the receiving channel named **ipc_shm_mru_notification** when the MRU is used
- an extended, non-preemptive task, without autostart and with higher priority than other tasks using shared memory driver must be configured with the following name: **ipc_shm_softirq**
- **two events must be configured to be used in ipc_shm_softirq task:**
  - IPC_EVENT_RX_IRQ: triggered when a message has been received from the remote core
  - IPC_EVENT_OS_FREE: triggered by the user application to call ipc_shm_free()

Note: the user application must not interfere with any of the OS objects above, other than the configuration of ISR and task priorities and the task stack size

Integration with AUTOSAR Run Time Environment (RTE) - only for S32ZE platform

This feature only has EAR quality level.

The folder **src\rte_integration** contains a wrapper over the driver called IpcShm and the corresponding arxml files (IpcShm_Bswmd.arxml, IpcShm_Services.arxml and IpcShm_Types.arxml) that can be used to integrate the driver with the Autosar RTE following the steps:

- import the IpcShm arxml files in the Autosar Toolchain (e.g.: Autosar Builder) to create and connect the Required Ports in the module that will use the driver. IpcShm provides the following port interfaces: IpcShm_InitInstance, IpcShm_FreeInstance, IpcShm_AcquireBuffer, IpcShm_ReleaseBuffer, IpcShm_TransmitBuffer and IpcShm_IsRemoteReady
- import the IpcShm arxml files in the RTE generator (e.g.: Elektrobit Tresos Studio)
- in the RTE configuration add a SwComponentInstance for the IpcShm (e.g.: IpcShm_Prototype) and map it to the desired Os Application
- in the RTE configuration add a BswModuleInstance for the IpcShm (e.g.: BSW_IpcShm) and map it to the provided IpcShm implementation (Impl_IpcShm). Map the two timming events (TimingEvent_MainFunction and TimingEvent_Init) to the desired Tasks (e.g.: a 1ms cyclic task)
- generate all the RTE files
- use the generated defines for the Required Port calls in the application to interract with the driver (e.g.: Rte_Call_RP_IpcShm_AcquireBuffer_Acquire)

3.2.2 Integration with FreeRTOS

For integration with FreeRTOS:

- an ISR must be registered for the configured RX external IRQ with the following handler name: **ipc_shm_hardirq** when MSMC is used
- an ISR must be registered for the configured MU RX IRQ with the following handler name: **ipc_shm_mu_notification** when MU is used
- an interrupt notification function must be set for the receiving channel named **ipc_shm_mru_notification** when the MRU is used
- a task with IPC_SOFTIRQ_PRIORITY priority must be created to be used by the shared memory driver and must be configured with the following name: **ipc_shm_softirq**
- the driver supports static and dynamic allocation (**configSUPPORT_STATIC_ALLOCATION** and **configSUPPORT_DYNAMIC_ALLOCATION**). In case both are selected then the task **ipc_shm_softirq** will be created using dynamic memory allocation

3.2.3 Integration with Zephyr

For integration with Zephyr:

- an ISR must be registered for the configured RX external IRQ with the following handler name: **ipc_shm_hardirq** when MSMC is used
- an ISR must be registered for the configured MU RX IRQ with the following handler name: **ipc_shm_mu_notification** when MU is used
- an interrupt notification function must be set for the receiving channel named **ipc_shm_mru_notification** when the MRU is used
- a thread with IPC_SOFTIRQ_PRIORITY priority and IPC_SOFTIRQ_STACK_SIZE stack size is created by the shared memory driver for deferred interrupt processing

3.2.4 Integration with XOS

For integration with XOS:

- an ISR must be registered for the configured RX external IRQ with the following handler name: **ipc_shm_hardirq** when MSMC is used
- an ISR must be registered for the configured MU RX IRQ with the following handler name: **ipc_shm_mu_notification** when MU is used
- an interrupt notification function must be set for the receiving channel named **ipc_shm_mru_notification** when the MRU is used
- a thread with IPC_SOFTIRQ_PRIORITY priority and IPC_SOFTIRQ_STACK_SIZE stack size is created by the shared memory driver for deferred interrupt processing

3.2.5 Integration in Baremetal

For integration in Baremetal:

- an ISR must be registered for the configured RX external IRQ with the following handler name: **ipc_shm_hardirq** when MSMC is used
- an ISR must be registered for the configured MU RX IRQ with the following handler name: **ipc_shm_mu_notification** when MU is used
- an interrupt notification function must be set for the receiving channel named **ipc_shm_mru_notification** when the MRU is used

3.3 Configuration Notes

There are five hardware-related parameters that can be configured at the driver API level: TX and RX inter-core interrupt IDs, local core ID, remote core ID and trusted cores.

The interrupt IDs are MSCM core-to-core directed interrupt IDs or MU/MRU interrupt sources. Users can only choose to use MSCM, MU or MRU driver for the corresponding interrupts between cores. The driver does not support using the same MRU channel for multiple instances running on the same core.

In case of using the MSCM core-to-core directed interrupt, the interrupt IDs for each platform can be selected from the RTD header files (ex: INT0_IRQn or MSCM_INT0_IRQn) or IPC_IRQ_NONE for the polling method.

The TX and RX interrupts can be disabled by setting their IDs to IPC_IRQ_NONE. When the RX (or TX) interrupt is disabled, the local (or remote) application must check for incoming messages by invoking the function ipc_shm_poll_channels(). Disabling both TX and RX interrupt is allowed.

The local and remote core IDs configuration is divided into core type and core index. Supported values for core type and index are defined in ipc_shm_core_type and ipc_shm_core_index enum. Local core ID and trusted cores configuration is reserved for use in Linux shared memory driver and has no effect in this implementation. Local and remote core IDs also have no effect when MU or MRU are used.

For ARM platforms a default value can be assigned to the remote core ID by choosing IPC_CORE_DEFAULT as the core type. When using this default value the core index is automatically chosen by the driver.

3.4 Cautions

The user must zero set the shared SRAM memory area before initializing the driver.

This driver provides direct access to physical memory that is mapped as non-cachable by default. To use cached memory the symbol IPC_D_CACHE_ENABLE needs to be defined.

Therefore, applications should make only aligned accesses in the shared memory buffers. Caution should be used when working with functions that may do unaligned accesses (e.g., string processing functions).

The driver ensures freedom from interference between local and remote memory domains by executing all write operations only in the local memory.

The driver is thread safe as long as only one thread is pushing and only one thread is popping: Single-Producer - Single-Consumer.

This thread safety is lock-free and needs one additional sentinel element in the ring buffers between the write and read indexes that is never written.

The driver is thread safe for different instances but not for same instance.

The driver ensures that a managed channel will no longer be used (by both sides) in case a memory overflow occurs and corrupts the buffer descriptor.

The driver ensures that an unmanaged channel will no longer be used (by both sides) in case a memory overflow occurs and corrupts the index.

The driver does not ensure the integrity and correctness of the data if the length exceeds the configured maximum length.

CHAPTER

FOUR

IPCF SHARED MEMORY INSTALLERS

4.1 Overview

The IPCF package contains a standalone installer and an update site for S32 Design Studio.

