AN11177

Inter Processor Communication on LPC43xx

Rev. 2 — 20 August 2014 Application note

Document information

Info	Content
Keywords	LPC4350FET256, LPC4350FET180, LPC4350FBD208, LPC4330FET256, LPC4330FET180, LPC4330FET100, LPC4330FBD144, LPC4320FET100, LPC4320FBD144, LPC4320FBD100, LPC4310FET100, LPC4310FBD144, LPC4300, dual core, IPC
Abstract	Inter-processor communication on dual core microcontrollers

Revision history

Rev	Date	Description
2	20140820	Bit masks updated in Fig 7 and Fig 8.
1	20120319	Initial version.

Contact information

For more information, please visit: http://www.nxp.com

For sales office addresses, please send an email to: salesaddresses@nxp.com

1. Introduction

This document provides information about:

- The API implementation for dual core communication on LPC43xx targets
- How to include or exclude functionality by means of a platform-wise configuration file
- System level settings and debug options to be aware of when changing the configuration

Some application examples show the API usage, with and without RTOS (FreeRTOS) support.

For the LPC4300 device, the M4 CPU will be referred to as “master” and the M0 CPU will be referred to as “slave” throughout this document.

The latest code for IPC is available on at http://www.lpcware.com/.

The application note software is written in Keil IDE. The following FAQs help to run the multi-core examples in LPCXpresso IDE.
http://www.lpcware.com/content/faq/how-run-multicore-examples-provided-lpcopen-lpc43xx-packages

http://www.lpcware.com/content/faq/lpcxpresso/lpc43xx-multicore-apps

2. Application Programming Interface

Three alternative implementations for IPC communication are provided: an interrupt based mechanism, a “message queue” based mechanism and a “mailbox” based mechanism.

The interrupt mechanism is the simplest, and can be used to send a notification (or signal) to the other core, without associated data.

The message queue mechanism follows the approach and guidelines mentioned in the device user manual. The mailbox approach is an alternative implementation.

The pros and cons of each are detailed in the next sections.

All implementations include common APIs which allow a master processor to download a slave processor application image, start, and halt the slave processor.

In the message queue or mailbox implementations, it is assumed that the “communication channels” (one queue, or one specific mailbox) are not shared between multiple tasks or functions.

For the message queue interface, this implies there is one entry point (or “gatekeeper” task) within the sending application where messages are inserted into the queue. Similarly, there is one entry point (or “gatekeeper” task) within the receiving application where messages are retrieved from the queue.

For the mailbox interface, this implies there is one entry point (or “gatekeeper” task) within the sending application where a message can be sent to one specific mailbox. Similarly, there is one entry point (or “gatekeeper” task) within the receiving application where a message gets processed from one specific mailbox.

The actual implementation was not designed for access sharing on one queue (or one specific mailbox) between multiple tasks or functions.

2.1 Interrupt

This implementation simply allows one core to send an interrupt to the other core, to be used as a notification of some sort of application specific defined event.

The interrupt routine associated with the notification interrupt is very compact.

The user has to specify a callback function, which is executed in the interrupt routine context when the interrupt is serviced, and can be used to perform some quick operation. If the callback is not used, it can be left implemented as an empty function.

For signaling to the “remote” core, the “local” processor issues the dedicated instruction SEV (send event) provided by the Cortex architecture.

Within the interrupt routine, a flag variable is also set, indicating an IPC notification has been received. This flag variable can be used by the application running on the receiving core to check for the status.

2.1.1 Implementation details

The module is composed of the following files:

- api\interrupt\inc
  - ipc_int.h
- api\interrupt\src
  - ipc_int.c

Clearing the interrupt flag is treated as a critical section, so interrupts are briefly disabled within the function used for quitting the flag status.

The priority of the interrupt routines on the master and the slave is configurable at build time.

On the LPC4300 implementation, on the master side all interrupts with priority equal or lower than the HOST_IPC_PRIORITY value will be masked by programming the BASEPRI register accordingly. On the slave side, interrupts are briefly disabled globally since the CPU hardware does not support such a selective masking.

2.1.2 Interrupt queue API set

The following APIs are provided:

Table 1. Interrupt APIs

Function name	Module	Return type	Parameters
IPC_masterInitInterrupt	ipc_int.c	void	intCallback_t masterCback
IPC_slaveInitInterrupt	ipc_int.c	void	intCallback_t slaveCback
IPC_sendInterrupt	ipc_int.c	void	Void
IPC_resetIntFlag	ipc_int.c	void	Void

2.1.3 IPC_masterInitInterrupt

Initializes the IPC communication by configuring and enabling the IPC interrupt on the master side, clears the interrupt pending flag, sets the master interrupt callback to the passed masterCback parameter (the callback function pointer).

2.1.4 IPC_slaveInitInterrupt

Initializes the IPC communication by configuring and enabling the IPC interrupt on the slave side, clears the interrupt pending flag, sets the slave interrupt callback to the slaveCback parameter (the callback function pointer).

2.1.5 IPC_sendInterrupt

Sends an interrupt signal to the remote CPU core.

In response, the remote core is interrupted, and executes the interrupt service routine associated with the IPC interrupt.

Within the interrupt routine, the local callback function is executed, and the local interrupt flag variable is set to MSG_PENDING

