AN4502
Application note

STM32 SMBus/PMBus™ expansion package for STM32Cube™

Introduction

This document describes the X-CUBE-SMBUS, STM32 SMBus/PMBus™ (system management bus/power management bus) firmware stack.

This stack is based on the specific STM32Cube™ HAL drivers available for all STM32 series, with the exception of older devices, in particular STM32F1, STM32F2, STM32F4 and STM32L1 series. These older devices are limited to SMBus 2.0 and are not compatible with the stack described in this AN. All other STM32 series (introduced after STM32F3) contain an I2C peripheral supporting the SMBus/PMBus™ stack introduced in this document. Some series are presented with examples. Others can have the stack configured using the STM32CubeMX embedded software package, X-CUBE-SMBUS.

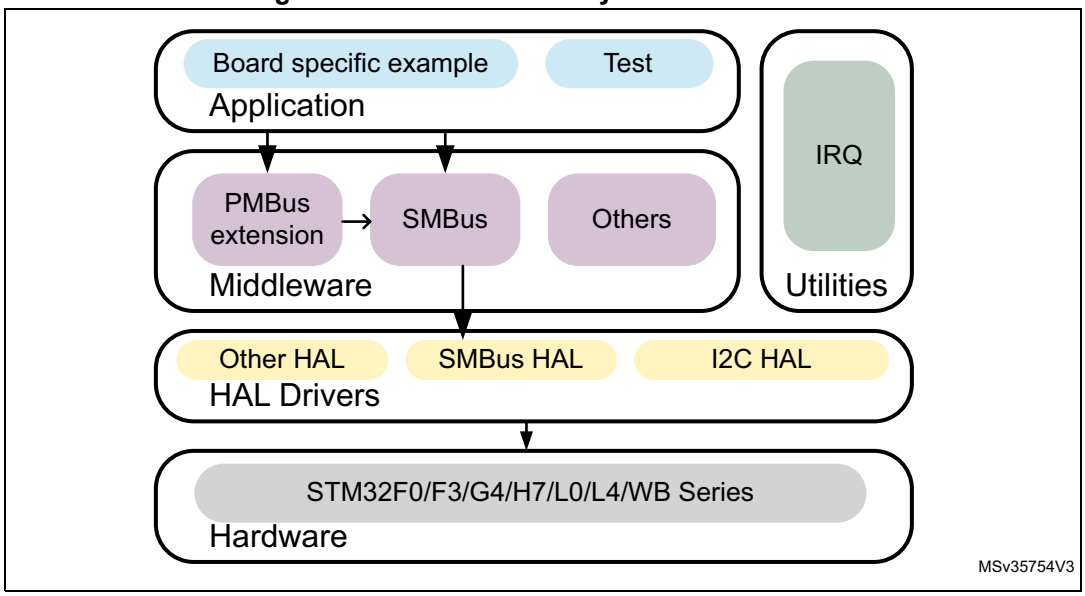
The SMBus was founded to standardize the power management related to internal communications, both in devices and in electronic systems. The SMBus builds on the widely adopted I^2C bus and defines the links and network layers of the OSI (open systems interconnection) model.

The PMBus™ uses the SMBus as its physical layer and adds command definitions along with other new features. Most of the new features fall into medium-to-high layers of the OSI model. The PMBus™ specifications are presented in this application note detailing the different aspects of their implementation.

The X-CUBE-SMBUS firmware provides the means to get information from the attached devices such as their manufacturer, their version and their capabilities. Once these informations are retrieved, the devices can be configured and managed using the SMBus transmitted commands.

By using this firmware, the devices also have the capability to asynchronously inform the bus host about events and errors. In its basic configuration the bus can be used to suspend and wake up simple devices. *Figure 1* presents a snapshot of the system architecture.

Figure 1. X-CUBE-SMBUS system architecture



Contents

List of tables

List of figures

1 General information

This firmware stack applies to STM32 Arm®[a]-based devices.

arm

Table 1. Definitions

Term	Description
Alert	Wired signal, active low, notifying SMBus host about urgency on SMBus device by an interrupt.
ARA	Alert response address
ARP	Address resolution protocol
HAL	Hardware abstraction layer (drivers)
LSb	Least significant bit
LSB	Least significant byte
Master	Master is any device that initiates SMBus transactions and drives the clock.
MSb	Most significant bit
MSB	Most significant byte
MSP	MCU specific package
PEC	Packet error check
PMBus	Power management bus
PSA	Persistent address
Slave	Slave is a target of an SMBus transaction which is driven by some master.
SMBus	System management bus
SMBus host	A host is a specialized master that provides the main interface to the system CPU. A host is a master-slave and supports the SMBus host notify protocol. There may be at most one host in a system.
SMBus device	A device may be designed so that it is never a master and always a slave. A device may act as a slave most of the time, but in special cases it may become a master. A device can also be designed to be a master only. A system host is an example of a device that acts as a host most of the time but that includes some slave behavior.
UDID	Unique device identifier – ARP specific value

a. Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

2 Resources

1. System Management Bus (SMBus) Specification Version 3.0 from www.smbus.org
2. PMBus Power Management Bus Specification Part I – Revision 1.3.1 from www.pmbus.org
3. PMBus Power Management Bus Specification Part II – Revision 1.3.1 from www.pmbus.org
4. PMBus Power System Management Protocol Application Note AN001 from www.pmbus.org

3 Features supported by the stack

3.1 SMBus features

The stack supports the following types of SMBus transactions (bus protocols) based on reference *[1]*:

- Quick command
- Send byte
- Receive byte
- Read byte/word
- Write byte/word
- Block write/read
- Process call
- Block process call
- SMBus host notify protocol
- ARP
- Zone commands
- PEC integrity protection
- Alert signal supported with an automated alert protocol treatment

Note: The Quick command, Receive byte and Zone read handling are implemented with limitations (see Section 6.4 for more details).

The implementation has been tested with 100 kHz, 400 kHz, and 1 MHz bus speeds with the STMicroelectronics Nucleo or Discovery boards.

Features not supported:

- Control signal
- Write protect
- Pin-driven address configuration
- Host side ARP management
- Any persistent attributes (the stack uses non-volatile data storage of its own)

These features are left for the customer to finish, as their implementation cannot be generalized.

3.2 PMBus features

The PMBus extends the above mentioned SMBus functionalities adding:

- Group command protocol
- Command table filled with command descriptors to ease application implementation
- Limited extended command support

The PMBus specification supported by X-CUBE-SMBUS firmware, is limited to part I and part II of the specification. The AVSBus™ (adaptive voltage scaling bus) interface described in part III of the specification is not supported at the moment of writing this application note (See *Section 2* for complete reference of the specifications).

4 SMBus stack handle

Each SMBus related action is called and executed with respect of the values and the settings stored at the SMBus stack handle structure. This structure keeps the basic peripheral configuration, the internal state machine status of the stack and the communicated data in the I/O buffer.

Some MCUs may support multiple SMBus connections using separate physical interfaces. Each physical interface must have its own stack handle.

4.1 The handle structure

The structure type is defines as follows:

```
typedef struct
{
  uint32_t      StateMachine;
  SMBUS_HandleTypeDef*  Device;
  uint8_t*      ARP_UDID;
  st_command*   CurrentCommand;
  st_command*   CMD_table;
  uint32_t      CMD_tableSize;
  SMBUS_ZoneStateTypeDef TheZone;
  uint16_t      Byte_count;
  uint16_t      SlaveAddress;
  uint8_t       OwnAddress;
  uint8_t       OpMode;
  uint8_t       SRByte;
  uint8_t       Buffer[];
}
SMBUS_StackHandle;
```

The structure attributes are explained in details in the following sections.

TheZone structure is part of the handle only if conditional flag PMBUS13 is defined. The structure is described in detail in *Zone configuration and associated data structure*.

4.1.1 StateMachine

The StateMachine holds the current processing state and any error encountered. Neat macros are defined to ease extraction of useful information from the StateMachine. For more information, refer to *Chapter 4.2*.

4.1.2 Device

Pointer to the HAL SMBUS handle that is linked to this stack. It keeps information about the HW peripheral instance and its SMBUS related configuration.

4.1.3 ARP_UDID

Pointer to the unique device ID used for address resolution protocol. This pointer may refer to a read only memory location. SMBus stack never attempts to modify this information.

4.1.4 CurrentCommand

Pointer to the command descriptor definition of a command currently processed. It usually points inside the command table. The command descriptor structure is documented in *Section 4.3*.

4.1.5 CMD_table

Pointer to the start of the list of supported command descriptors. It must be set before the stack initialization call. The command descriptors must follow the structure described in *Section 4.3*. The PMBus source file contains a predefined table with the PMBus commands defined in reference *[3]*.