4.2 Standalone installer

To install the IPCF driver follow the steps:

1. Launch the standalone installer.
2. In the Welcome window click **Next**.
3. Read and accept the license terms. Click **Next** to complete the installation.
4. Choose what package to install and click **Next**.
5. Choose a destination folder and click **Install**.

   Default is *C:/NXP/IPCF_<version>*
6. After the installation is completed click **Finish**.
7. After installation IPCF can be added to a new Tresos project or the IPCF examples provided in software package can be used.

4.3 S32 Design Studio installer

IPCF delivered as an Update Site for S32 Design Studio. In this case it must be installed after the installation of the RTD package by following the next steps:

1. Launch S32 Design Studio and select a workspace.
2. Choose **Help** -> **Install New Software ...**
3. Click on **Add**, next **Archive ...** and select the update site file contain in the IPCF release.
4. Check the IPCF package to be installed and continue the installation process.
5. After installation IPCF can be added to a new S32DS project or the IPCF examples provided in software package can be used.

CHAPTER

FIVE

IPCF SHARED MEMORY SAMPLE APPLICATIONS

5.1 Overview

The IPCF package contains examples that demonstrate the features of the driver.

Details about building and running are available in the description.txt files provided for each example.

CHAPTER

SIX

IPCF SHARED MEMORY INTEGRATION

6.1 Configuration

The IPCF Shared Memory driver can be configured in S32 Design Studio and Tresos.

Users can modify the values for local_shm_addr, remote_shm_addr, shm_size. Those changes must also be reflected in the linker file. The shared SRAM must be SHAREABLE and NON-CACHEABLE. If the shared memory is cacheable then the user must invalidate the cache before using the IPC send functions.

If users select IRQ_NONE-POLLING for inter_core_tx_irq or inter_core_rx_irq the polling method is used (see ipc_shm_poll_channels API function). For better performance it is recommended to select an inter-core interrupt.

Users must select the remote core type and index. These values must be set inversely in the other core configuration.

Users can configure the IPCF communication channels (UNMANAGED or MANAGED: pools with num_bufs and buf_size values).

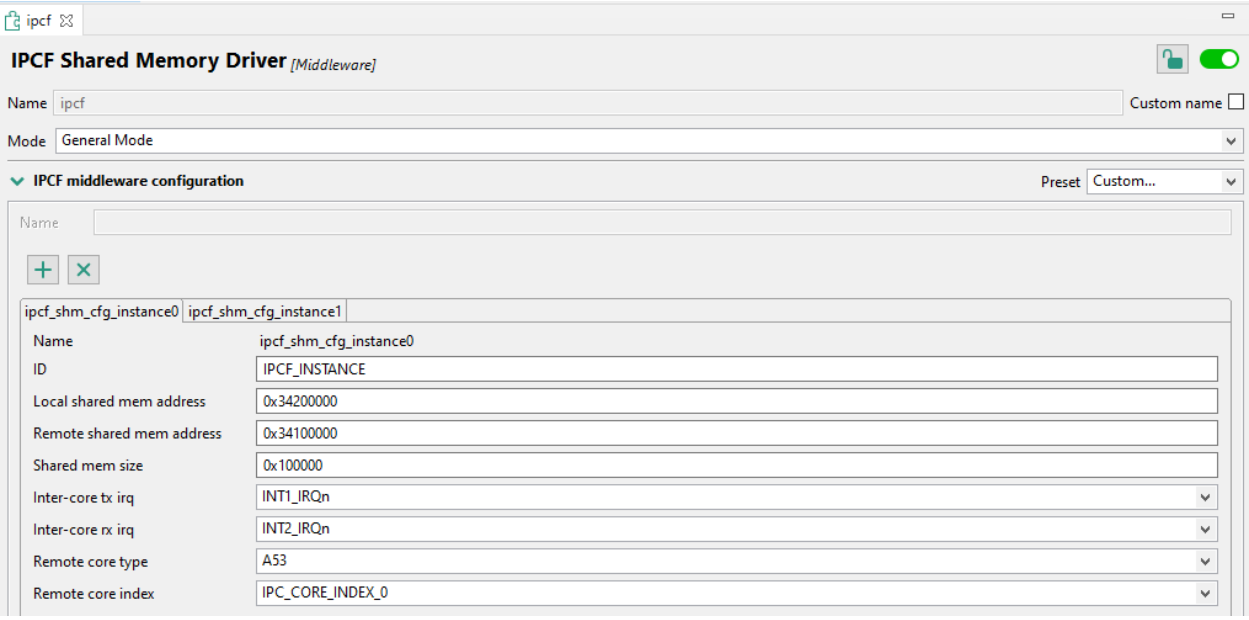
The maximum number of instances is defined by IPC_SHM_MAX_INSTANCES (maximum 255). The maximum number of shared memory channels is defined by IPC_SHM_MAX_CHANNELS (maximum 255). The maximum number of buffer pools that can be configured for a managed channel is defined by IPC_SHM_MAX_POOLS (maximum 255). The maximum number of buffers per pool is defined by IPC_SHM_MAX_BUFS_PER_POOL (maximum 65535).

Users can add a new IPCF instance to communicate with another core.

In Tresos the value configured for the “Inter Core Rx IRQ” parameter should be corelated with the value configured for the “Local Core” parameter. E.g.: if “Local Core” is selected as IPC_CORE_M33 then in “Inter Core Rx IRQ” an interrupt that is dedicated to the M33 core must be selected.

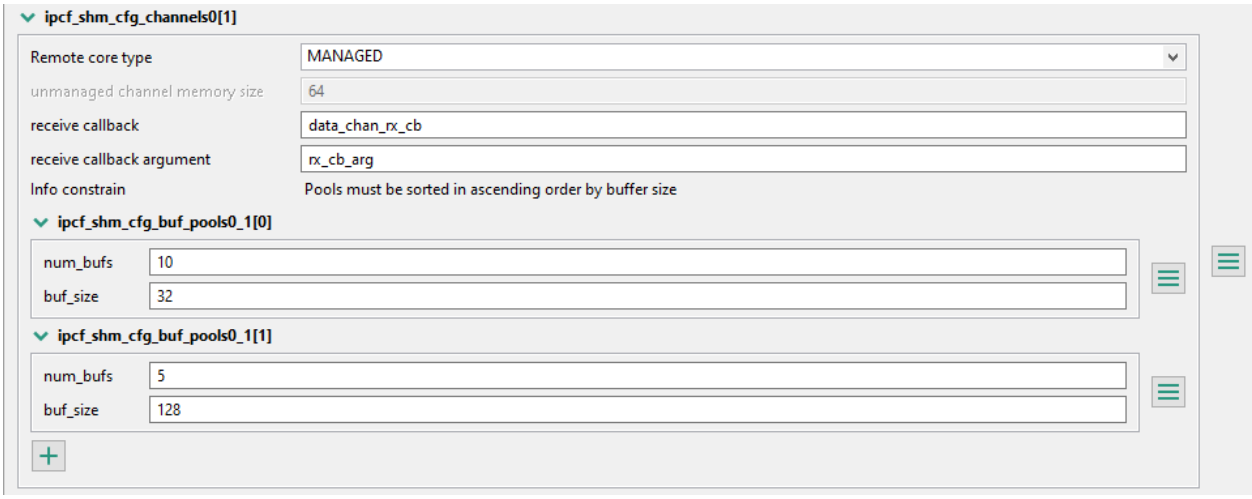
More information about parameters, values and structure types can be found in IPCF DRIVER API chapter. More details about the integration of the IPCF driver are available in the description of the sample application provided in the package.