2.1.6 IPC_resetIntFlag

Clears the interrupt flag variable to the value NO_MSG

2.1.7 Interrupt API example usage flowcharts

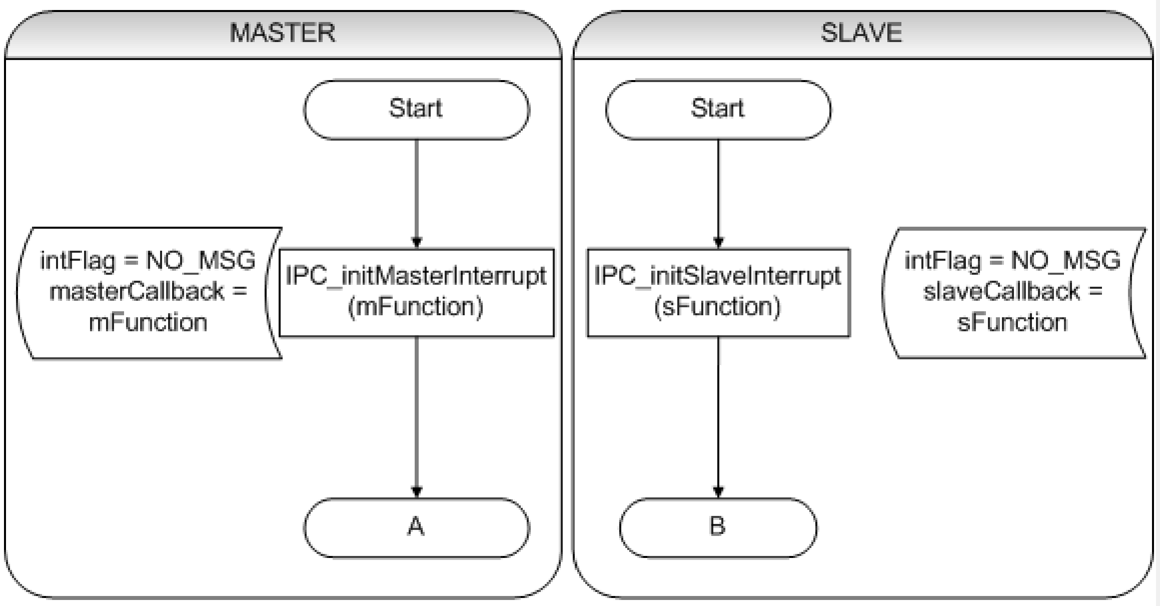


Fig 1. Interrupt API setup

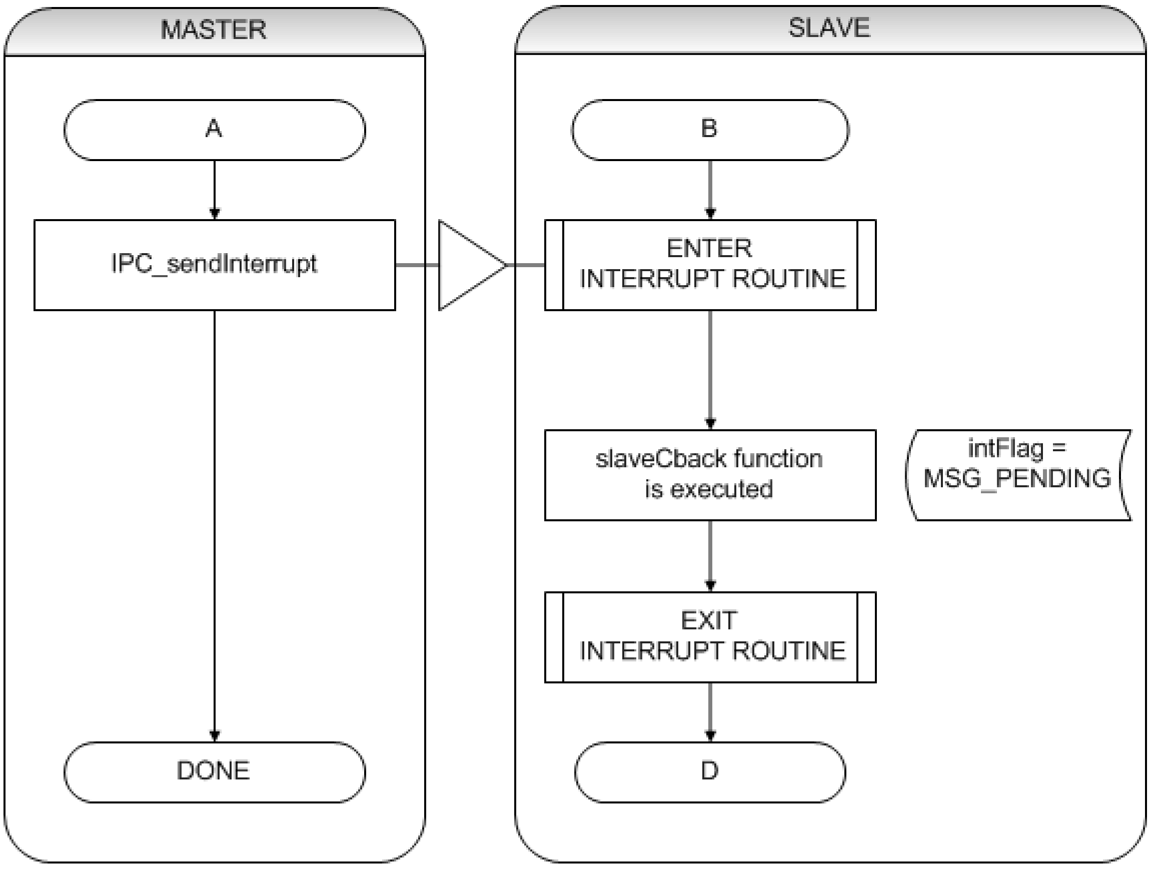


Fig 2. Master to Slave interrupt

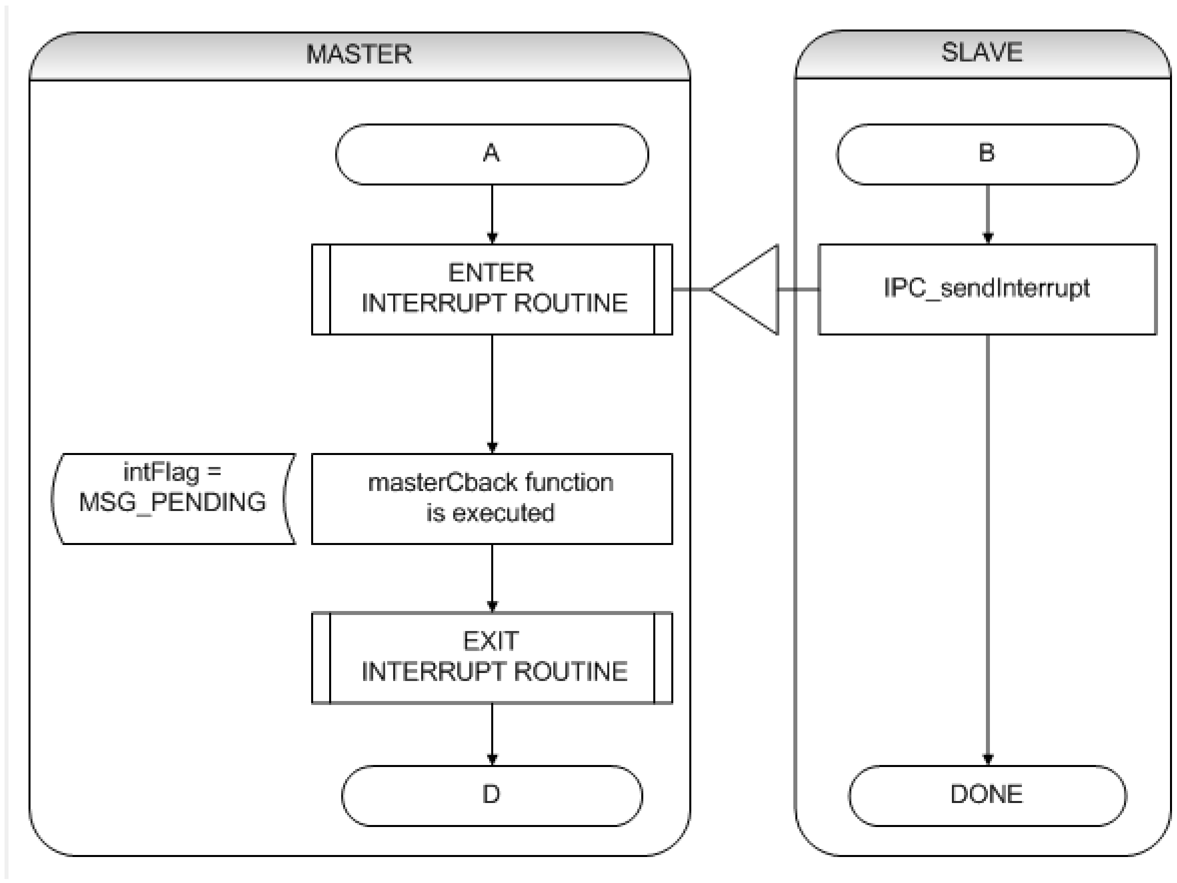


Fig 3. Slave to Master interrupt

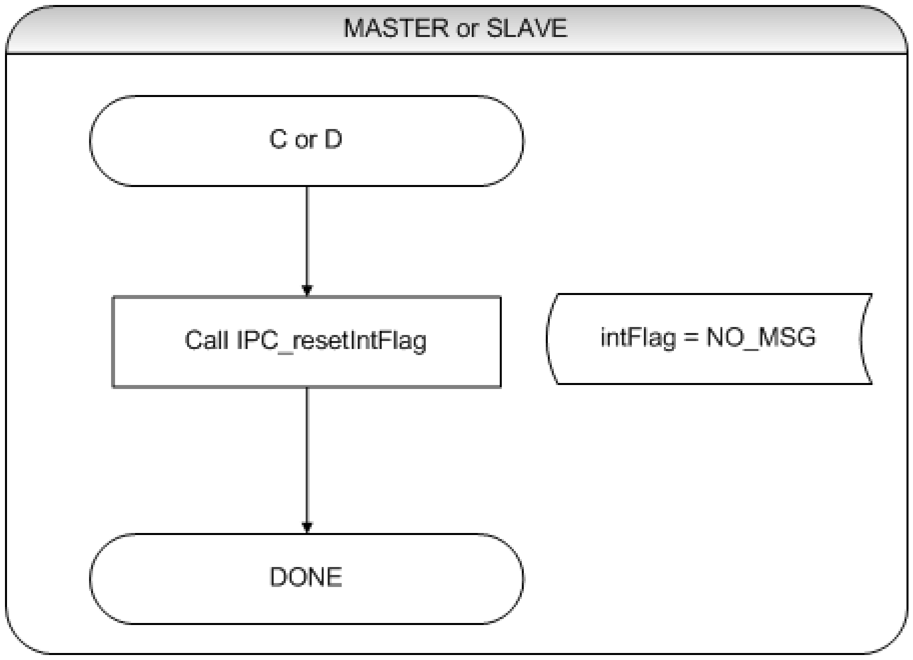


Fig 4. Quitting the interrupt flag

2.2 Message queue

This implementation follows the approach detailed in the device user manual, which should be referred to for the complete specification.

There are two areas of shared memory being defined, which are used to store the messages that each processor wants to send to the other.

There is one buffer (HOST COMMAND BUFFER) dedicated to the commands being sent from the master processor to the slave processor, and one other (separate) buffer (HOST MESSAGE BUFFER) dedicated to the messages which the slave processor sends back to the master processor.

Fig 5 shows a representation of the concept:

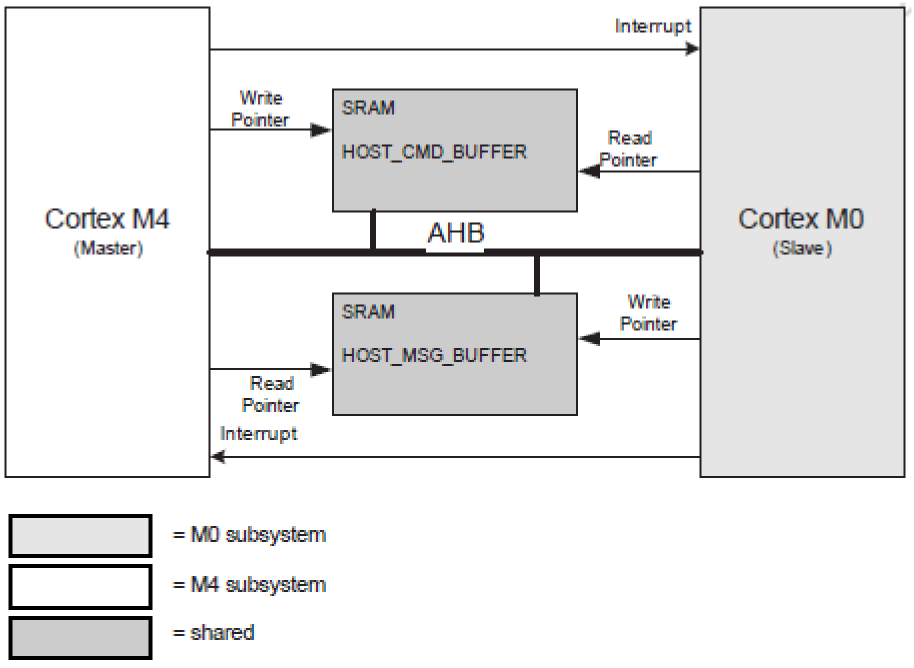


Fig 5. Message Queue based IPC

Only the master processor is allowed to write commands to the Command Buffer, and will receive messages by reading out of the Message Buffer.

Only the slave processor is allowed to write messages to the Message Buffer, and receives commands by reading out of the Command Buffer.

Once a processor writes new messages within the buffer, it notifies the other processor that there is data (commands or messages) available to process.

For signaling to the “remote” core, an interrupt mechanism is used. For signaling to the “remote” core, the “local” processor issues the dedicated instruction SEV (send event) provided by the Cortex architecture.

The interrupt routine associated with the notification interrupt is very compact, and sets just a flag variable. The application checks the flag content at its convenience in order to determine if there is new data to process.

2.2.1 Implementation details

The module is composed of the following files:

- api\queue\inc
  - ipc_queue.h
  - ipc_bufdef.h
- api\queue\src
  - ipc_cmd_buffer.c
  - ipc_msg_buffer.c
  - ipc_queue.c

The logical messages for each queue are defined within the master_msg.h and slave_ipc_msg.h files. These are part of the platform configuration files.

One “IPC block” (defined in ipc_msg_buffer.c and ipc_cmd_buffer.c), is composed of the following elements:

- The start address of the message queue buffer
- The end address of the message queue buffer
- One read pointer which points to the next message to be read
- One write pointer which points to the next free location
- A memory buffer which is used to hold the messages

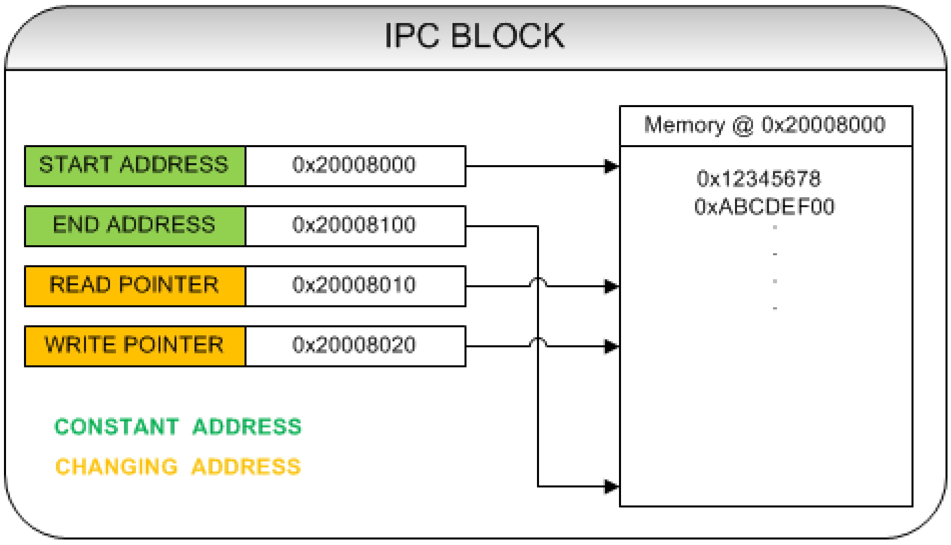


Fig 6. IPC Block example for message queue

Modification of the write pointer is restricted to the processor which is writing messages to the buffer. The write pointer always points to the location within the buffer where the next message can be written (next free location).