Note: The stack by default searches for command definitions in CMD_table. For an alternate solution, the pointer can be redirected to another command descriptor list. Command table search method is implemented and may be modified in the *STACK_SMBUS_LocateCommand function (refer to Section 5.1.3). In some special cases as the ARP, the stack bypasses this table and look elsewhere (refer to Section 6.3).*

4.1.6 CMD_tableSize

Number of commands listed in CMD_table. It must be set before the stack initialization call.

4.1.7 TheZone

This structure maintains information about the read and the write zones assigned to the device when the zone support feature is enabled. Then also contains information about read and write zones currently used. The content of the structure is checked by slave devices to handle the zone command flow. For further information, refer to *Chapter 7.2.3*

4.1.8 Byte_count

The actual position in I/O buffer. The application function may use this value to check size of the input after calling the *STACK_SMBUS_GetBuffer* command(*Section 5.5.2*).

4.1.9 SlaveAddress

It is used by the host to store the address of the device that the stack is currently communicating with.

It is used by the slave to remember the address under which the command that is currently being processed was accepted. It is useful both for devices that support more than one address as well as for general call and zone access.

4.1.10 OwnAddress

It must be initialized with the value of the device address, or left as zero in case of the device relying on the ARP. For host, this attribute stores the received address after the alert signal treatment, as the host is only addressed in the host's notify command.

Note: For ARP enabled systems, it is important to initialize this attribute with the correct value if the device has either the ARP_AV or ARP_AR flag set by default (refer to Section 4.2) having fixed or persistent address. The same principle applies for Alert signal usage.

4.1.11 OpMode

Internal value, here the stack among else stored the value of direction option for the current command. It uses values defined in *cmnd_query (refer to Section 4.3.2)*.

4.1.12 SRByte

Simple response byte preset. This value is used as reply in case the Receive byte transaction comes. This value gets its MSb forced to one before reply transmission if flag RCV_BYTE_LMT is set (refer to *Section 4.2*).

4.1.13 Buffer

The common I/O buffer used for the SMBus communication. It is not recommended to access the buffer directly but rather to use the *STACK_SMBUS_GetBuffer (Section 5.5.2)* function call. The buffer size is based on the STACK_NBYTE_SIZE HAL compilation parameter, indicating the maximum amount of user data transferred in the command. The user can modify this constant to balance the capability and the footprint.

4.2 State machine explained

The state machine states are coded by a combination of flags in the StateMachine attribute of the SMBUS_StackHandle. The StateMachine is a 32 bit unsigned integer used as register bitmap of flags.

Reset value is 0x00000000 and is defined as **SMBUS_SMS_NONE.**

In the code, all state machine defines start with SMBUS_SMS_ prefix, unlike here in this section where only the variable part of the identifier is used.

Table 2. StateMachine attribute bits

31	30	29	28	27	26	25	24
Res	ARP_AM	ARP_AR	ARP_AV	Res	PEC_ ACTIVE	RCV_ BYTE_LMT	RCV_ BYTE_OFF
-	r	rw	r	-	w	w	w
23	22	21	20	19	18	17	16
ERR_ PECERR	Res	ERR_BTO	ERR_HTO	ERR_OVR	ERR_ ACKF	ERR_ ARLO	ERR_ BERR
r/reset	-	r/reset	r/reset	r/reset	r/reset	r/reset	r/reset
15	14	13	12	11	10	9	8
Res	Res	QUICK_ CMD_W	QUICK_ CMD_R	Res	Res	ERROR	ZONE_ READ
-	-	r/reset	r/reset	-	-	r/reset	-
7	6	5	4	3	2	1	0
ALERT_ ADDRESS	ALERT_ PENDING	IGNORED	RESPON SE_ READY	PROCESS ING	RECEIVE	TRANSMIT	READY
r/reset	r	r	r	r	r	r	r

Bit 31 Reserved

Bit 30 **ARP_AM**: Address resolution protocol address match (ARP command)

Bit 29 **ARP_AR**: Address resolution protocol address resolved flag processing

Bit 28 **ARP_AV**: Address resolution protocol address valid flag

Bit 27 Reserved

Bit 26 **PEC_ACTIVE**: PEC feature activation flag
It must remain the same during all operations.

Bit 25 **RCV_BYTE_LMT**: All Receive byte slave responses are ORed with 0x80, to prevent stalling the bus.

Bit 24 **RCV_BYTE_OFF**: Any direct read is treated as Quick command read to prevent stalling the bus with attempted data transmission.

Bit 23 **ERR_PECERR**: PEC error - data integrity lost

Bit 22 Reserved

Bit 21 **ERR_BTO**: Bus timeout

Bit 20 **ERR_HTO**. Timeout error

Bit 19 **ERR_OVR**: Overrun/underrun error
It enables SCL stretching to prevent this error.

Bit 18 **ERR_ACKF**: ACK failure error - probably wrong slave address.

Bit 17 **ERR_ARLO**: Arbitration lost error
The device lost the bus. Usually the slave wants to transmit but master stops, frequent with Quick command read.

Bit 16 **ERR_BERR**: Bus error
Misplaced start or stop, try lower-speed.

Bits 15:14 Reserved

Bit 13 **QUICK_CMD_W**: A Quick command write was received (device-specific).
This is a notification flag. The application must reset it after taking appropriate action.

Bit 12 **QUICK_CMD_R**: A Quick command read was received (device-specific).
This is a notification flag. Up to the application to reset it once the command is treated.

Bits 11:10 Reserved

Bit 9 **ERROR**: Any other error.

Bit 8 **ZONE_READ**: Flag set on a slave device to indicate that the status read is not completed.

Bit 7 **ALERT_ADDRESS**: Alert signal treated, address read (host-specific)
This is a notification flag the application must reset each time the stack sets it.

Bit 6 **ALERT_PENDING**: Alert signal detected, address not known yet
Most likely the host retrieves the address automatically soon.

Bit 5 **IGNORED**: Arbitration lost or command is refused by software
The stack ignores the rest of the current command frame. Particularly important for process call and read commands, where there are two address matches.

Bit 4 **RESPONSE_READY**: Slave has a command reply ready for transmission in the I/O buffer.

Bit 3 **PROCESSING**: Processing block flag (variable length transmission)

Bit 2 **RECEIVE**: State of receiving data on the bus

Bit 1 **TRANSMIT**: State of writing data to the bus

Bit 0 **READY**: The stack is listening and ready to process a request.

Note: The HAL driver has its own state machine, working with lower-level and more detailed states during the communication processing. The state machines are loosely related by function calls, interrupt requests and callbacks.