6.1.1 S32DS Configuration

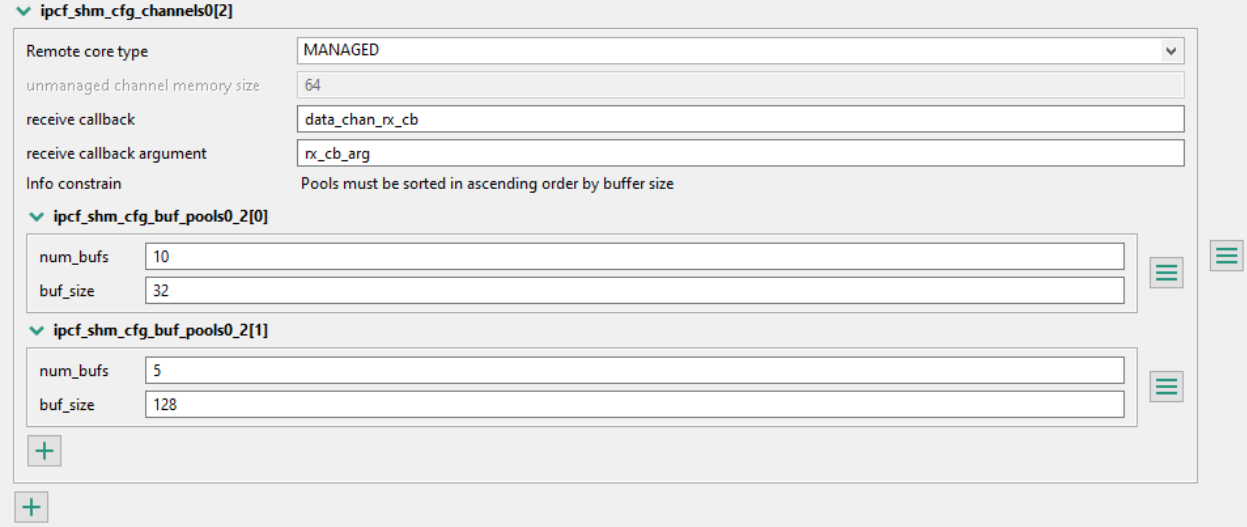


IPCF DS32 configuration example.

IPCF DS32 unmanaged channel configuration example.

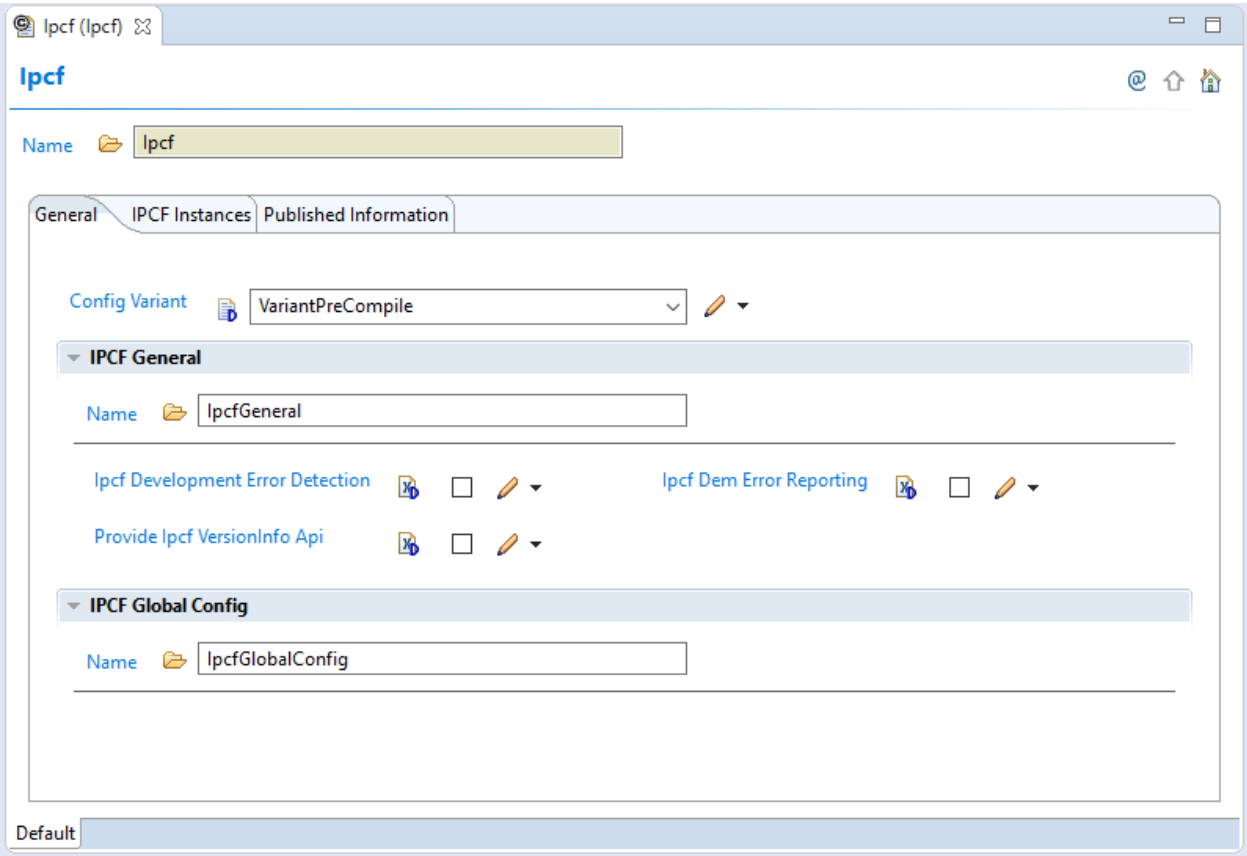


IPCF DS32 managed channel 1 configuration example.