Modification of the read pointer is restricted to the processor which is reading messages to the buffer. The read pointer always points to the location within the buffer where the next message can be read. The read pointer is always “following” the write pointer.

Every time a processor writes messages to the queue, care is taken that the resulting write pointer value never gets equal or greater than the read pointer.

When the read pointer is equal to the write pointer the queue is considered empty, which means all messages have been read out by the remote processor, and no new messages have been inserted.

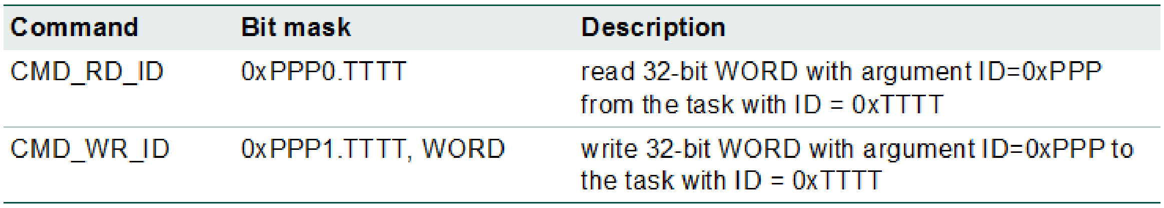
Since there is a dedicated function to notify the remote processor of the availability of new messages, it is possible to write a burst of several messages into the queue, instead of triggering one interrupt for every message. This can reduce the overhead on the remote processor by reducing the rate at which interrupts are issued.

The setting and clearing of the “message pending” or “command pending” is treated as a critical section, so interrupts get disabled for a short period of time when changing the flag status. The priority of the interrupt routines is configurable at build time.

On the LPC4300 implementation, on the master side all interrupts with priority equal or lower than the HOST_IPC_PRIORITY value will be masked by programming the BASEPRI register accordingly. On the slave side interrupts will be disabled globally for a short period of time since the CPU hardware does not support such a selective masking.

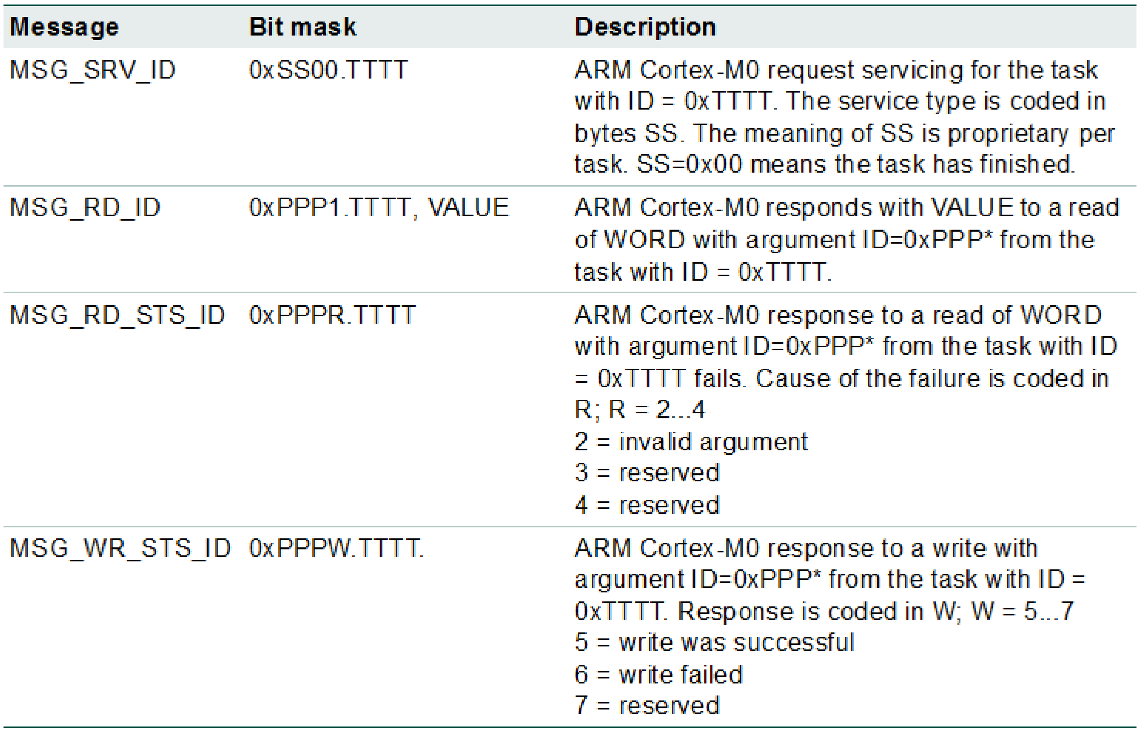
2.2.2 Command and message types

The following tables list the implemented command and message types which are included in the message queue API.



Command	Bit mask	Description
CMD_RD_ID	0xPPP0.TTTT	read 32-bit WORD with argument ID=0xPPP from the task with ID = 0xTTTT
CMD_WR_ID	0xPPP1.TTTT, WORD	write 32-bit WORD with argument ID=0xPPP to the task with ID = 0xTTTT

Fig 7. Command types



Message	Bit mask	Description
MSG_SRV_ID	0xSS00.TTTT	ARM Cortex-M0 request servicing for the task with ID = 0xTTTT. The service type is coded in bytes SS. The meaning of SS is proprietary per task. SS=0x00 means the task has finished.
MSG_RD_ID	0xPPP1.TTTT, VALUE	ARM Cortex-M0 responds with VALUE to a read of WORD with argument ID=0xPPP* from the task with ID = 0xTTTT.
MSG_RD_STS_ID	0xPPPR.TTTT	ARM Cortex-M0 response to a read of WORD with argument ID=0xPPP* from the task with ID = 0xTTTT fails. Cause of the failure is coded in R; R = 2...4 2 = invalid argument 3 = reserved 4 = reserved
MSG_WR_STS_ID	0xPPPW.TTTT.	ARM Cortex-M0 response to a write with argument ID=0xPPP* from the task with ID = 0xTTTT. Response is coded in W; W = 5...7 5 = write was successful 6 = write failed 7 = reserved

Fig 8. Message types

2.2.3 Message queue API set

The following APIs are provided:

Table 2. Message Queue APIs

Function name	Module	Return type	parameters
IPC_masterInitQueue	ipc_queue.c	void	cmdToken* cmdBuf uint32_t cmdBufSize msgToken* msgBuf uint32_t msgBufSize
IPC_slaveInitQueue	ipc_queue.c	void	void
IPC_slaveFlushMsgQueue	ipc_queue.c	qStat	void
IPC_masterFlushCmdQueue	ipc_queue.c	qStat	void
IPC_getCmdType	ipc_queue.c	cmd_t	cmdToken* item
IPC_masterPushCmd	ipc_queue.c	qStat	cmdToken* item
IPC_slavePopCmd	ipc_queue.c	qStat	maxCmd_t* item
IPC_cmdNotifySlave	ipc_queue.c	void	void
IPC_getMsgType	ipc_queue.c	msg_t	msgToken* token
IPC_slavePushMsg	ipc_queue.c	qStat	msgToken* item
IPC_masterPopMsg	ipc_queue.c	qStat	maxMsg_t* item
IPC_msgNotifyMaster	ipc_queue.c	void	void
IPC_msgPending	ipc_queue.c	uint8_t	void
IPC_cmdPending	ipc_queue.c	uint8_t	void

2.2.3.1 IPC_masterInitQueue

Initializes the IPC communication queues (address ranges, read and write pointers, contents of the command and message buffer), configures and enables the IPC interrupt on the master side.

2.2.3.2 IPC_slaveInitQueue

Configures and enables the IPC interrupt on the slave side.

2.2.3.3 IPC_slaveFlushMsgQueue

Called on the slave side and resets the queue to a known state (empty). The contents of the queue are lost.

The function always returns the queue status value QEMPTY.

2.2.3.4 IPC_masterFlushCmdQueue

Called on the master side and resets the command queue to a known state (empty). The contents of the queue are lost.

The function returns the queue status value QEMPTY.

2.2.3.5 IPC_getCmdType

Used to determine a command type from the cmdToken parameter.

Returns the type of command supported, CMD_RD_ID or CMD_WR_ID.

In case of error, returns INVALID_CMD.

2.2.3.6 IPC_getMsgType

Used to determine a message type from the msgToken parameter.

Returns the type of command supported, CMD_RD_ID or CMD_WR_ID.

In case of error, returns INVALID_MSG.

2.2.3.7 IPC_masterPushCmd

Inserts a command in the command queue.

The parameter cmdToken* is a pointer to a command type structure. The function inserts the command in the queue, if there is enough free space to hold it, and returns the status of the operation.

If the operation is successful, it returns QINSERT.

If there is no more space in the queue, it returns QFULL without inserting the message.

In case the message is not valid (the passed header is incorrect) it returns QERROR.

2.2.3.8 IPC_slavePushMsg

Inserts a message in the message queue.

The parameter ipcMsgToken* is a pointer to a message type structure. The function inserts the message in the queue by copy, if there is enough free space to hold it, and returns the status of the operation.

If the operation is successful, it returns QINSERT.

If there is not enough space in the queue, it returns QFULL without inserting the message.

If the message is not valid (the header is incorrect) it returns QERROR.

2.2.3.9 IPC_cmdNotifySlave

This function is used by the master to trigger an interrupt on the slave, after one or more commands have been inserted into to command queue.

2.2.3.10 IPC_msgNotifyMaster

This function is used by the slave to trigger an interrupt on the master side, after one or more commands have been inserted into to command queue.

2.2.3.11 IPC_slavePopCmd

This function is used by the slave to extract a command from the queue.

The parameter is a pointer to a maxCmd_t structure, which is large enough to hold the biggest command that can be retrieved from the queue, since it is not known which kind of command has been posted by the master.

The command data is retrieved from the queue and copied within the provided structure pointer, after which the original contents of the queue get invalidated and QVALID is returned.