Figure 2. Simplified state transition diagram

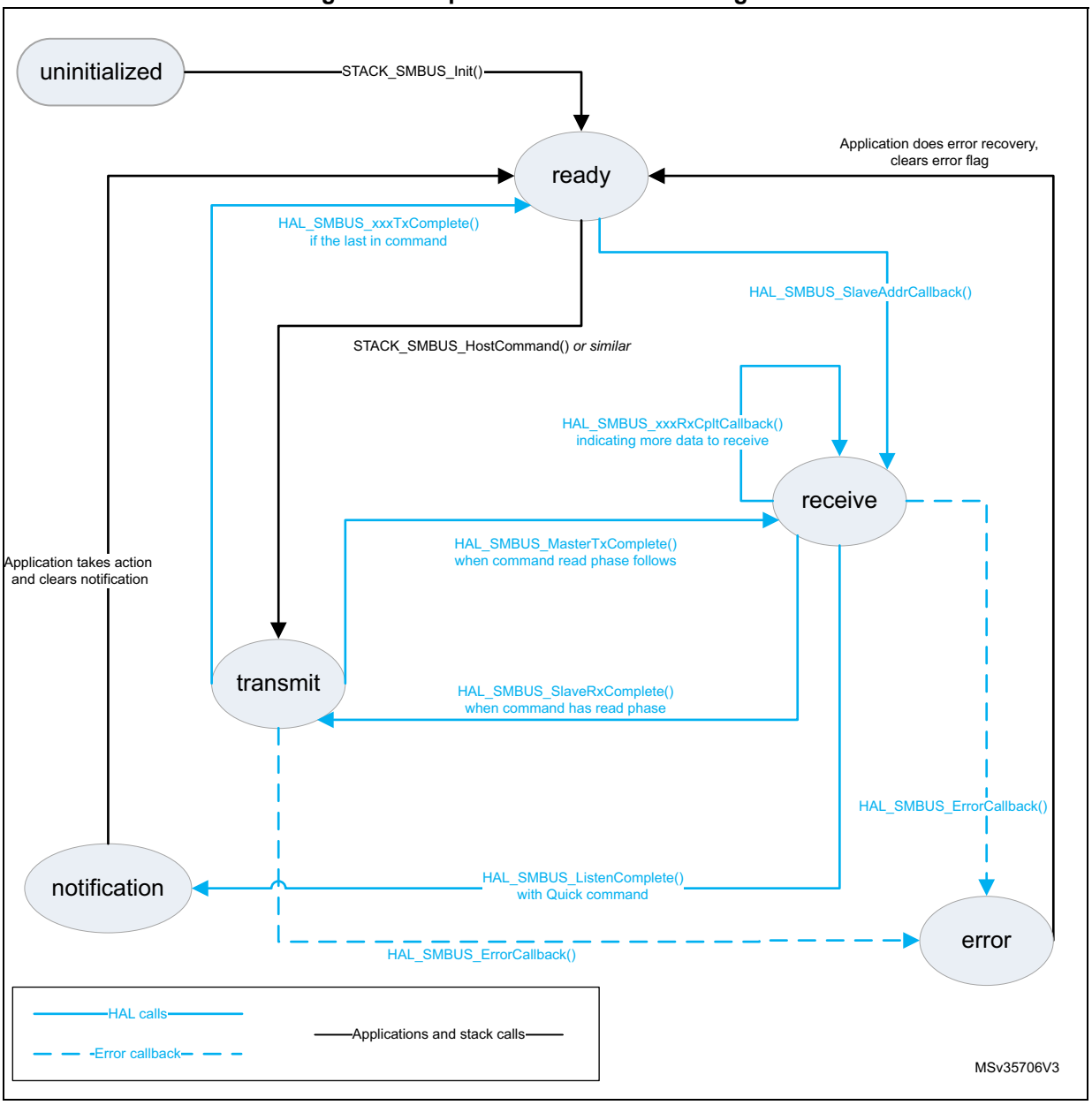


Figure 2 illustrates basic state transitions based on flags in StateMachine (listed in *Table 2*). In this diagram both states error and notification represent flags marked as r/reset (read and reset) in *Table 2*. These flags are set by the stack and the application should reset them after taking appropriate action.

In *Figure 2*, the blue color lines indicate HAL calls, while the black lines represent the application and the stack.

For more detailed flows specific to different transaction types, please see *Section 6*.

4.3 Command descriptor

This section describes the format of items included at the command table (*Section 4.1.5*). These records represent supported commands that are provided as parameter to function STACK_SMBUS_HostCommand (*Section 5.2.1*).

Each record has to follow this exact structure:

```
typedef struct {
  uint8_t cmnd_code;
  uint8_t cmnd_query;
  uint8_t cmnd_master_Tx_size;
  uint8_t cmnd_master_Rx_size;
} st_command;
```

where the attributes roles are described in the next sections.

4.3.1 cmnd_code

This is a command code. It is the first byte written in most transaction types, identifying the command and subsequent data meaning.

4.3.2 cmnd_query

This attribute determines the supported command types. The command code may be used in write/read transactions, in process call, in block transfer and in combinations thereof. It indicates which values the OpMode (*Section 4.1.11*) may assume while processing this particular command.

The allowed values are shown in *Table 3*.

Table 3. Command query modes

Identifier	Value	Description
READ	0x20	Read operation / fixed size read only command
WRITE	0x40	Write operation / fixed size write only command
READ_OR_WRITE	0x60	Command executed either as read or as write of fixed size
BLOCK_READ	0x30	Read block operation / read only command
BLOCK_WRITE	0x50	Write block operation / write only command
BLK_PRC_CALL	0x90	Block process call - write, then read, variable sizes
PROCESS_CALL	0x80	Simple process call - write, then read, fixed sizes
BLK_RD_OR_WR	0x70	Command can be executed either as read or as write of variable size

4.3.3 cmnd_master_Tx_size

Number of bytes transmitted from the host, including the command code byte.

4.3.4 cmnd_master_Rx_size

Number of bytes transmitted in the slave response.

5 SMBus stack API interfaces

This section describes the functions and macros provided for the application layer.

5.1 Callbacks

Callback functions are meant to be implemented in the user application. The stack only implements these functions as weak to support that substitution. The number of callbacks was reduced to keep the application role simple and the execution efficient:

5.1.1 STACK_SMBUS_AlertClbk

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure that contains the configuration information for the specified SMBus (*Section 4*).

No return value.

This callback is specific to the host side. It informs the application that the address has been retrieved from the slave that just recently sent an alert address.

5.1.2 STACK_SMBUS_AddrAccpt

Parameters:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

AddrMatchCode - Address match code, 16-bit unsigned integer.

Returns the ErrorStatus response code.

The stack asks the application whether to accept call with a particular address, intended for more complex devices that serve a range of addresses.

5.1.3 STACK_SMBUS_LocateCommand

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

No return value.

This function is defined as weak, to provide easy means for the application to redefine the way the structured command information is located within the command table. Implementation is also modifiable by compile time switches ARP and DENSE_CMD_TABLE. Slave specific.

5.1.4 STACK_SMBUS_ExecuteCommand

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

Returns the HAL_StatusTypeDef response code.

This call represents the command processing implementation.

This function performs the following actions:

1. Get the command code or flag from the stack handle. If the current command pointer is NULL, this function checks the state machine (*Section 4.2*) for flags indicating the codeless commands (Quick command read and Quick command write).

Note: The Receive byte bus transaction is an exception, the slave command callback is not called for it.

2. Get the input/output buffer pointer (function *STACK_SMBUS_GetBuffer* (*Section 5.5.2*) if necessary.
3. Perform the command action.
4. If applicable, provide the response to the command to the input/output buffer.

5.2 Host specific commands

5.2.1 STACK_SMBUS_HostCommand

Parameters:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

pCommand - pointer to command descriptor of the command to be transmitted, NULL for Quick command.

address - slave address to be used in the transmission, 16-bit unsigned integer.

direction - either READ or WRITE, some commands have the choice with the same ordinal.

Do not use BLOCK_READ and BLOCK_WRITE. It is tied to command definition and implicit. Leave zero for process call, 32 bit unsigned integer type.

Returns the HAL_StatusTypeDef response code.

Call this function with selected parameters to send a command to the bus. This function serves to perform Quick command write, Send byte, Write byte/word, Read byte/word, Process call and all block transfers.

For most commands, it is necessary to first call function *STACK_SMBUS_GetBuffer* (*Section 5.5.2*), and prepare the command output data.

5.2.2 STACK_SMBUS_HostRead

Parameters:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

data - pointer to the 8-bit integer variable where response should be stored.

address - slave address to be used in the transmission, 16-bit unsigned integer.

Returns the HAL_StatusTypeDef response code.

Simplified call with no command code and read-only transmission. This function can be used by the bus master to issue Quick command read or Receive byte.

5.3 Device specific commands

5.3.1 STACK_SMBUS_NotifyHost

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

Returns the HAL_StatusTypeDef response code.

Initiates the host notify protocol by assuming master role and transmitting the command.

5.3.2 STACK_SMBUS_SendAlert

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

No return value.

Initiates the alert response protocol by issuing the alert signal.

Note: The signal itself is only the first step of the whole protocol. For the process to conclude correctly the OwnAddress must be initialized in the stack handle.

5.3.3 STACK_SMBUS_ExtendCommand

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

No return value.

A callback added for users wanting to customize the Extended command processing (see *Section 7.2.2* or protocol specification for more details). Called when received input command code equals the Extend command reserved value.

5.4 ARP specific commands

The address resolution protocol introduces several irregularities to the SMBus functionality, namely command processing. For the ARP to work correctly, the device must be configured as discoverable (DEV_DIS build option defined. For further information, refer to *Section 9.2*). To process the ARP commands correctly, following functions are introduced:

5.4.1 STACK_SMBUS_LocateCommandARP

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

No return value.

This is a subroutine of the *STACK_SMBUS_LocateCommand* (*Section 5.1.3*) called in case of ARP (device addressed on ARP default address). It bypasses the search for command in the main command table and identifies the ARP applicable command.

5.4.2 STACK_SMBUS_ExecuteCommandARP

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

Returns the HAL_StatusTypeDef response code, zero means success.

This function is called instead of the regular *STACK_SMBUS_ExecuteCommand* (*Section 5.1.4*) in case of ARP (device addressed on ARP default address). It directly implements the actions needed to finish these commands.

5.5 Common functions

5.5.1 STACK_SMBUS_Init

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

Returns the HAL_StatusTypeDef response code, zero means success.