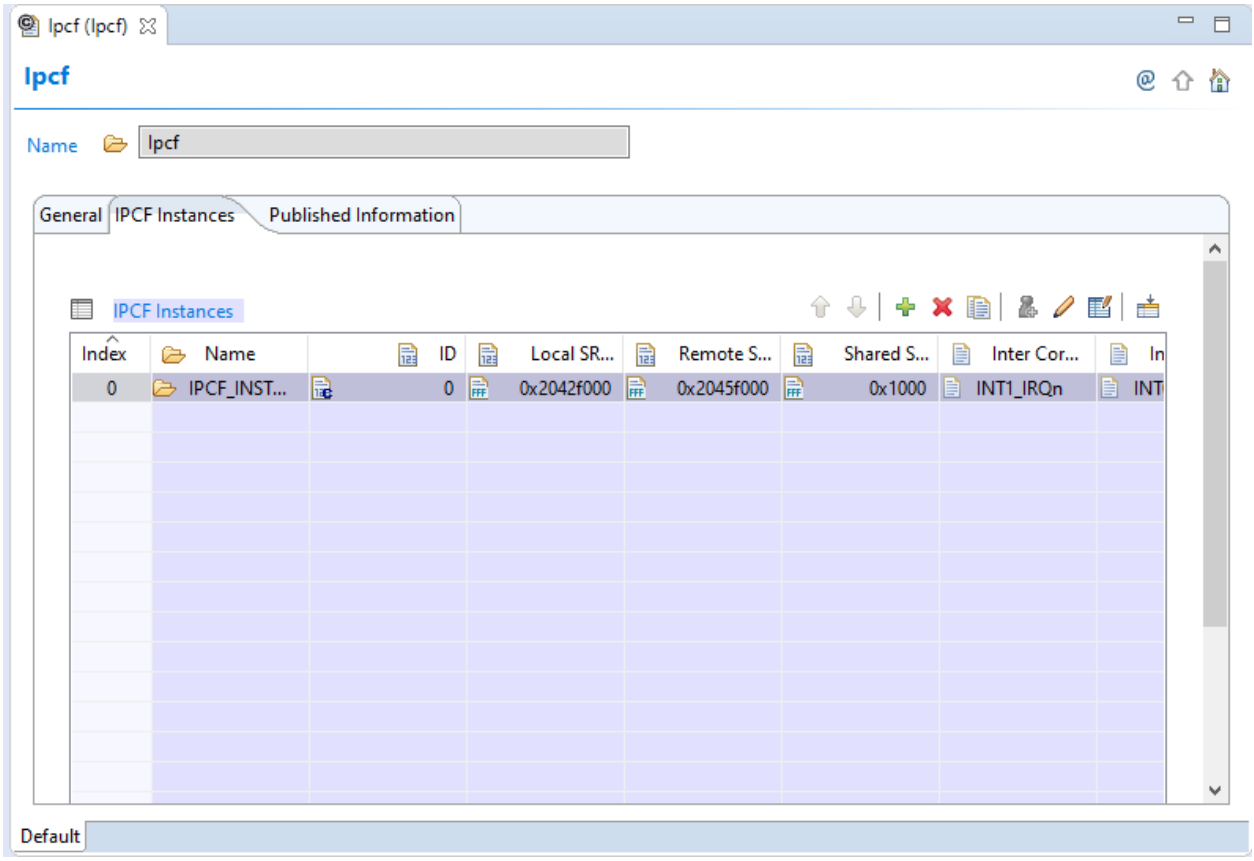


IPCF DS32 managed channel 2 configuration example.

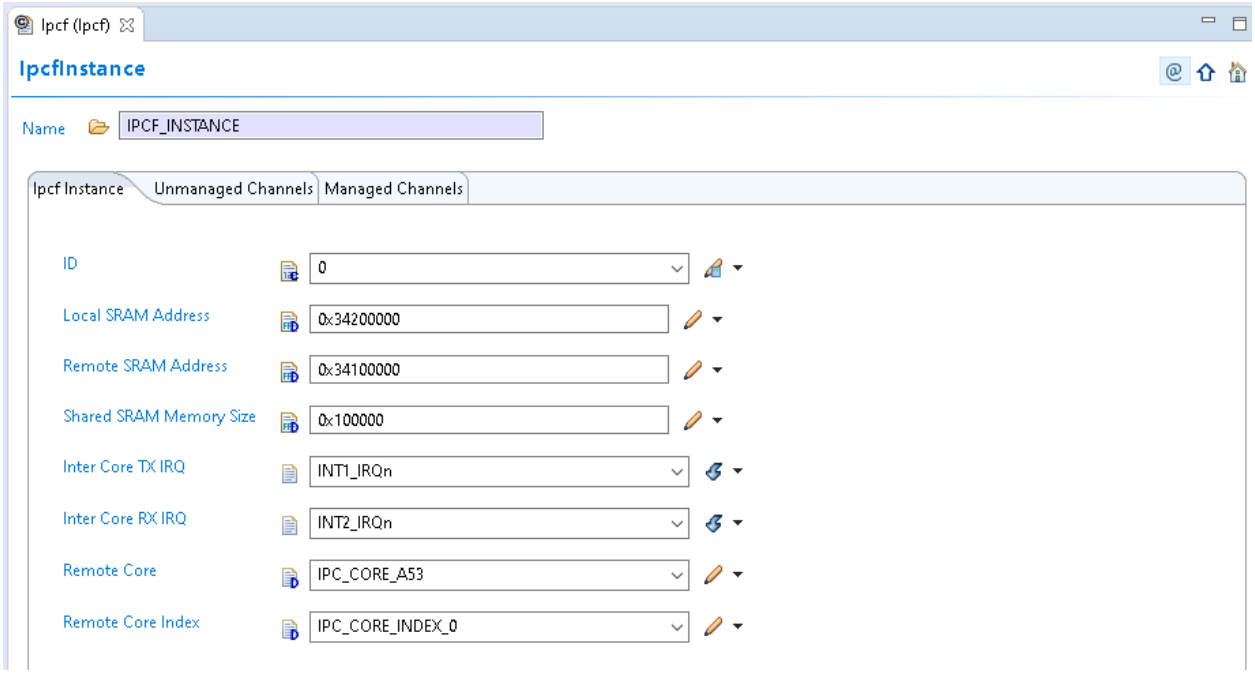
6.1.2 Tresos Configuration



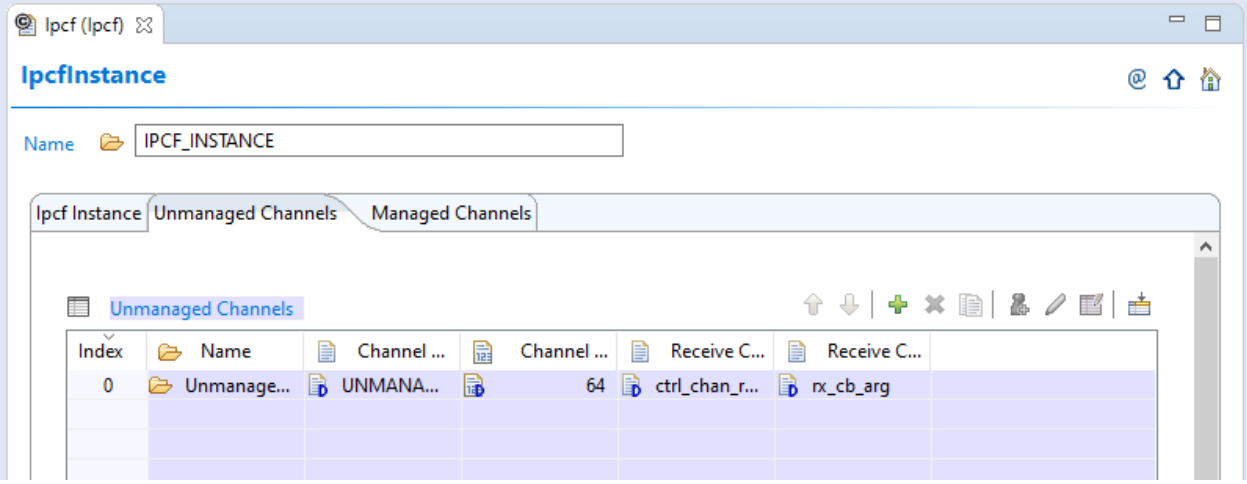
IPCF Tresos configuration example.