When all commands are retrieved, the function returns QEMPTY and a call to *IPC_cmdPending* would return NO_TOKEN if no new commands were posted by the master in the meantime.

In case the retrieved command is not valid (the header is incorrect) it returns QERROR.

The function is able to retrieve only one command at the time.

2.2.3.12 IPC_masterPopMsg

This function is used to extract a message from the queue.

The parameter is a pointer to a maxMsg_t structure, which is large enough to hold the biggest message that can be retrieved from the queue, since it is not known which kind of message has been posted by the slave.

Otherwise, the message data is retrieved and copied within the provided structure pointer, after which the contents of the queue get invalidated. After this, the read pointer is updated.

When all messages are retrieved, the function returns QEMPTY and a call to *IPC_msgPending* would return NO_TOKEN if no new messages were posted by the slave in the meantime.

In case the message is not valid (the header is incorrect) it returns QERROR.

The function is able to retrieve only one message at the time.

2.2.3.13 IPC_cmdPending

This function is called on the slave side and returns the status of the command queue.

If there are new commands in the queue, it returns PENDING (true)

If the message queue is empty, it returns NO_TOKEN (false)

Can be used within C code, for example in an “if” flow control statement.

Note: this function is used on the slave side. Calling this function on the master side will lead to unpredictable results.

2.2.3.14 IPC_msgPending

This function is called on the master side and returns the status of the message queue.

If there are new messages in the queue, it returns PENDING (true)

If the message queue is empty, it returns NO_TOKEN (false)

Can be used within C code, for example in an “if” flow control statement.

Note: this function is used on the master side. Calling this function on the slave side will lead to unpredictable results.

2.2.4 Command and Message header generation macros

To simplify the preparation of a command and message headers, some helper macros are provided according to the header definitions detailed in Fig 7 and Fig 8.

2.2.4.1 Command header macros

MAKE_READ_CMD_HEADER(rdCmd,id,arg)

MAKE_WRITE_CMD_HEADER(wrCmd,id,arg)

The first parameter is the name of the rdCmd or wrCmd structure, id is the task ID and arg is the argument value.

2.2.4.2 Message header macros

MAKE_SRV_MSG_HEADER(srvMsg,id,sType)

The first parameter is the name of the srvMsg structure, id is the task ID and sType is the type of service requested.

MAKE_RD_MSG_HEADER(rdMsg,id,arg)

The first parameter is the name of the rdMsg structure, id is the task ID and arg is the argument value.

MAKE_RDSTS_MSG_HEADER (rdStsMsg,id,arg,failCode)

The first parameter is the name of the rdStsMsg structure; id is the task ID, arg is the argument value and failCode needs to be set to INVALID_ARG.

MAKE_WRSTS_MSG_HEADER(wrStsMsg,id,arg,response)

The first parameter is the name of the wrStsMsg structure, id is the task ID, arg is the argument value and response needs to be set to WRITE_SUCCESSFUL or WRITE_FAILED.

2.2.5 Message Queue API example usage flowcharts

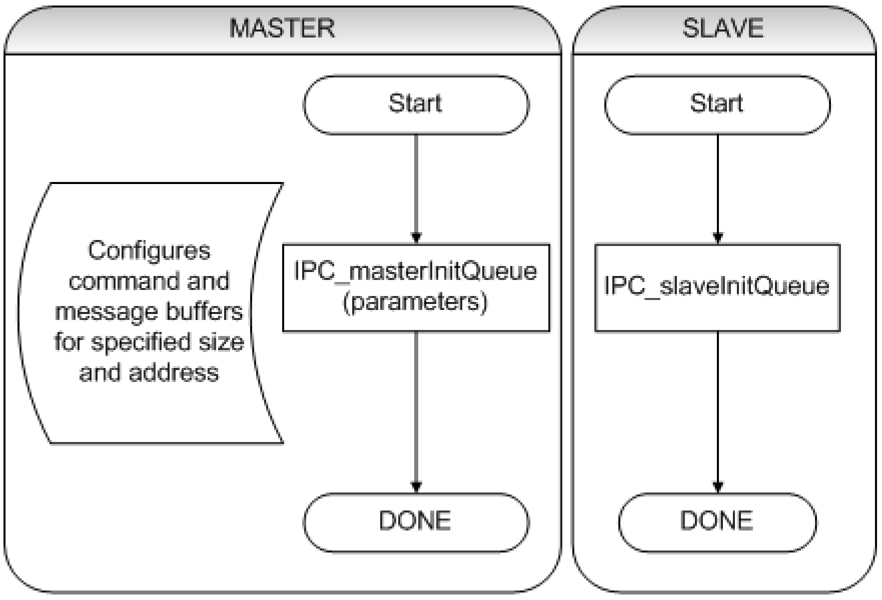

MASTER
SLAVE
Start
Start
Configures command and message buffers for specified size and address
IPC_masterInitQueue (parameters)
IPC_slaveInitQueue
DONE
DONE

Fig 9. Message queue setup

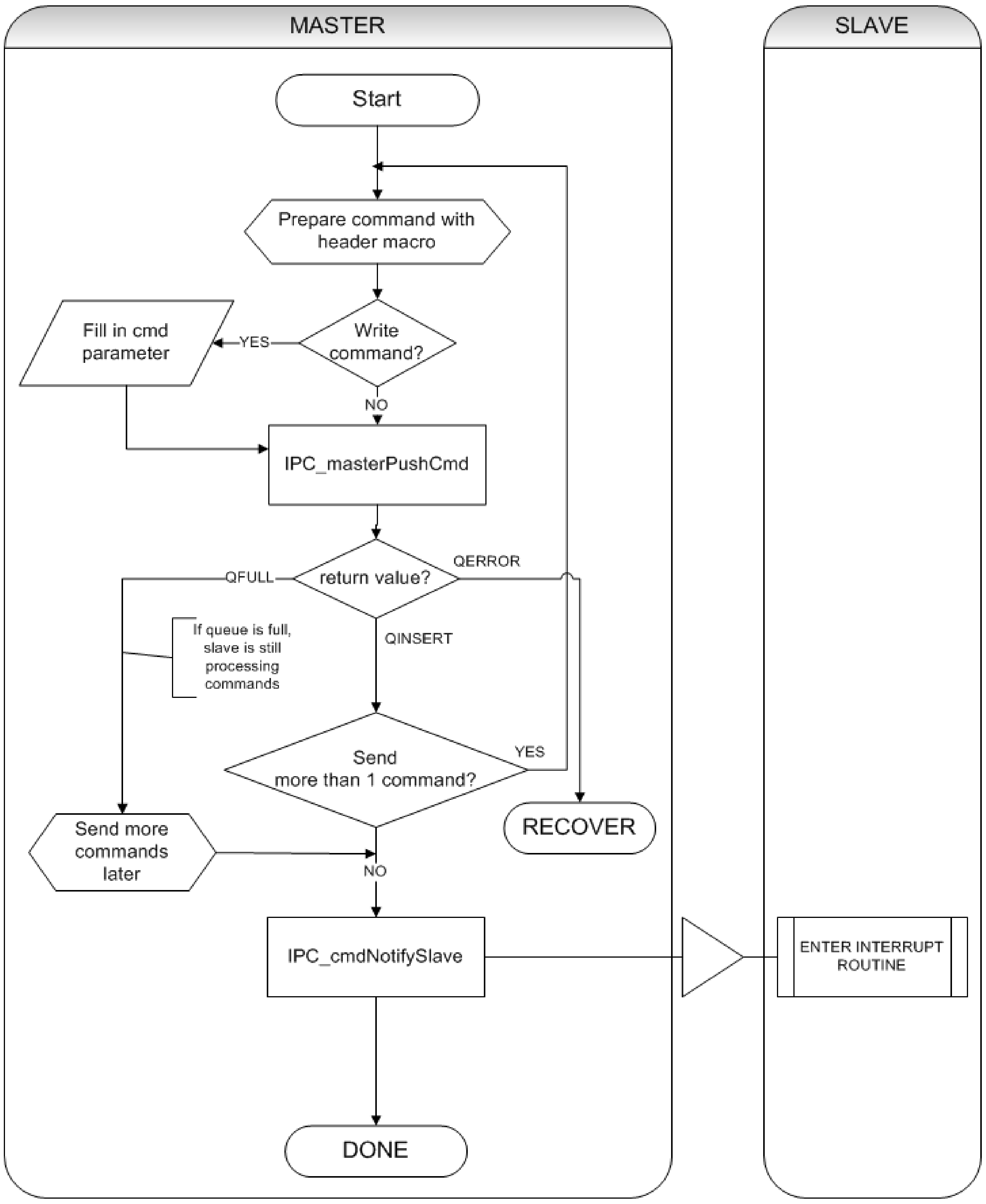


Fig 10. Master sending a command to the slave processor

Note: in the example shown in Fig 10, the sending processor (in this case the master) first fills the queue with the messages he wants to send, and then triggers the interrupt.

In some cases, it could be desirable that the receiving processor immediately starts emptying the queue.

To achieve that, an interrupt could be triggered by calling IPC_cmdNotifySlave (or IPC_msgNotifyMaster) after the first data item has been pushed into the queue.

In this way the receiving processor will start earlier to process the data.