Initializing the SMBus stack. It can only be done successfully after the HAL initialization finished.

5.5.2 STACK_SMBUS_GetBuffer

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

Returns pointer to the stack I/O buffer, array of bytes.

Function called to retrieve the input/output buffer pointer.

Note: Though the pointer remains the same value, it is still recommended to keep calling this function each time to make sure the buffer is available for use.

5.6 Useful macros defined by the stack

5.6.1 STACK_SMBUS_IsReady

Return value is 1 in case the stack is ready to process a new command (not busy), 0 otherwise.

5.6.2 STACK_SMBUS_IsError

Return value is 1 if serious error occurred on the SMBus, or else 0.

5.6.3 STACK_SMBUS_IsBusy

Return value is 1 if a communication is ongoing, or 0 in case it is not.

5.6.4 STACK_SMBUS_IsAlert

Return value is 1 if the host stack has the address of the slave that asserted alert signal obtained, otherwise it is 0.

5.7 Private functions

5.7.1 STACK_SMBUS_ResolveContext

Parameter:

hsmbus - Pointer to a SMBUS_HandleTypeDef structure (from HAL) that contains the configuration information for the specified SMBus.

Returns pointer to the stack SMBUS_StackHandle structure.

Used by the stack to link the stack handle with physical device, useful for applications with multiple SMBus interfaces.

5.7.2 STACK_SMBUS_ReadyIfNoAlert

Parameter:

pStackContext - Pointer to a SMBUS_StackHandle structure (*Section 4*).

No return value.

The function used to automate conclusion of a transaction on the host side.

6 SMBus operation overview

This section describes the typical flow of calls and events at application, stack, and HAL levels when executing commands for both host side and device side of the SMBus stack.

The key differences between commands are highlighted using bold type in the following figures.

6.1 Host side

A properly initialized SMBus stack is only conveying commands, not taking any action by itself. The exception to this rule is the alert signal and the host notify protocol functionality. Each time the stack detects an alert signal assert, it automatically retrieves the slave address of the device that issued the alert. When the previous action is successful, the master stack notifies the application using a callback function. In a similar manner, there is a callback for the host notify.

On the master device, the application calls a stack API function to initiate the command sent to the slave. Sending a command consists of the following five simple steps:

1. Retrieve the I/O buffer pointer from the stack by calling *STACK_SMBUS_GetBuffer* (*Section 5.5.2*). The stack checks its readiness before returning the pointer.
2. Prepare the command output data, if any. This is excluding the command code. Only write to the appointed buffer one byte for a Send byte command, or two bytes for a Send word. There is no check done by the stack on completeness or integrity of the message in buffer. Only exception is a forced hardcap on amount of data in block transfer to STACK_NBYTE_SIZE (defined at compile time).
3. Call the *STACK_SMBUS_HostCommand* (*Section 5.2.1*) or *STACK_SMBUS_HostRead* (*Section 5.2.2*) to initiate the SMBus communication.
4. Poll the stack status for command completion using *STACK_SMBUS_IsBusy* (*Section 5.6.3*) or *STACK_SMBUS_IsReady* (*Section 5.6.1*) macros. Make sure no errors were reported by the stack (*STACK_SMBUS_IsError* (*Section 5.6.2*)).
5. Optionally read the slave response data from the I/O buffer.

For more details about a particular transaction type processing, refer to the sequence diagrams in the next sections.

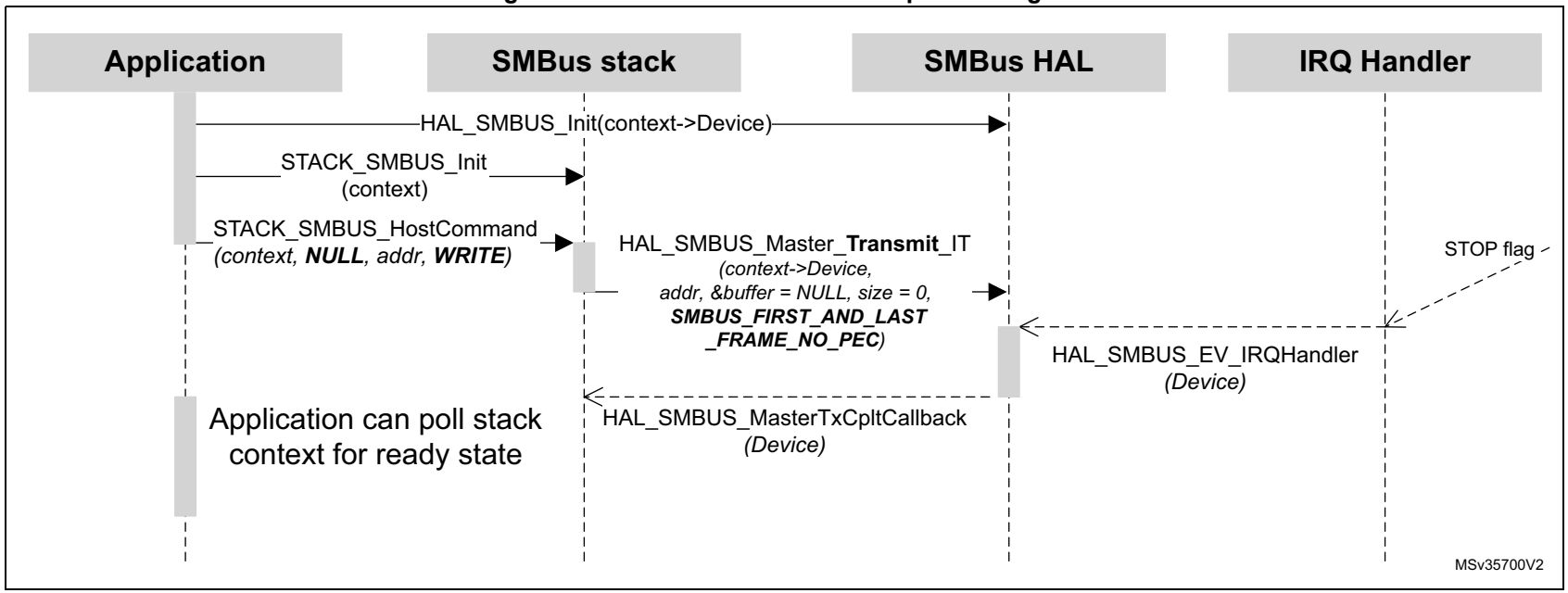
6.1.1 Quick command

The Quick command is used generally for small devices that have limited support of SMBus.

The Rd/Wr bit in the frame may be used to turn on or off a device function, or enable/disable a low-power Standby mode.

Quick command write

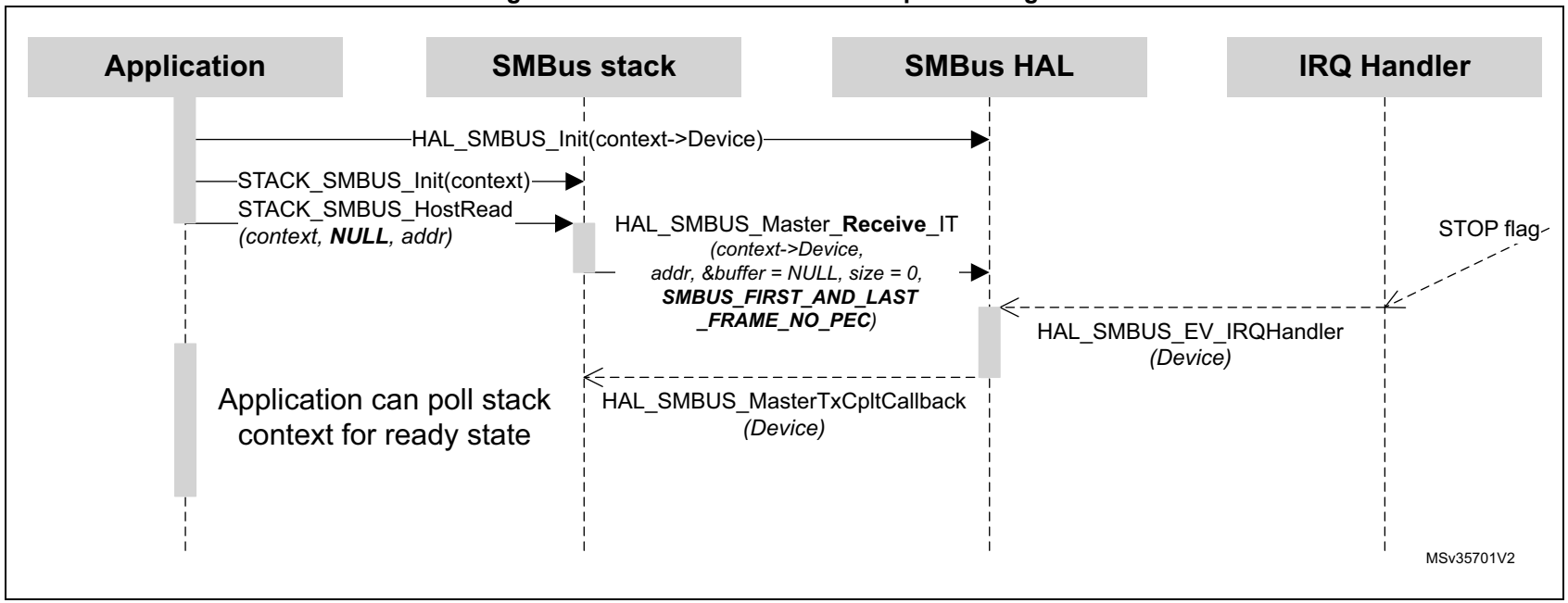
Figure 3. Quick command write - sequence diagram



The Quick command read has no record in the command table. It is sent by calling *STACK_SMBUS_HostCommand* with NULL as *command descriptor parameter.