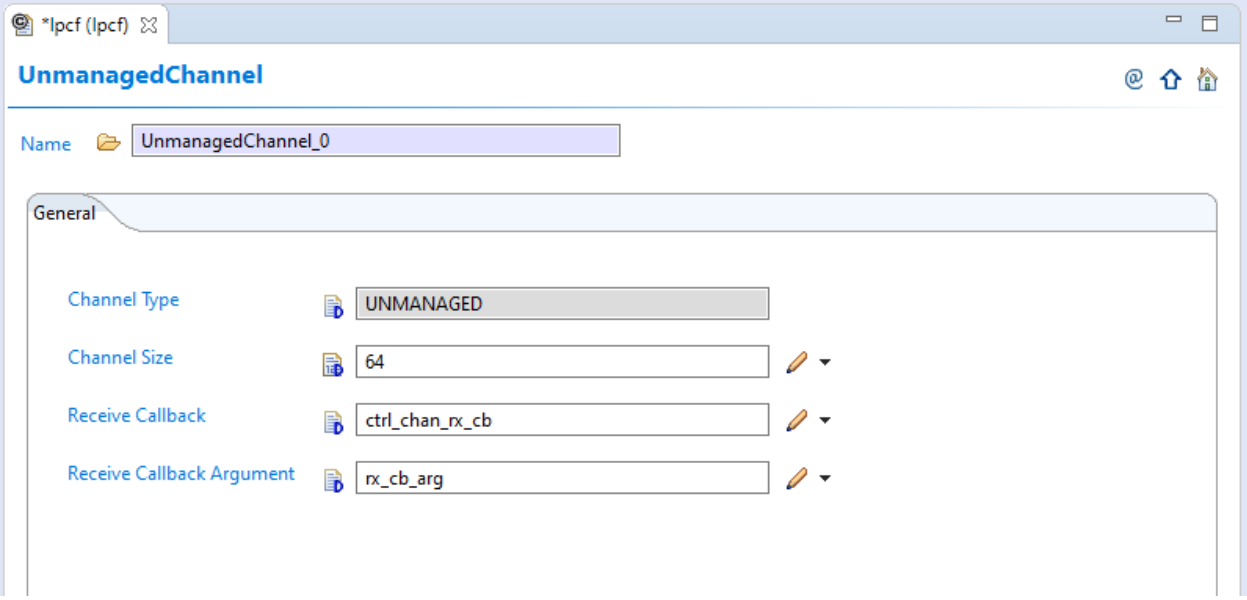
IPCF Tresos instance configuration example.



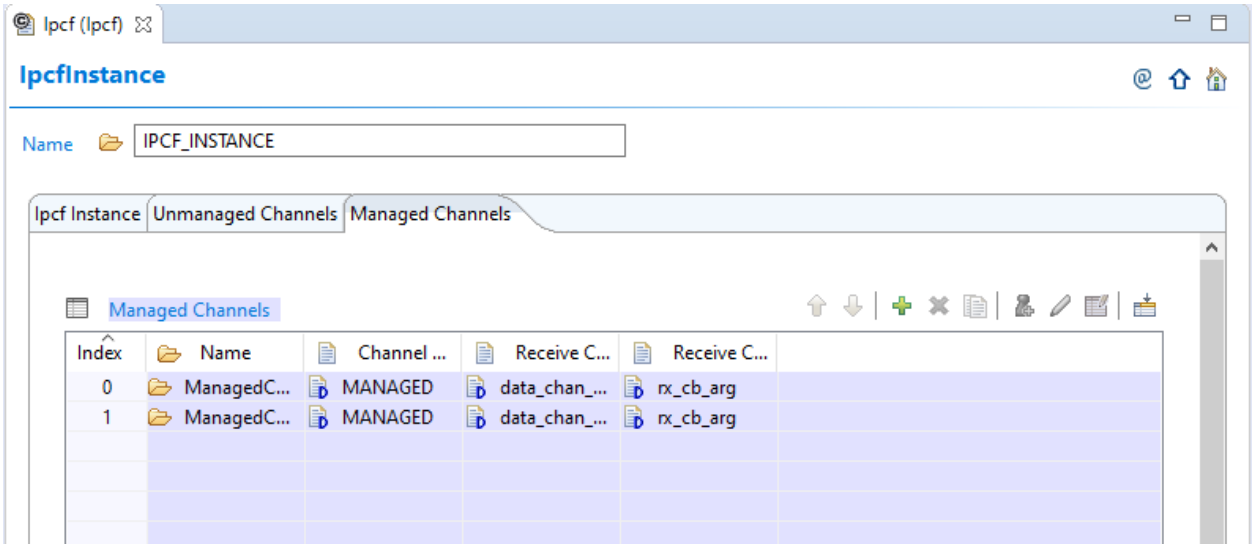
IPCF Tresos instance configuration example.



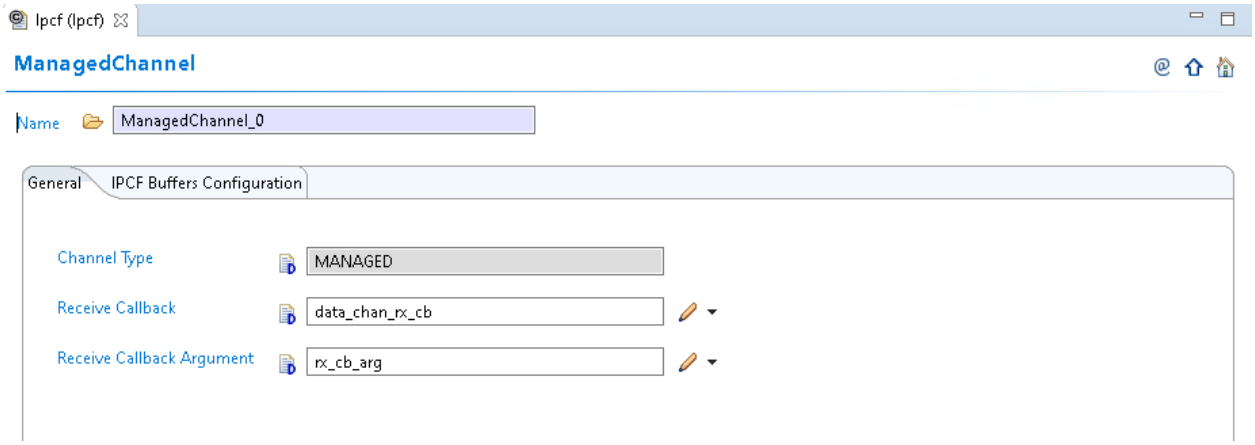
IPCF Tresos unmanaged channel configuration example.