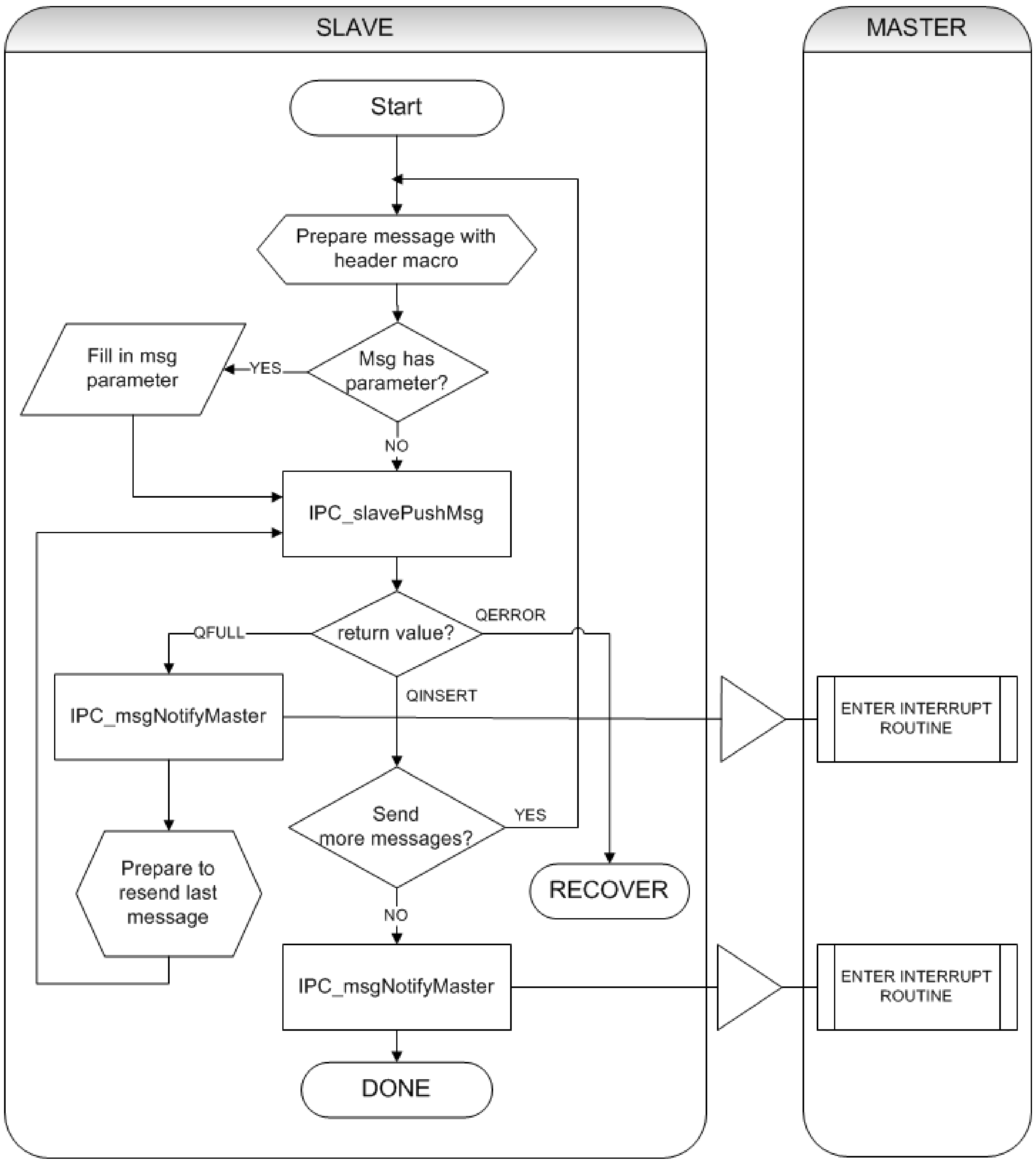


Fig 11. Slave sending a message to the master processor

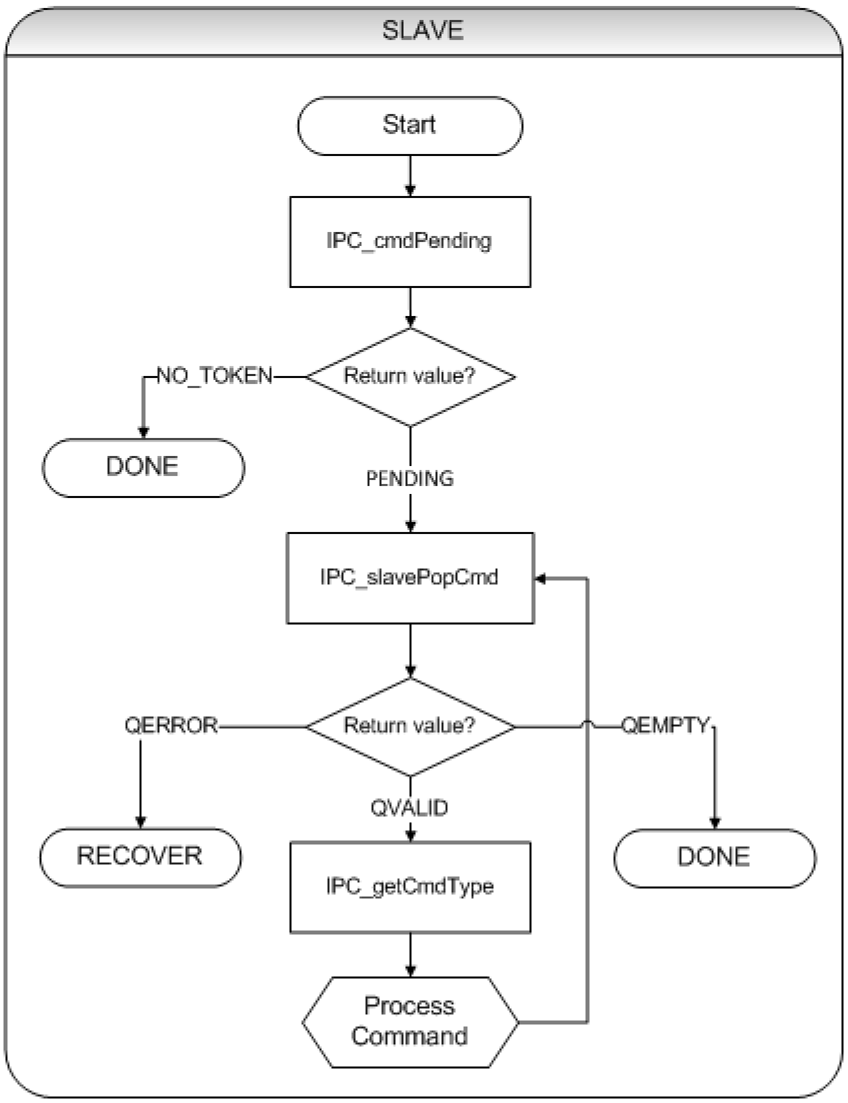


Fig 12. Slave processing a command from the master

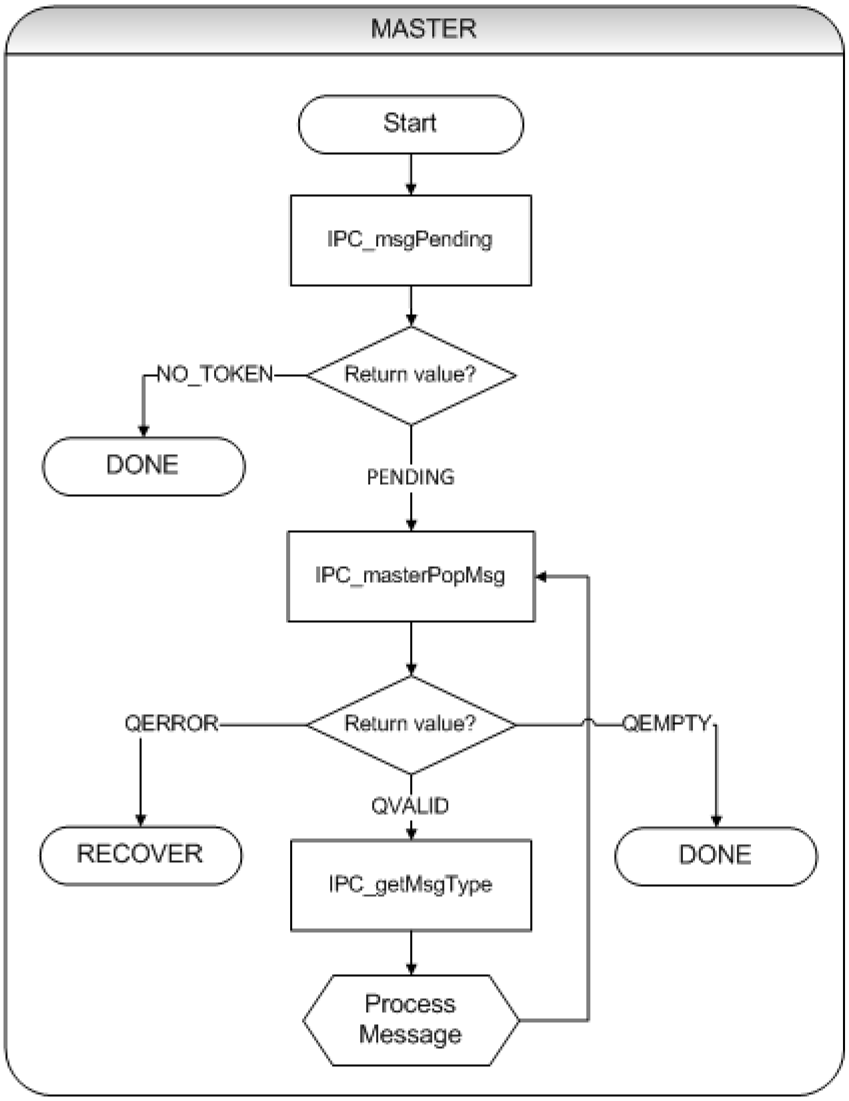


Fig 13. Master processing a message from the slave

2.3 Mailbox

This is an alternative implementation of an IPC model.

The principle is based on the concept of a “mailbox”, that is, a placeholder in RAM memory where the sending processor can place a “message” for the receiving processor. A sending processor manages a “local” mailbox where it can receive a message, and can send a message to a “remote” mailbox (which is managed by the remote processor).

A mailbox is defined by the following items:

- Message type
- Message id, which could be a progressive number. This can be useful if the transmission of a specific message is split into several transactions which have the same message type, e.g., so that the ID value is used to identify the message order
- 32-bit parameter, useful to pass additional data to the other processor, for example in form of a 32-bit pointer
- Callback function, which can be executed from the receiving processor when a new message has been detected in the mailbox, within the interrupt routine context.

  The purpose of the callback function is to allow the application to perform some operations as soon as the mailbox interrupt has been received and the presence of a message has been detected in the mailbox. This to reduce the IPC interrupt processing latency, where required.

All of the above items is application specific and can be freely defined by the user.

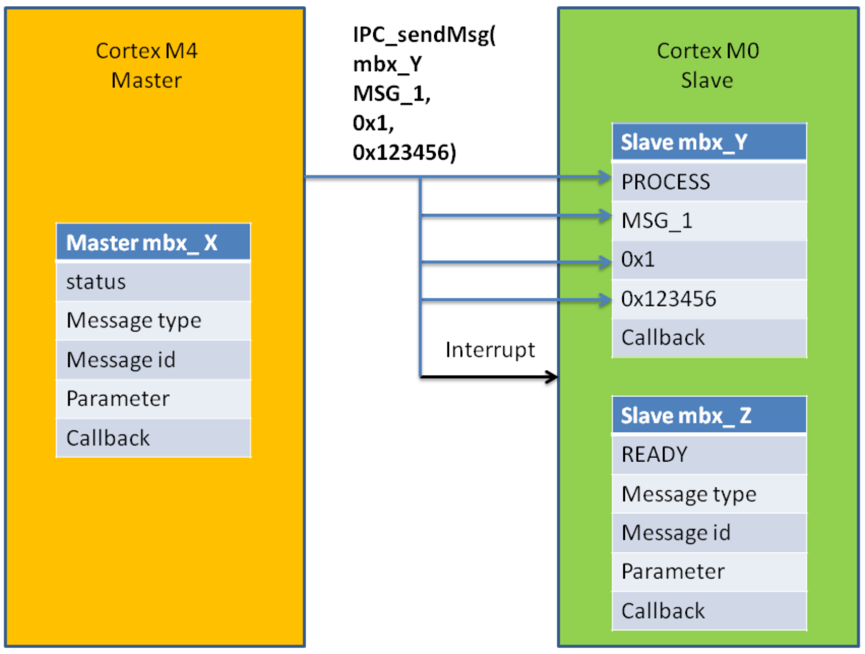


Fig 14. Mailbox based IPC

Fig 14 shows an example where there is one mailbox (X) defined on the master side and two mailboxes (Y and Z) defined on the slave side.