Quick command read

Figure 4. Quick command read - sequence diagram



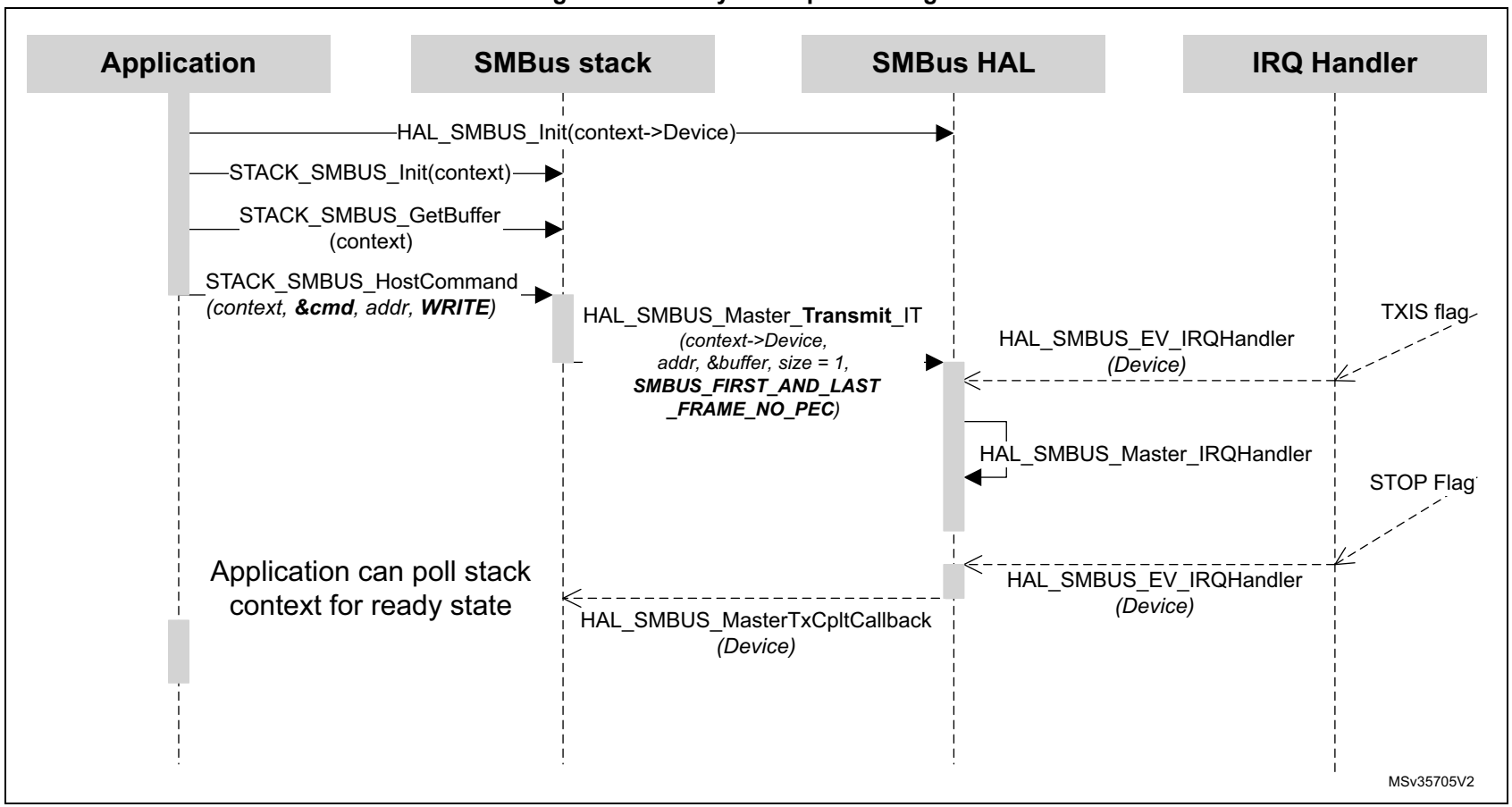
The Quick command read has no record in the command table. It is sent by calling *STACK_SMBUS_HostRead* with NULL as *data parameter.

6.1.2 Send byte

A simple device may recognize its own slave address and accept up to 256 possible encoded commands in the frame.

All or part of the data byte contribute to a “command code”.

Figure 5. Send byte - sequence diagram



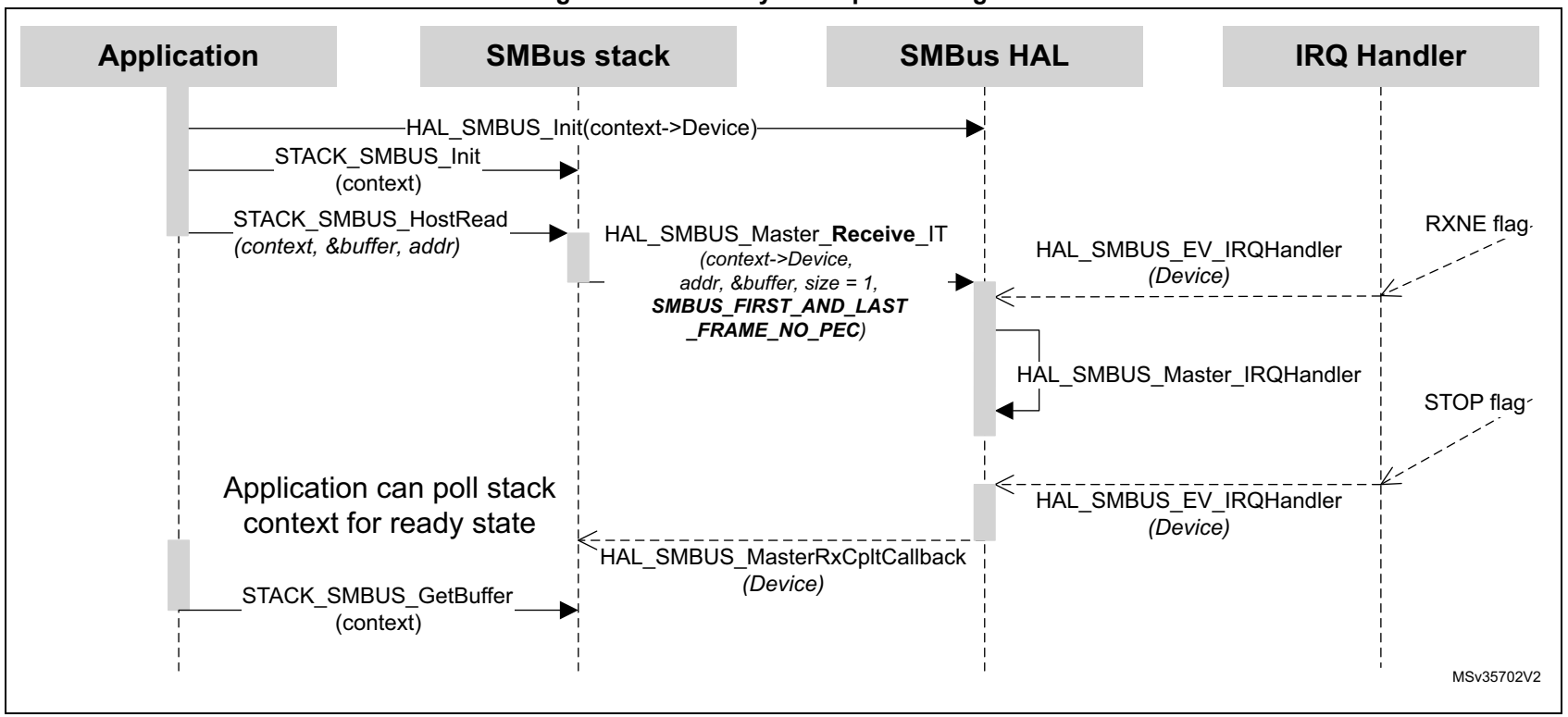
The Send byte command descriptor has WRITE for cmnd_query and 1 for cmnd_master_Tx_size.