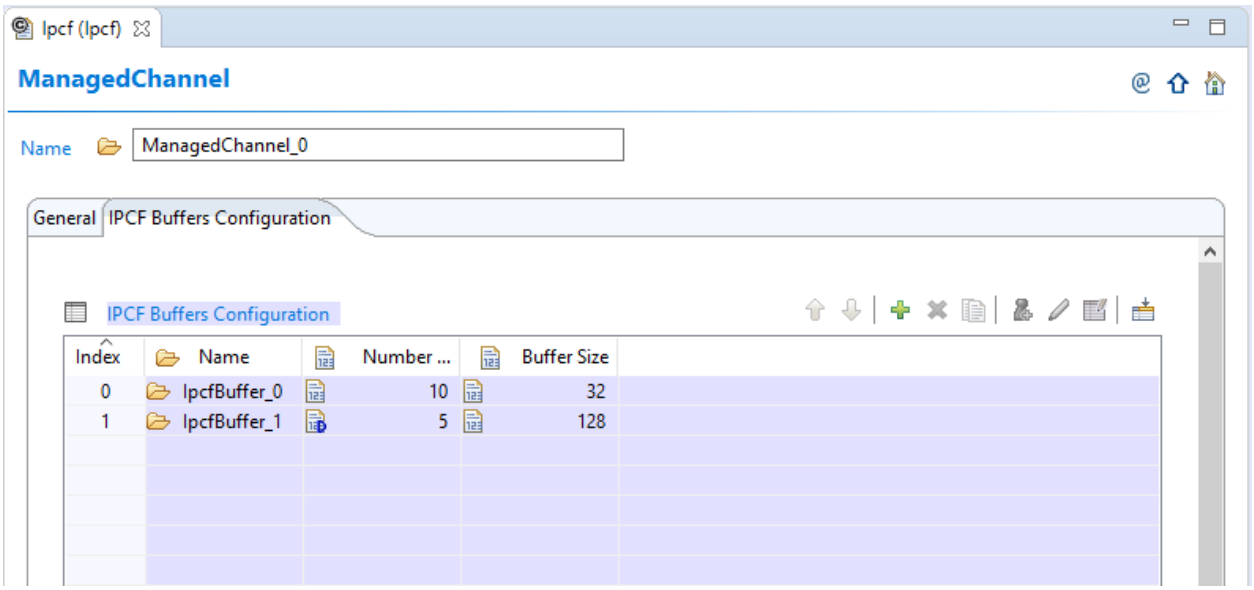
IPCF Tresos unmanaged channel configuration example.



IPCF Tresos managed channel configuration example.



IPCF Tresos managed channel 1 configuration example.



IPCF Tresos managed channel buffers configuration example.