The master is calling the IPC_sendMsg function to send a message of type MSG_1 to the mailbox Y of the slave, with message id 0x1 and parameter 0x123456.

After the data has been written, an interrupt is triggered to the slave.

The mailbox status can be queried to determine if:

- a message can be sent (mailbox is READY), or
- if the receiving core still needs to process it (mailbox is BUSY), or
- if a processing error occurred (ERROR).

Once a message has been posted to the mailbox, the remote mailbox status gets updated (set to PROCESS) and an interrupt is triggered on the receiving side.

Within the application side, the reception of a message in a mailbox will set a flag for signaling that there are pending items. There is one flag dedicated for each defined mailbox, which can take the value MSG_PENDING or NO_MSG.

For deferred processing, this allows the application to query the message pending flags at its convenience, and react as desired.

Note: the mailbox flags will be set **after** the callback function has been executed. Within the callback function the mailbox message, the message Id and the parameter are passed as parameters. The mailbox status shall not be queried by this user defined callback function. The application is responsible to clear the mailbox status when finished, and clear the messaging flags, by calling the appropriate API function.

The application can define any number of mailboxes on each receiving side, and it is a user choice how to logically associate a specific mailbox with the desired IPC messages.

An example configuration could use a dedicated mailbox for each “logical event” or message that needs to be communicated to the other core. In this way, the mailbox number automatically identifies the type of service requested, and there is no need for the application to determine the message type, which is then implicit.

On the other hand, an application could be configured to have just one single mailbox, over which all possible messages are sent to the other core. In this scenario, there is less memory being needed for the mailbox system, at the price of having to query on each received message type, in order to determine which kind of service is being requested.

Any other mixed configuration is possible, allowing complete flexibility for the configuration of the messaging scheme in terms of mapping between message types and mailboxes.

The number of total mailboxes can be different on each core side, so the system does not need to be symmetric in terms of functionality. The user can define one mailbox on the master side, and 5 mailboxes on the slave side, for example.

Typically messages which have a low transmission rate, do not have stricter requirements in terms of latency, or are expected to be mutually exclusive, might share one same mailbox.

Other messages which have a much higher transmission rate, or stricter requirements in terms of latency, might be dedicated to one (or more) separate mailboxes to improve the communication throughput.

2.3.1 Implementation details

The module is composed of the following files:

- api\mbx\inc
  - ipc_mbx.h
- api\mbx\src
  - ipc_mbx.c
  - ipc_buffer.c

The logical messages for each mailbox are defined within the master_ipc_msg.h and slave_ipc_msg.h files. These are part of the platform wide configuration files and are not part of the Mailbox API implementation.

The module ipc_buffer.c defines the structures used for holding the master and slave mailboxes.

2.3.1.1 Mailbox status

The mailbox status can be one of the following:

- READY

  The mailbox is ready to accept a new message
- PROCESS

  The mailbox contains a new message which shall be processed. The mailbox shall not be overwritten with new information. The remote processor should have received an interrupt notification, but has not started yet processing the message
- BUSY

  The mailbox is still occupied by a new message, so its contents shall not be overwritten, but the remote processor is processing the message
- ERROR_OCCURRED

  The remote processor handled the message but there was an execution error which prevented a successful completion of the command

A remote mailbox shall always be queried for the READY or ERROR_OCCURRED status before the sending processor issues a new message for the remote CPU (by using the *IPC_sendMsg* API).

The local processor can query the mailbox status periodically, to find out if it can send a new message to the remote processor when the mailbox is READY again.

Alternatively, the user could implement a higher level protocol where the receiving processor always acknowledges back to the sending processor with a “work completed” message. In this scenario, the sending processor does not need to poll for the mailbox status.

It is application dependant, which strategy fits best the application, and both ways of communication might be used and even mixed as desired (use poll on some mailboxes, use an acknowledge interrupt for others)

Once a new message is placed into the mailbox, by calling the *IPC_sendMsg* API, the status of the remote mailbox is automatically changed to PROCESS and afterwards an interrupt gets triggered to the remote processor for notification.

The remote processor will be notified by the IPC interrupt. Within the interrupt context the associated callback function (if used) is executed and afterwards the mailbox flag is changed from NO_MSG to MSG_PENDING.

The remote processor shall set the mailbox status to BUSY once it acknowledges the new message, and starts processing it. The mailbox status shall be kept to BUSY for all the time of processing, or at least as long as the information within the mailbox has not been completely retrieved on the remote side.

If an error occurred, the remote processor shall set the mailbox status to ERROR_OCCURRED. It is application dependent, and user specified, how the system handles this situation.

The sending processor might, for example, decide to resend the same message again, or perform some other types of recovery actions.

After the processing of the message, or whenever the mailbox content has been acquired and the receiving processor can accept a new message, the status of the mailbox can be set back to READY. This will signal the remote processor that it can send a new message.

2.3.2 Mailbox API set

The following APIs are provided:

Table 3. Mailbox APIs

Function name	Module	Return type	Parameters
IPC_initMasterMbx	ipc_mbx.c	void	CbackItem cbackTable[] Mbx* masterMbxPtr Mbx* slaveMbxPtr
IPC_initSlaveMbx	ipc_mbx.c	void	CbackItem cbackTable[] Mbx* masterMbxPtr Mbx* slaveMbxPtr
IPC_queryLocalMbx	ipc_mbx.c	mbxStat_t	mbxId_t mbxNum
IPC_getMsgType	ipc_mbx.c	msg_t	mbxId_t mbxNum
IPC_getMsgId	ipc_mbx.c	msgId_t	mbxId_t mbxNum
IPC_getMbxParameter	ipc_mbx.c	mbxParam_t	mbxId_t mbxNum
IPC_queryRemoteMbx	ipc_mbx.c	mbxStat_t	mbxId_t mbxNum
IPC_resetMbxFlag	ipc_mbx.c	void	mbxId_t mbxNum
IPC_lockMbx	ipc_mbx.h	void	mbxId_t mbxNum
IPC_freeMbx	ipc_mbx.h	void	mbxId_t mbxNum
IPC_setMbxErr	ipc_mbx.h	void	mbxId_t mbxNum
IPC_sendMsg	ipc_mbx.c	void	mbxId_t mbxNum msg_t msg msgId_t msgNum mbxParam_t param

2.3.2.1 IPC_initMasterMbx

This function must be called on the master side to initialize the mailbox system, and needs to be called before using any of the mailbox APIs. It is responsible for clearing the status, the internal variables and enabling the IPC interrupts.

It also initializes the defined user callback functions within the mailbox system by plugging in the callback function pointers. The callback table has to be defined in file *master_mbx_callbacks.c*, which is part of the platform wide configuration. The table pointer is passed as a function argument.

The other two parameters are the addresses of the master and slave mailbox locations, i.e. where in memory the mailboxes are located. This is also a project wide configuration which is defined by the user.

2.3.2.2 IPC_initSlaveMbx

This function must be called on the slave side to initialize the mailbox system, and needs to be called before using any of the mailbox APIs. It is responsible for clearing the status, the internal variables and enabling the IPC interrupts.

It also initializes the defined user callback functions within the mailbox system by plugging in the callback function pointers. The callback table has to be defined in file *slave_mbx_callbacks.c*, which is part of the platform wide configuration. The table pointer is passed as a function argument.

The other two parameters are the addresses of the master and slave mailbox locations, i.e. where in memory the mailboxes are located. This is also a project wide configuration which is defined by the user.

Note: in case for a specific mailbox no callback function is desired, a special function named *IPC_dummyCallback* is provided within the API definition, which is just returning to the caller without doing anything (an empty function). This might be specified as a default, for all entries in the master or slave callback table which are unused.

2.3.2.3 IPC_queryLocalMbx

This function can be used by a processor to query its own (local) mailbox to determine the status. The passed parameter is the mailbox number which is defined by the user via an enumerated type.

2.3.2.4 IPC_getMsgType

This function is used to retrieve the type of command received in the local mailbox.

The passed parameter is the mailbox number, and the returned value is an enumerated type which is defined by the user within the *master_ipc_msg.h* and *slave_ipc_msg.h* files.

By default this is implemented as an 8-bit unsigned integer.

2.3.2.5 IPC_getMsgId

This function is used to retrieve the message ID received in the local mailbox. The parameter is the mailbox number, and the function returns one msgId_t type value. By default this is implemented as a 16-bit unsigned integer.

2.3.2.6 IPC_getMbxParameter

This function is used to retrieve the local mailbox parameter, which is implemented with a 32-bit unsigned integer type.

2.3.2.7 IPC_resetMbxFlag

This function is used by the receiving processor to clear the notification flag which gets set once a message is posted and the corresponding IPC interrupt has been triggered.

The parameter is the mailbox number.

Since it needs to run in a critical section, interrupts are disabled in a short period of time.

2.3.2.8 IPC_queryRemoteMbx

This function is used to determine the status of a “remote” mailbox, and can be used by the local processor to find out if the remote processor is ready to accept a new message.

The parameter is the mailbox number.

2.3.2.9 IPC_sendMsg

This function is used to send a message to a specific mailbox. The parameters are the mailbox number, the message type, the message ID, and the parameter.

The application should check the mailbox status to be READY before calling this function to send a message; otherwise the mailbox contents will be overwritten.

This function also takes care of issuing an interrupt to the remote processor for notifying about the presence of a new message in the mailbox.

2.3.2.10 IPC_lockMbx

This macro (function) is used locally to set the mailbox status to BUSY by the receiving processor. The parameter is the mailbox number.

2.3.2.11 IPC_freeMbx

This macro (function) is used locally to set the mailbox status to READY. This signals to the remote processor that a new message can be sent. The parameter is the mailbox number.

2.3.2.12 IPC_setMbxErr

This macro (function) is used locally to set the mailbox status to ERROR. This signals to the remote processor that an error occurred. The parameter is the mailbox number.