6.1.3 Receive byte

The Receive byte is similar to a Send byte. The only difference is the direction of transfer.

A “NACK” (1 in the Ack bit) means the end of transfer.

Figure 6. Receive byte - sequence diagram



The Receive byte has no record in the command table, due to the absence of command code identifier. It is sent by calling *STACK_SMBUS_HostRead* with valid pointer as *data parameter.

6.1.4 Receive byte with PEC

According to the PMBus specification *[2]*, the host must use PEC at any transaction at the end of each message transfer.

Each protocol can use PEC except for Quick Command. It is forbidden to change the PEC flag value in the SMBus stack StateMachine during an ongoing communication.

Figure 7. Receive byte with PEC OK - sequence diagram

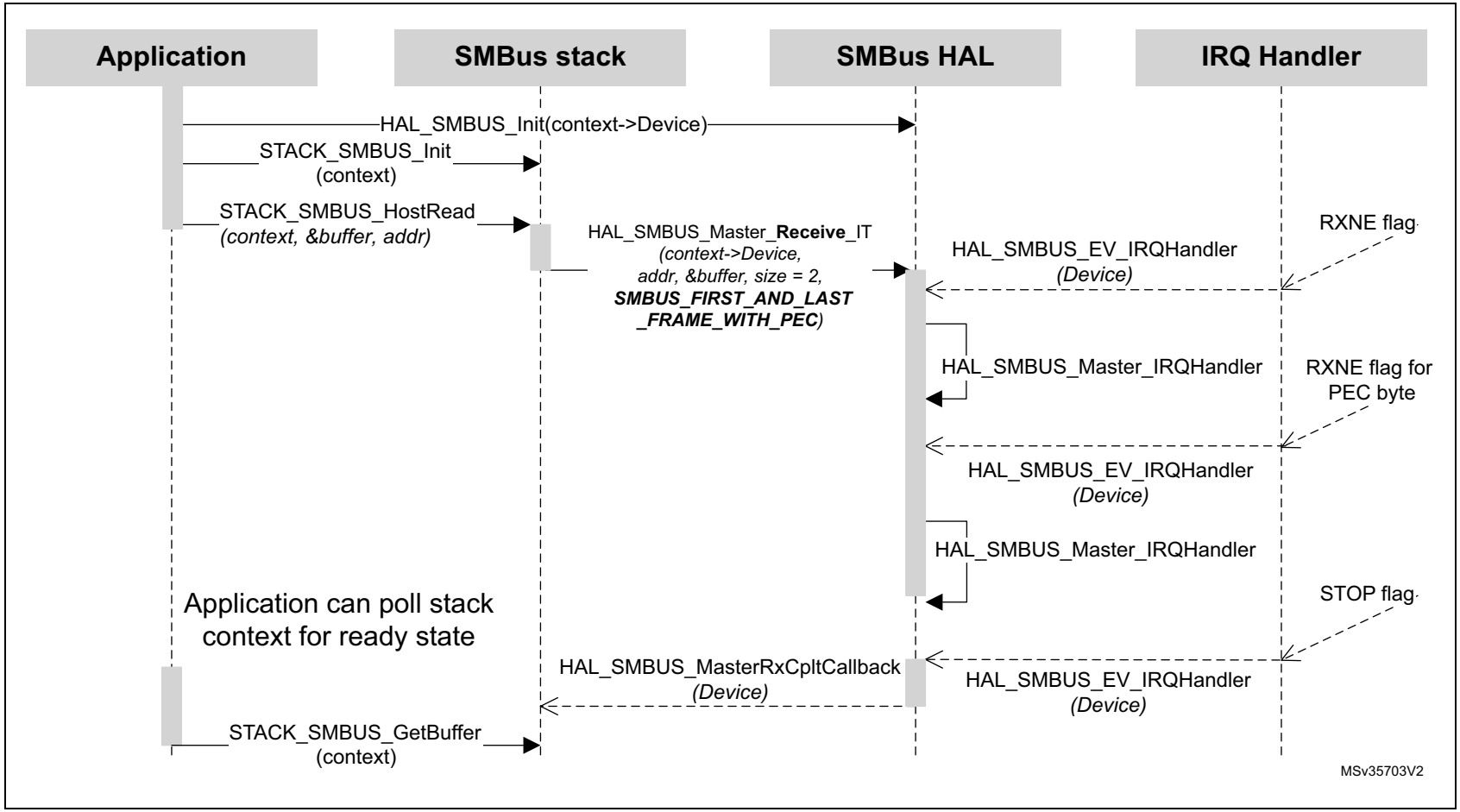
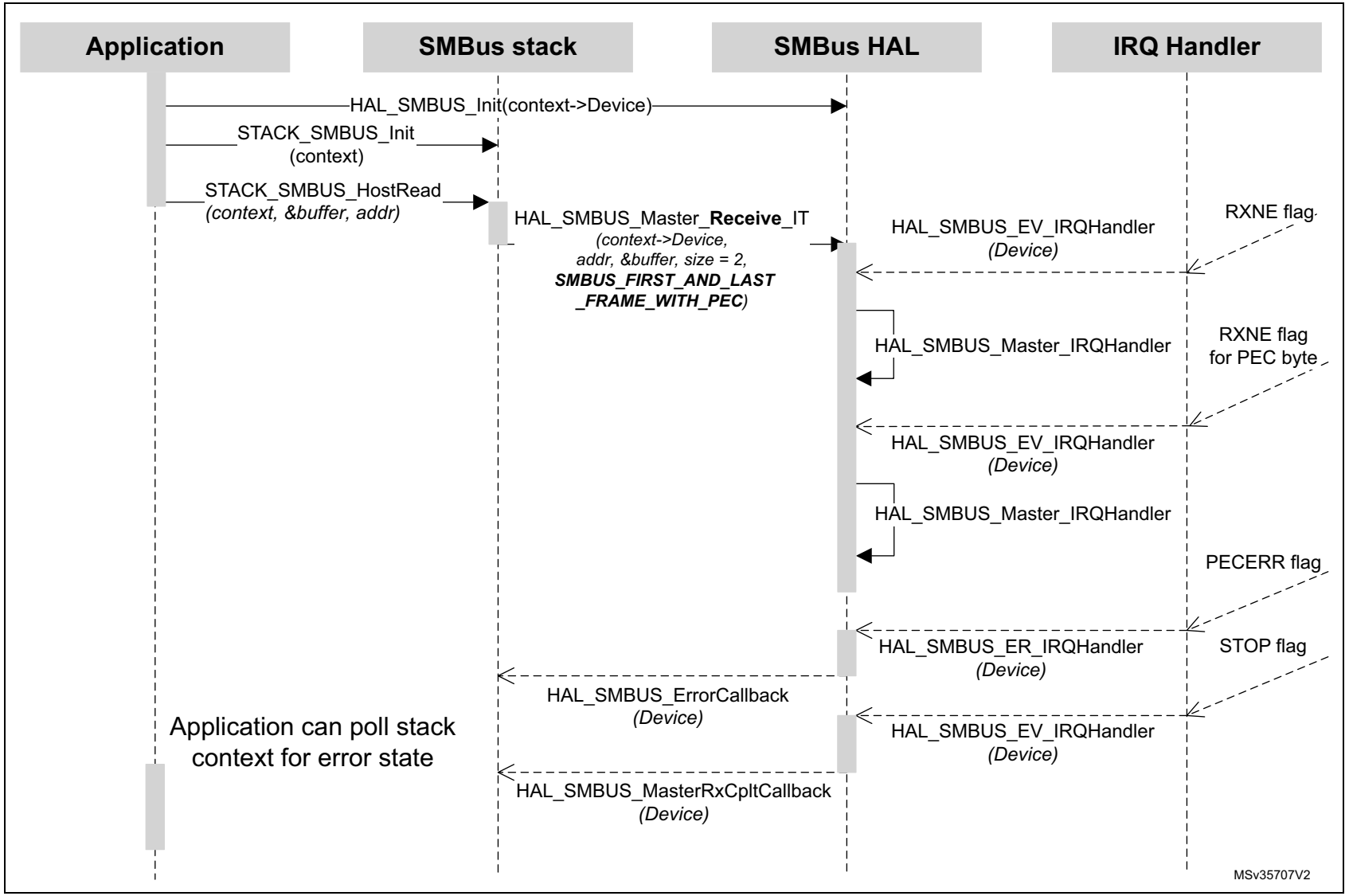