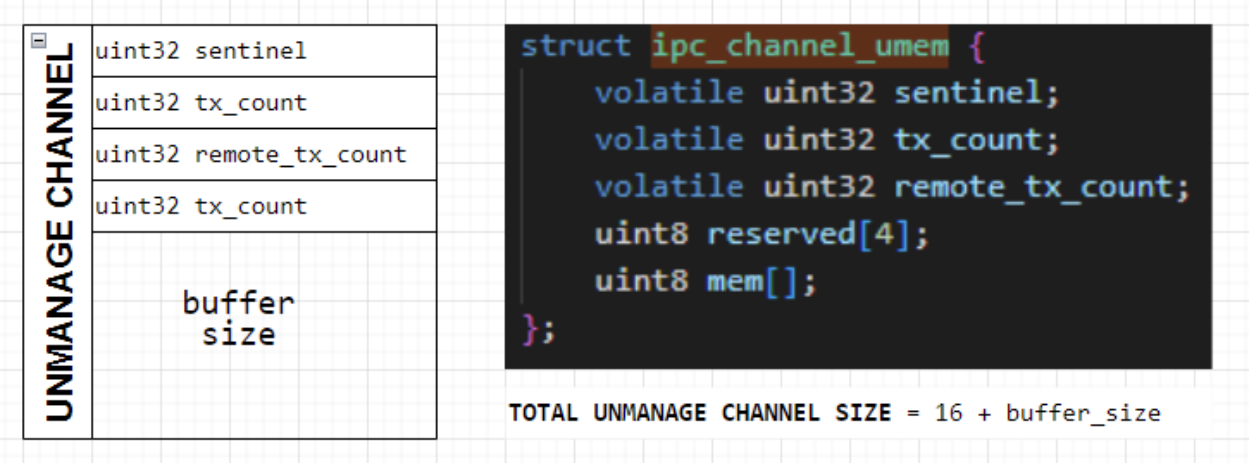
CHAPTER

SEVEN

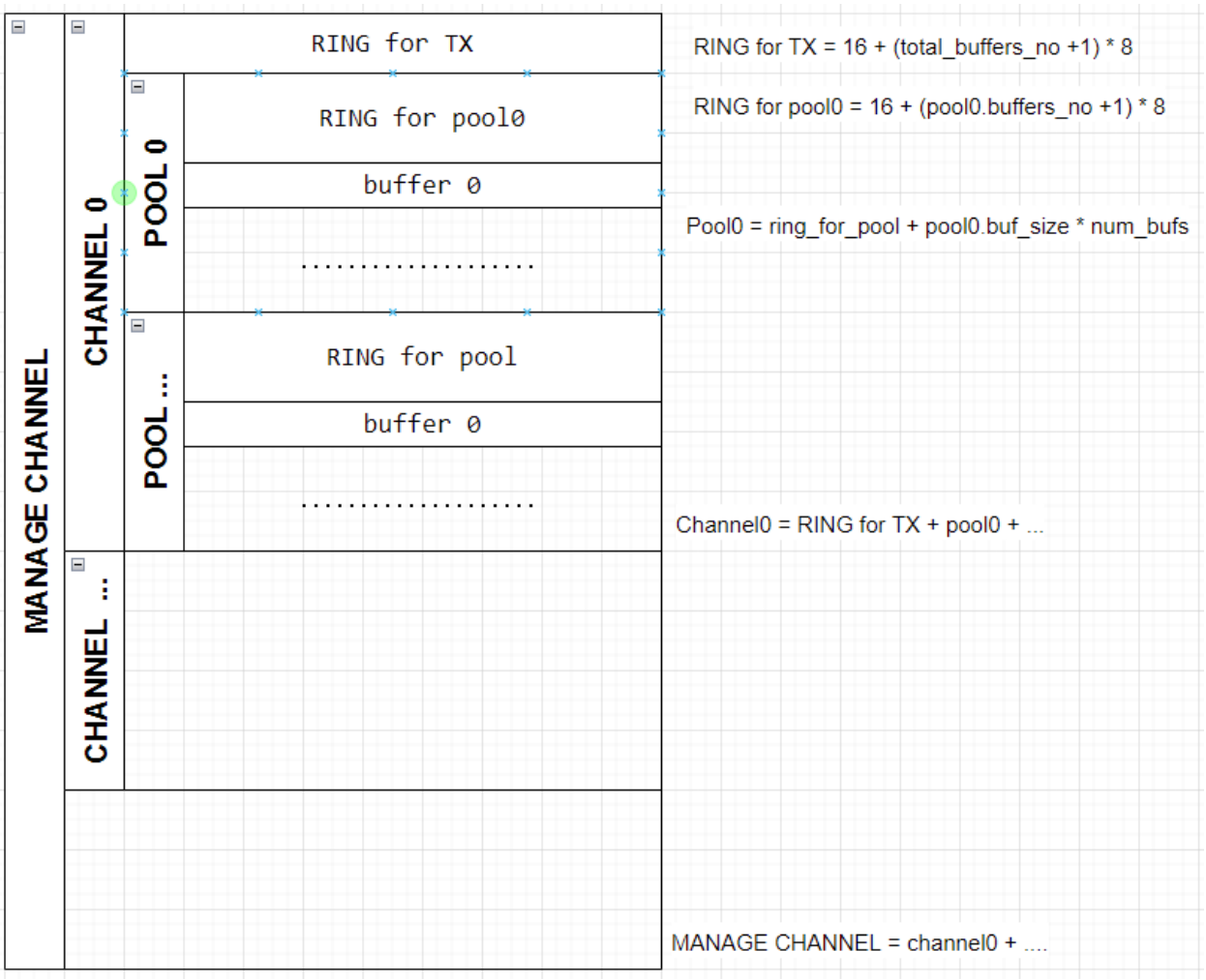
IPCF SHARED MEMORY LAYOUT

7.1 Memory layout and size

Users can calculate the shareable memory size using the following formulas:



IPCF unmanaged channel memory layout and size.



IPCF managed channel memory layout and size.

CHAPTER

EIGHT

IPCF SHARED MEMORY DRIVER API

8.1 IPCF Shared Memory Driver API

8.1.1 enum ipc_shm_channel_type

enum **ipc_shm_channel_type**

channel type

Definition

```
enum ipc_shm_channel_type {
    IPC_SHM_MANAGED,
    IPC_SHM_UNMANAGED
};
```

Constants

IPC_SHM_MANAGED
channel with buffer management enabled

IPC_SHM_UNMANAGED
buf mgmt disabled, app owns entire channel memory

Description

For unmanaged channels the application has full control over channel memory and no buffer management is done by ipc-shm device.

8.1.2 enum ipc_shm_queue_type

enum **ipc_shm_queue_type**

queue type

Definition

```
enum ipc_shm_queue_type {
    IPC_SHM_CHANNEL_QUEUE,
    IPC_SHM_POOL_QUEUE
};
```

Constants

IPC_SHM_CHANNEL_QUEUE
Channel queue

IPC_SHM_POOL_QUEUE
Pool buffer queue

8.1.3 enum ipc_shm_core_type

enum **ipc_shm_core_type**

core type

Definition

```
enum ipc_shm_core_type {
    IPC_CORE_DEFAULT,
    IPC_CORE_A53,
    IPC_CORE_M7,
    IPC_CORE_M4,
    IPC_CORE_Z7,
    IPC_CORE_Z4,
    IPC_CORE_Z2,
    IPC_CORE_R52,
    IPC_CORE_M33,
    IPC_CORE_BBE32
};
```

Constants

IPC_CORE_DEFAULT
used for letting driver auto-select remote core type

IPC_CORE_A53
ARM Cortex-A53 core

IPC_CORE_M7
ARM Cortex-M7 core

IPC_CORE_M4
ARM Cortex-M4 core

IPC_CORE_Z7
PowerPC e200z7 core

IPC_CORE_Z4
PowerPC e200z4 core

IPC_CORE_Z2
PowerPC e200z2 core

IPC_CORE_R52
ARM Cortex-R52 core

IPC_CORE_M33
ARM Cortex-M33 core

IPC_CORE_BBE32
Tensilica ConnX BBE32EP core

8.1.4 enum ipc_shm_core_index

enum **ipc_shm_core_index**
core index

Definition

```
enum ipc_shm_core_index {
    IPC_CORE_INDEX_0,
    IPC_CORE_INDEX_1,
    IPC_CORE_INDEX_2,
    IPC_CORE_INDEX_3,
    IPC_CORE_INDEX_4,
    IPC_CORE_INDEX_5,
    IPC_CORE_INDEX_6,
    IPC_CORE_INDEX_7
};
```

Constants

IPC_CORE_INDEX_0
Processor index 0

IPC_CORE_INDEX_1
Processor index 1

IPC_CORE_INDEX_2
Processor index 2

IPC_CORE_INDEX_3
Processor index 3

IPC_CORE_INDEX_4
Processor index 4

IPC_CORE_INDEX_5
Processor index 5

IPC_CORE_INDEX_6
Processor index 6

IPC_CORE_INDEX_7
Processor index 7

8.1.5 struct ipc_shm_pool_cfg

struct **ipc_shm_pool_cfg**
memory buffer pool parameters

Definition

```
struct ipc_shm_pool_cfg {
    uint16 num_bufs;
    uint32 buf_size;
}
```

Members

num_bufs
number of buffers

buf_size
buffer size

8.1.6 struct ipc_shm_managed_cfg

struct **ipc_shm_managed_cfg**

managed channel parameters

Definition

```
struct ipc_shm_managed_cfg {
    uint8 num_pools;
    struct ipc_shm_pool_cfg *pools;
    void (*rx_cb)(void *cb_arg, const uint8 instance, uint8 chan_id, void *buf, uint32␣
↪size);
    void *cb_arg;
}
```

Members

num_pools
number of buffer pools

pools
memory buffer pools parameters

rx_cb
receive callback

cb_arg
optional receive callback argument

8.1.7 struct ipc_shm_unmanaged_cfg

struct **ipc_shm_unmanaged_cfg**

unmanaged channel parameters

Definition

```
struct ipc_shm_unmanaged_cfg {
    uint32 size;
    void (*rx_cb)(void *cb_arg, const uint8 instance, uint8 chan_id, void *mem);
    void *cb_arg;
}
```

Members

size
unmanaged channel memory size

rx_cb
receive callback

cb_arg
optional receive callback argument

8.1.8 struct ipc_shm_channel_cfg

struct **ipc_shm_channel_cfg**
channel parameters

Definition

```
struct ipc_shm_channel_cfg {
    enum ipc_shm_channel_type type;
    union {
        struct ipc_shm_managed_cfg managed;
        struct ipc_shm_unmanaged_cfg unmanaged;
    } ch;
}
```

Members

type
channel type from *enum ipc_shm_channel_type*

ch
undescribed

ch.managed
managed channel parameters

ch.unmanaged
unmanaged channel parameters

8.1.9 struct ipc_shm_remote_core

struct **ipc_shm_remote_core**
remote core type and index

Definition

```
struct ipc_shm_remote_core {
    enum ipc_shm_core_type type;
    enum ipc_shm_core_index index;
}
```

Members

type
core type from *enum ipc_shm_core_type*

index
core number

Description

Core type can be IPC_CORE_DEFAULT, in which case core index doesn't matter because it's chosen automatically by the driver.

8.1.10 struct ipc_shm_local_core

struct **ipc_shm_local_core**
local core type, index and trusted cores

Definition

```
struct ipc_shm_local_core {
    enum ipc_shm_core_type type;
    enum ipc_shm_core_index index;
    uint32 trusted;
}
```

Members

type
core type from *enum ipc_shm_core_type*

index
core number targeted by remote core interrupt

trusted
trusted cores mask

Description

Core type can be IPC_CORE_DEFAULT, in which case core index doesn't matter because it's chosen automatically by the driver.