2.3.3 Callback function definition

A callback function is executed within the context of the interrupt routine, before the flag variable associated with the mailbox is set.

The parameters provided to the callback are the message type, the message ID, and the mailbox parameter.

The callback functions are defined by the user within the following files:

- master_mbx_callbacks.c
- master_callbacks.h
- slave_mbx_callbacks.c
- slave_mbx_callbacks.h

The callback function prototypes are defined as:

```
void (*mbxCallback_t) (msg_t msg, msgId_t idNum, mbxParam_t param)
```

The callback table needs to have one entry for each defined mailbox, and defines the association between the mailbox number and the related callback.

The order in the table is not important, since the callback function addresses are plugged-in at runtime by the IPC initialization routines.

2.3.4 Mailbox API usage example flowcharts

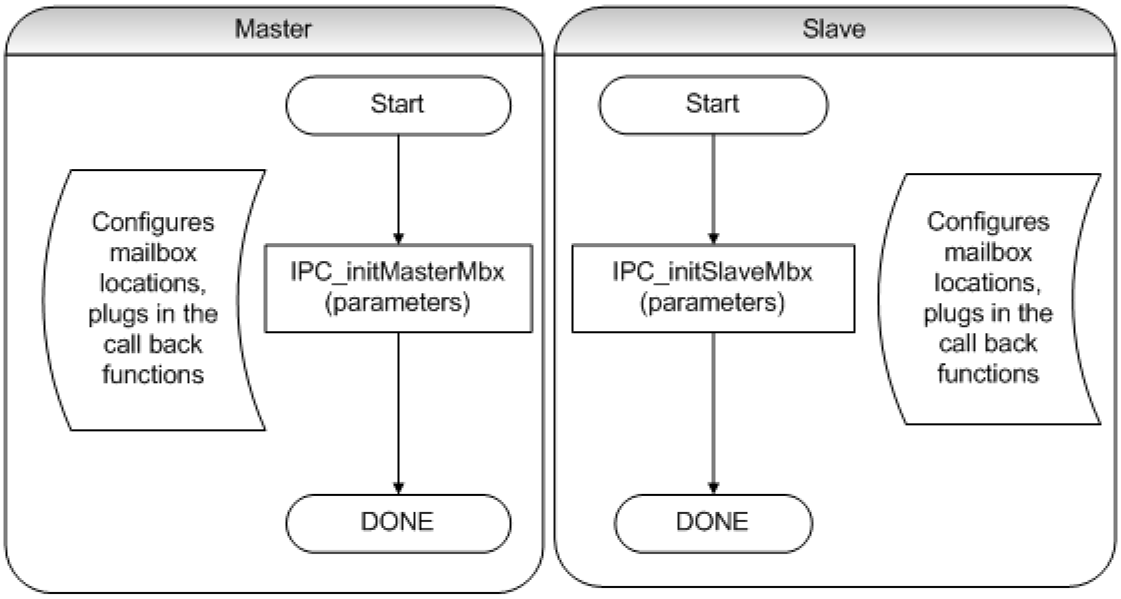

Master
Start
Configures mailbox locations, plugs in the call back functions
IPC_initMasterMbx (parameters)
DONE
Slave
Start
IPC_initSlaveMbx (parameters)
Configures mailbox locations, plugs in the call back functions
DONE

Fig 15. Mailbox setup

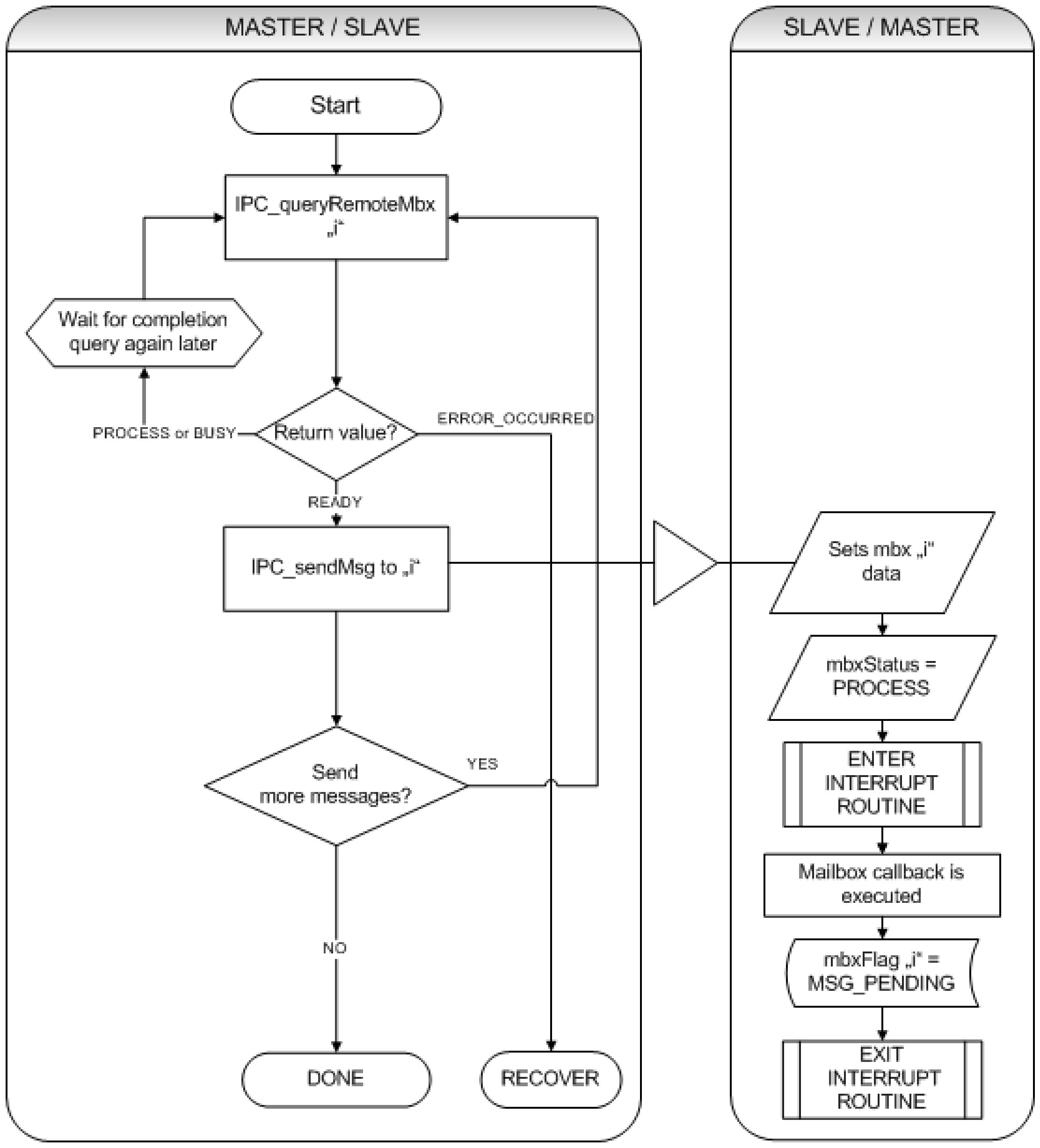


Fig 16. Sending a message to a mailbox (from master or slave)

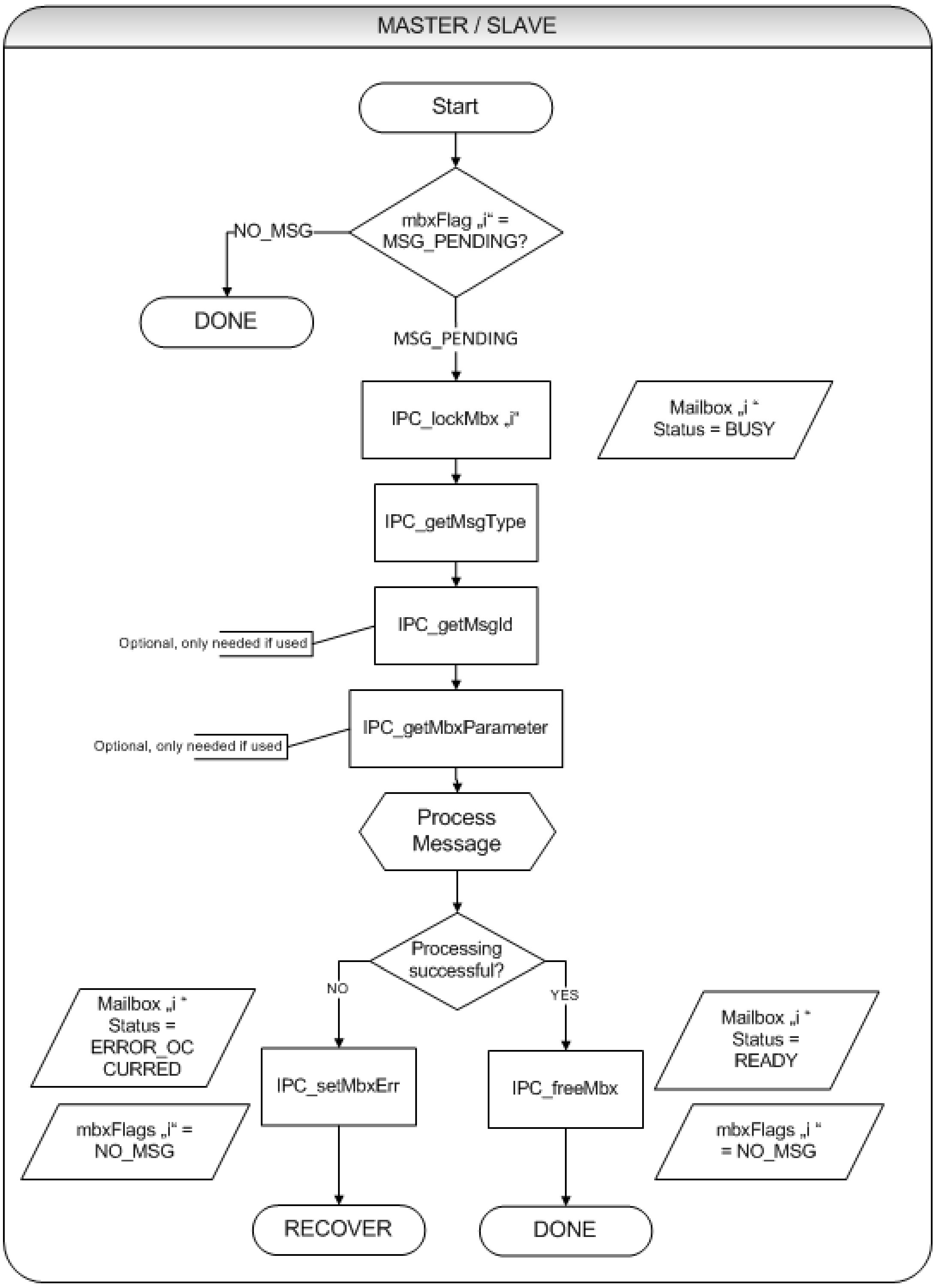


Fig 17. Processing a received mailbox message (master or slave)

2.4 Slave initialization routines

The following functions are provided by the master processor to initialize the system

Table 4. Slave processor initialization functions

Function name	Module	Return value	Parameters
IPC_downloadSlaveImage	ipc_queue.c ipc_mbx.c	void	uint32_t slaveRomStart const unsigned char slaveImage[] uint32_t imageSize
IPC_startSlave	ipc_queue.c ipc_mbx.c	void	void
IPC_haltSlave	ipc_queue.c ipc_mbx.c	void	void

2.4.1 IPC_downloadSlaveImage

This function is responsible for downloading a slave processor image to a specific memory address, assuming this is located on a volatile memory (RAM). The address where the image should be downloaded to is defined by the parameter slaveRomStart.