Figure 8. Receive byte with PEC error - sequence diagram



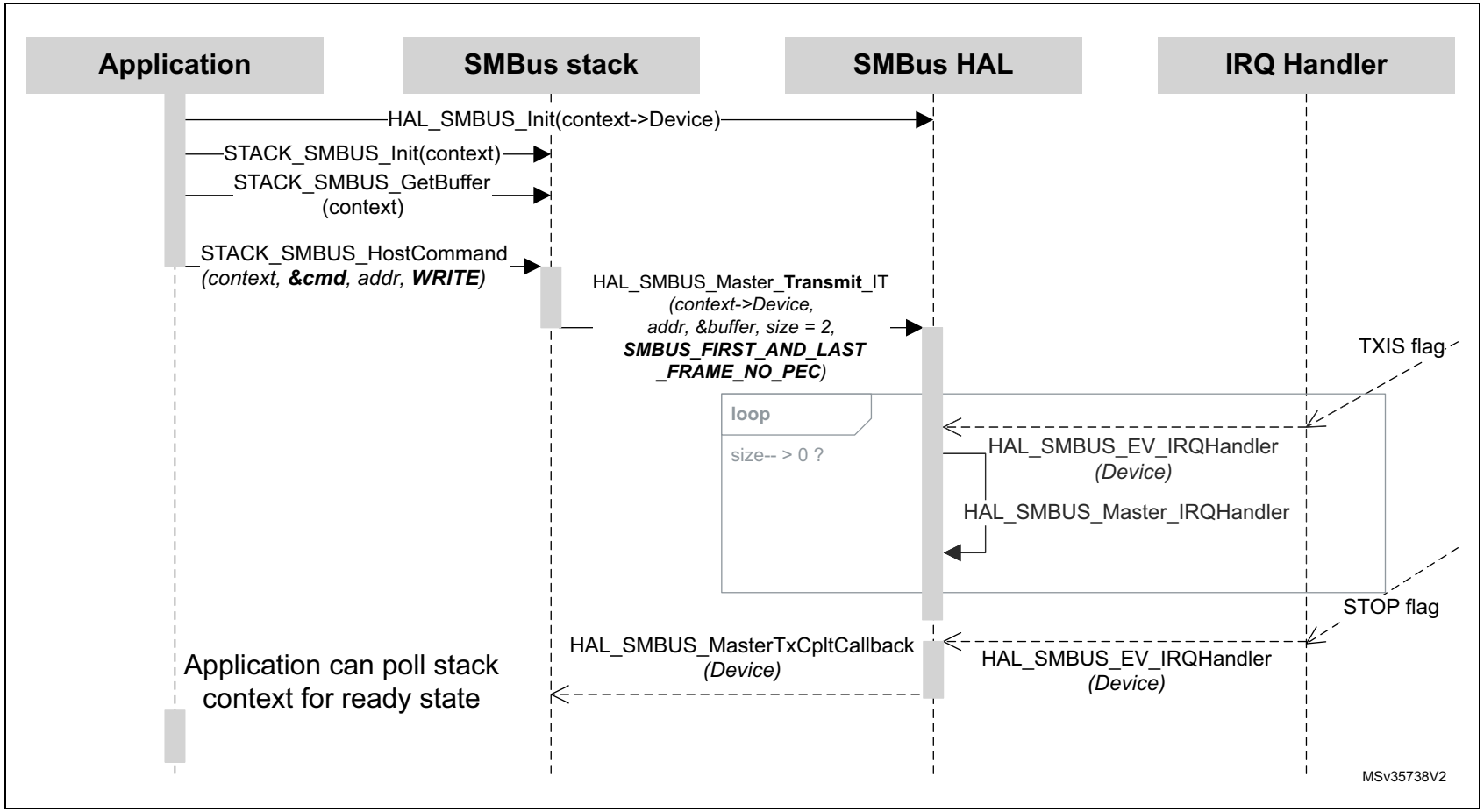
The Receive byte has no record in the command table, due to the absence of command code identifier. It is sent by calling *STACK_SMBUS_HostRead* with valid pointer as *data parameter.

6.1.5 Write byte

The first byte of a Write byte is the command code.

The next byte is the data to be written.

Figure 9. Write byte - sequence diagram

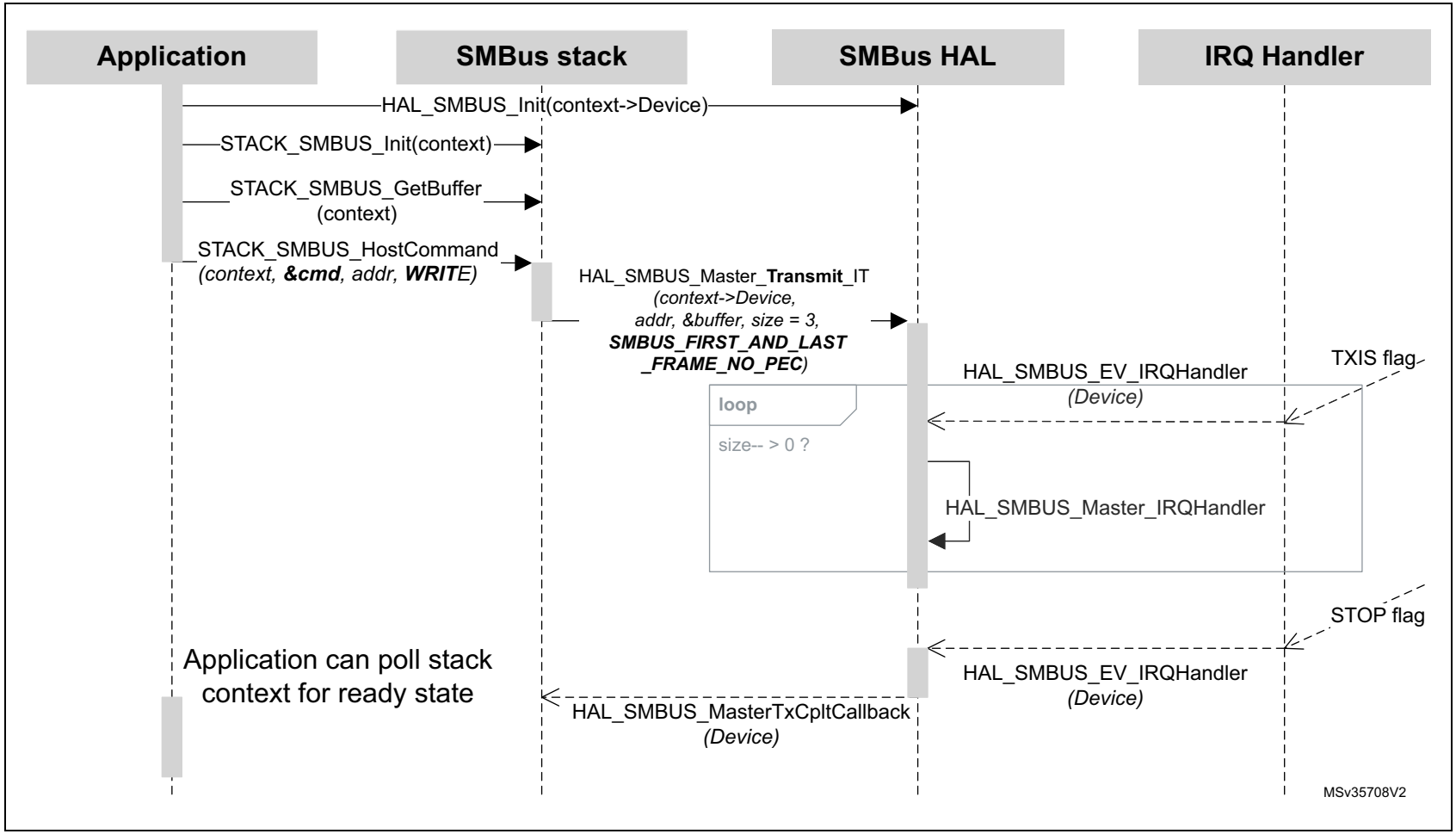


The Write byte command descriptor has WRITE for cmnd_query and 2 for cmnd_master_Tx_size.