Trusted cores mask specifies which cores (of the same core type) have access to the inter-core interrupt status register of the targeted core. The mask can be formed from *enum ipc_shm_core_index*.

8.1.11 struct ipc_shm_cfg

struct **ipc_shm_cfg**

IPC shm parameters

Definition

```
struct ipc_shm_cfg {
    uintptr local_shm_addr;
    uintptr remote_shm_addr;
    uint32 shm_size;
    sint16 inter_core_tx_irq;
    sint16 inter_core_rx_irq;
    uint8 mru_tx_channel_id;
    uint8 mru_rx_channel_id;
    struct ipc_shm_local_core local_core;
    struct ipc_shm_remote_core remote_core;
    uint8 num_channels;
    struct ipc_shm_channel_cfg *channels;
#ifdef USING_OS_AUTOSAROS
    ISRType isr_id_handler;
#endif
}
```

Members

local_shm_addr
local shared memory physical address

remote_shm_addr
remote shared memory physical address

shm_size
local/remote shared memory size

inter_core_tx_irq
inter-core interrupt reserved for shm driver Tx

inter_core_rx_irq
inter-core interrupt reserved for shm driver Rx

mru_tx_channel_id
mru channel index for shm driver Tx

mru_rx_channel_id
mru channel index for shm driver Rx

local_core
local core targeted by remote core interrupt

remote_core
remote core to trigger the interrupt on

num_channels
number of shared memory channels

channels
IPC channels parameters array

isr_id_handler
the name of OsIsr defined to handle the interrupt (only if using AutosarOS)

Description

The TX and RX interrupts used must be different. For ARM platforms, a default value can be assigned to the local and remote core using IPC_CORE_DEFAULT. Local core is only used for platforms on which Linux may be running on multiple cores, and is ignored for RTOS and baremetal implementations.

Local and remote channel and buffer pool configurations must be symmetric.

8.1.12 struct ipc_shm_instances_cfg

struct **ipc_shm_instances_cfg**
IPC shm parameters

Definition

```
struct ipc_shm_instances_cfg {
    uint8 num_instances;
    struct ipc_shm_cfg *shm_cfg;
}
```

Members

num_instances
number of shared memory instances

shm_cfg
IPC shm parameters array

8.1.13 ipc_shm_init_instance

sint8 **ipc_shm_init_instance**(uint8 instance, const struct *ipc_shm_cfg* *cfg)

Initialize the specified instance of the IPC-Shm driver

Parameters

- **instance** (uint8) – instance id
- **cfg** (const struct *ipc_shm_cfg**) – ipc-shm instance configuration

Description

Return IPC_SHM_E_OK on success, error code otherwise

8.1.14 ipc_shm_init

sint8 **ipc_shm_init**(const struct *ipc_shm_instances_cfg* *cfg)

initialize shared memory device

Parameters

- **cfg** (const struct *ipc_shm_instances_cfg**) – configuration parameters

Description

Function is non-reentrant.

Return

0 on success, error code otherwise

8.1.15 ipc_shm_free_instance

void **ipc_shm_free_instance**(const uint8 instance)

Deinitialize the specified instance of the IPC-Shm driver

Parameters

- **instance** (const uint8) – instance id

Description

Function is non-reentrant.

8.1.16 ipc_shm_free

void **ipc_shm_free**(void)

release all instances of shared memory device

Parameters

- **void** – no arguments

Description

Function is non-reentrant.

8.1.17 ipc_shm_acquire_buf

void *__ipc_shm_acquire_buf__(const uint8 instance, uint8 chan_id, uint32 mem_size)

request a buffer for the given channel

Parameters

- **instance** (`const uint8`) – instance id
- **chan_id** (`uint8`) – channel index
- **mem_size** (`uint32`) – required size

Description

Function used only for managed channels where buffer management is enabled. Function is thread-safe for different channels but not for the same channel.

Return

pointer to the buffer base address or NULL if buffer not found

8.1.18 ipc_shm_release_buf

sint8 **ipc_shm_release_buf**(const uint8 instance, uint8 chan_id, const void *buf)

release a buffer for the given channel

Parameters

- **instance** (`const uint8`) – instance id
- **chan_id** (`uint8`) – channel index
- **buf** (`const void*`) – buffer pointer

Description

Function used only for managed channels where buffer management is enabled. Function is thread-safe for different channels but not for the same channel.

Return

0 on success, error code otherwise

8.1.19 ipc_shm_tx

sint8 **ipc_shm_tx**(const uint8 instance, uint8 chan_id, void *buf, uint32 size)

send data on given channel and notify remote

Parameters

- **instance** (`const uint8`) – instance id
- **chan_id** (`uint8`) – channel index
- **buf** (`void*`) – buffer pointer
- **size** (`uint32`) – size of data written in buffer

Description

Function used only for managed channels where buffer management is enabled. Function is thread-safe for different channels but not for the same channel.

Return

0 on success, error code otherwise

8.1.20 ipc_shm_unmanaged_acquire

void ***ipc_shm_unmanaged_acquire**(const uint8 instance, uint8 chan_id)

acquire the unmanaged channel local memory

Parameters

- **instance** (`const uint8`) – instance id
- **chan_id** (`uint8`) – channel index

Description

Function used only for unmanaged channels. The memory must be acquired only once after the channel is initialized. There is no release function needed. Function is thread-safe for different channels but not for the same channel.

Return

pointer to the channel memory or NULL if invalid channel

8.1.21 ipc_shm_unmanaged_tx

sint8 **ipc_shm_unmanaged_tx**(const uint8 instance, uint8 chan_id)

notify remote that data has been written in channel

Parameters

- **instance** (`const uint8`) – instance id
- **chan_id** (`uint8`) – channel index

Description

Function used only for unmanaged channels. It can be used after the channel memory has been acquired whenever is needed to signal remote that new data is available in channel memory. Function is thread-safe for different channels but not for the same channel.

Return

0 on success, error code otherwise

8.1.22 ipc_shm_is_remote_ready

sint8 **ipc_shm_is_remote_ready**(const uint8 instance)

check whether remote is initialized

Parameters

- **instance** (`const uint8`) – instance id

Description

Function used to check if the remote is initialized and ready to receive messages. It should be invoked at least before the first transmit operation. Function is thread-safe.

Return

0 if remote is initialized, error code otherwise

8.1.23 ipc_shm_poll_channels

sint8 **ipc_shm_poll_channels**(const uint8 instance)

poll the channels for available messages to process

Parameters

- **instance** (`const uint8`) – instance id

Description

This function handles all channels using a fair handling algorithm: all channels are treated equally and no channel is starving. Function is thread-safe for different instances but not for same instance.

Return

number of messages processed, error code otherwise

CHAPTER

NINE

REVISION HISTORY

Table 1: Revision

Rev	Date	Author	Description
1.00	20-05-2024	IPCF team	update for current release

CHAPTER

TEN

SUPPORT

For technical support please visit:

- https://www.nxp.com/support

To contact NXP please visit:

- https://www.nxp.com/about/about-nxp/about-nxp/contact-us:CONTACTUS