The slave image consists of an array of bytes, which is passed via the slaveImage array parameter. The third parameter imageSize represents the size in bytes of the slave processor image.

It is the user responsibility to ensure that these parameters are all consistent between the master and slave linker scatter files, and the definitions provided in platform_config.h.

This function prepares the slave image for execution, but does not start the slave processor. If the slave processor is already out of reset and running, the function will place it back in reset before downloading the image.

2.4.2 IPC_startSlave

This function is used to take the slave processor out of reset and start it. Before calling this function, it is mandatory that a valid slave processor image has been downloaded first by the master processor via the *IPC_downloadSlaveImage* function; otherwise the behavior will be undefined.

2.4.3 IPC_haltSlave

This function places the slave processor back in reset, but does not change the content of its code or data sections. The slave processor will remain in reset, until the *IPC_startSlave* function is called again.

The application might also download a different slave image by calling *IPC_downloadSlaveImage* before the next *IPC_startSlave* is called.

3. Configuration options and platform settings

3.1 Interrupt

In the file platform_config.h the user can specify the system configuration for the message queue system. The following IPC specific build options are provided:

3.1.1 PLATFORM

Used to specify which hardware board is being used for the tests. Currently supported platforms are:

HITEX_BOARD

3.1.2 DEVICES

Used to specify which microcontroller device is being used. Currently supported devices are:

LPC43xx

3.1.3 SLAVE_IMAGE_FILE

Used to specify the name of the C file which defines the byte array holding the slave image. Default value is the string “CM0_image.c”

3.1.4 MASTER_INTERRUPT_PRIORITY

Used to specify the priority of the IPC interrupt on the **master** processor. The IPC interrupt gets triggered when a slave calls the *IPC_sendInterrupt* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 7 as the Cortex-M4 master has 3 priority bits implemented in the NVIC.

3.1.5 SLAVE_INTERRUPT_PRIORITY

Used to specify the priority of the IPC interrupt on the **slave** processor. The IPC interrupt gets triggered when a master calls the *IPC_ sendInterrupt* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 3 as the Cortex-M0 slave has 2 priority bits implemented in the NVIC.

3.2 Message queue

In the file platform_config.h, the user can specify the system configuration for the message queue system. The following IPC specific build options are provided:

3.2.1 PLATFORM

Used to specify which hardware board is being used for the tests. Currently supported platforms are:

HITEX_BOARD

3.3 DEVICES

Used to specify which microcontroller device is being used. Currently supported devices are:

LPC43xx

3.4 SLAVE_IMAGE_FILE

Used to specify the name of the C file which defines the byte array holding the slave image. Default value is the string “CM0_image.c”

3.4.1 MASTER_CMDBUF_SIZE

Integer value used to specify how many 32-bit items the command buffer can hold. Messages with bigger total size than the simplest item (32-bit) will consume multiple items for each message.

3.4.2 MASTER_QUEUE_PRIORITY

Used to specify the priority of the IPC interrupt on the **master** processor. The IPC interrupt gets triggered when a slave calls the *IPC_msgNotifyMaster* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 7 as the Cortex-M4 master has 3 priority bits implemented in the NVIC.

3.4.3 SLAVE_MSGBUF_SIZE

Integer value used to specify how many 32-bit items the command buffer can hold. Messages with bigger size than the simplest item (32-bit) will consume multiple items for each message.

3.4.4 SLAVE_QUEUE_PRIORITY

Used to specify the priority of the IPC interrupt on the **slave** processor. The IPC interrupt gets triggered when a master calls the *IPC_cmdNotifySlave* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 3 as the Cortex-M0 slave has 2 priority bits implemented in the NVIC.

3.4.5 MASTER_ROM_START, MASTER_ROM_LEN

Specifies the master code memory location and the size of the assigned range

3.4.6 MASTER_RAM_START, MASTER_RAM_LEN

Specifies the master data memory location and the size of the assigned range

3.4.7 SLAVE_ROM_START, SLAVE_ROM_LEN

Specifies the slave code memory location and the size of the assigned range

3.4.8 SLAVE_RAM_START, SLAVE_RAM_LEN

Specifies the slave data memory location and the size of the assigned range

3.4.9 MASTER_CMD_BLOCK_START

Specify the memory location at which the Command interface structure and buffer are located

3.4.10 SLAVE_MSG_BLOCK_START

Specify the memory location at which the Message interface structure and buffer are located

Note: items 3.4.5 to 3.4.10 need to be consistent with the linker scatter file configurations for the two processors.

3.5 Mailbox

In the file platform_config.h, the user can specify the system configuration for the mailbox system. The following build options are provided:

3.5.1 PLATFORM

Used to specify which hardware board is being used for the tests. Currently supported platforms are:

HITEX_BOARD

3.5.2 DEVICES

Used to specify which microcontroller device is being used. Currently supported devices are:

LPC43xx

3.5.3 MASTER_MAILBOX_PRIORITY

Used to specify the priority of the IPC interrupt on the **master** processor. The IPC interrupt gets triggered when the slave processor calls the *IPC_sendMsg* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 7 as the Cortex-M4 master has 3 priority bits implemented in the NVIC

3.5.4 SLAVE_MAILBOX_PRIORITY

Used to specify the priority of the IPC interrupt on the **slave** processor. The IPC interrupt gets triggered when the master processor calls the *IPC_sendMsg* function.

This follows the CMSIS convention, e.g., on the LPC4300 it can assume a value from 0 to 3 as the Cortex-M0 slave has 2 priority bits implemented in the NVIC.

3.5.5 MASTER_ROM_START, MASTER_ROM_LEN

Specifies the master code memory location and the size of the assigned range

3.5.6 MASTER_RAM_START, MASTER_RAM_LEN

Specifies the master data memory location and the size of the assigned range

3.5.7 SLAVE_ROM_START, SLAVE_ROM_LEN

Specifies the slave code memory location and the size of the assigned range

3.5.8 SLAVE_RAM_START, SLAVE_RAM_LEN

Specifies the slave data memory location and the size of the assigned range

3.5.9 MASTER_MBX_START, MASTER_MBX_LEN

Specify the memory location at which the mailbox data for the master processor is located.

3.5.10 SLAVE_MBX_START, SLAVE_MBX_LEN

Specify the memory location at which the mailbox data for the slave processor is located.

Note: items 3.5.5 to 3.5.10 need to be consistent with the linker scatter file configurations for the two processors.

4. Application examples

4.1 Interrupt, Mailbox, Message Queue

In these examples, the two processors exchange asynchronously messages, and the application shows how the interrupt, mailbox or message queue APIs could be used to send signals, messages, access the data, and use the associated application flags.

4.2 Mailbox, Message Queue with RTOS support

In these examples, the FreeRTOS v. 7.0.2 operating system is running on both processors independently as separate images.

Two unofficial ports for the Cortex-M4 (without FPU support, is actually the Cortex-M3 port) and the Cortex-M0 processor are provided, and the configuration file for the RTOS (freeRTOSConfig.h) is included within each main source folder of the examples.

A template for the configuration file can be found within the FreeRTOS\config template of the port.

For an official port of the FreeRTOS operating system on these two architectures, please refer to the website FreeRTOS.org.

The functions which initialize the mailbox or message queue systems, as well as the communication APIs, are provided within two separately defined tasks.

Note: in case of the mailbox example, it is potentially possible for the application to call RTOS related APIs within the provided callback function, if used.

In this case, like for other OS related interrupts (refer to the freeRTOS documentation), the user has to take care that the priority assigned to the IPC interrupt routine (and thus the priority at which the callback will be executed) is configured within the range assigned to the RTOS:

configKERNEL_INTERRUPT_PRIORITY to

configMAX_SYSCALL_INTERRUPT_PRIORITY

The freeRTOS kernel will take care that a priority ceiling mechanism is applied, for each interrupt which might request OS services, so that there are no preemption issues which might disrupt the scheduler operations.

5. Legal information

5.1 Definitions

Draft — The document is a draft version only. The content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included herein and shall have no liability for the consequences of use of such information.

5.2 Disclaimers

Limited warranty and liability — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

Right to make changes — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

Suitability for use — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

Applications — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

Export control — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

Evaluation products — This product is provided on an "as is" and "with all faults" basis for evaluation purposes only. NXP Semiconductors, its affiliates and their suppliers expressly disclaim all warranties, whether express, implied or statutory, including but not limited to the implied warranties of non-infringement, merchantability and fitness for a particular purpose. The entire risk as to the quality, or arising out of the use or performance, of this product remains with customer.

In no event shall NXP Semiconductors, its affiliates or their suppliers be liable to customer for any special, indirect, consequential, punitive or incidental damages (including without limitation damages for loss of business, business interruption, loss of use, loss of data or information, and the like) arising out the use of or inability to use the product, whether or not based on tort (including negligence), strict liability, breach of contract, breach of warranty or any other theory, even if advised of the possibility of such damages.

Notwithstanding any damages that customer might incur for any reason whatsoever (including without limitation, all damages referenced above and all direct or general damages), the entire liability of NXP Semiconductors, its affiliates and their suppliers and customer's exclusive remedy for all of the foregoing shall be limited to actual damages incurred by customer based on reasonable reliance up to the greater of the amount actually paid by customer for the product or five dollars (US$5.00). The foregoing limitations, exclusions and disclaimers shall apply to the maximum extent permitted by applicable law, even if any remedy fails of its essential purpose.

5.3 Trademarks

Notice: All referenced brands, product names, service names and trademarks are property of their respective owners.

6. List of figures

7. List of tables

8. Contents

Date of release: 20 August 2014
Document identifier: AN11177