6.1.6 Write word, Write 32 and Write 64

The first byte of a Write word is the command code. The next two bytes (alternatively 4 or 8 in the case of SMBus 3.0 additional protocols) are the data to be written

Figure 10. Write word - sequence diagram



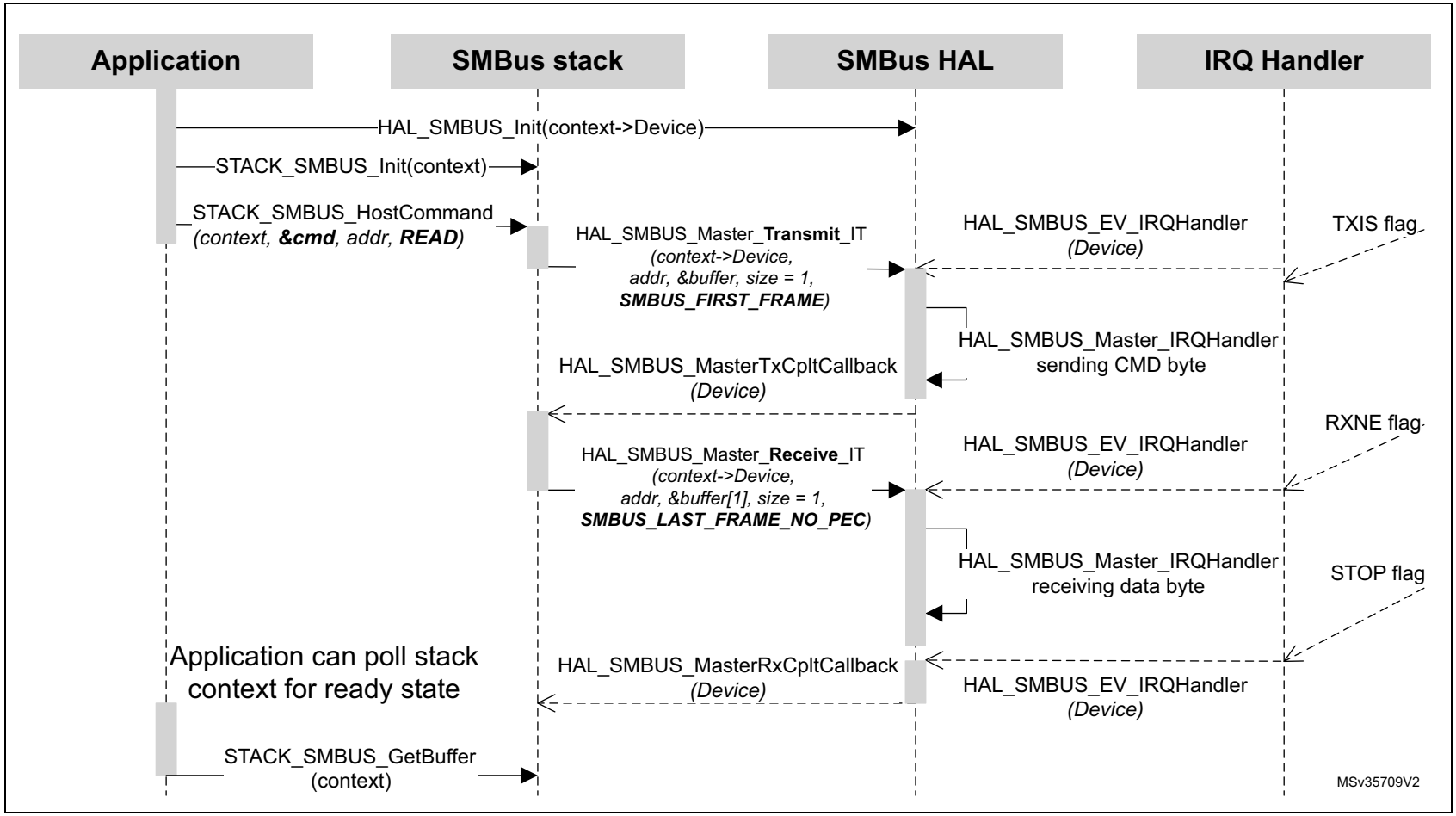
The Write word command descriptor has WRITE for cmnd_query and 3 for cmnd_master_Tx_size. Write 32 has cmnd_master_Tx_size of 5. In the case of Write 64, it is equal to 9.

6.1.7 Read byte

The first byte of a Read byte is the command code.

The next one is the read data. Between these two parts, a repeated start condition is necessary.

Figure 11. Read byte - sequence diagram

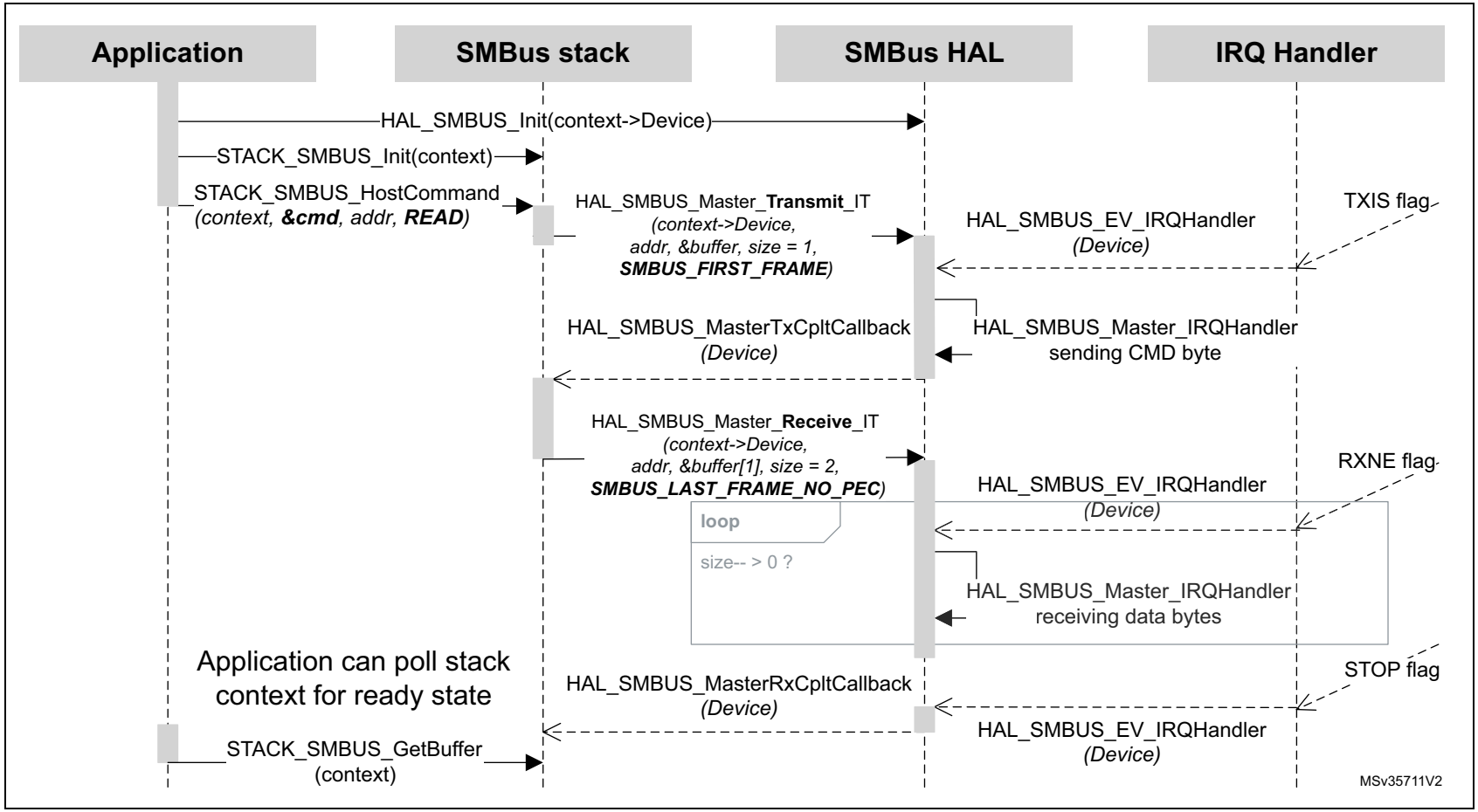


The Read byte command descriptor has READ for cmnd_query and value 1 for both cmnd_master_Tx_size and cmnd_master_Rx_size.

6.1.8 Read word, Read 32 and Read 64

The first byte of a Read word is the command code. The next two bytes (alternatively 4 or 8 in the case of SMBus 3.0 additional protocols) are the read data. Between these two parts, a repeated start condition is necessary.

Figure 12. Read word - sequence diagram

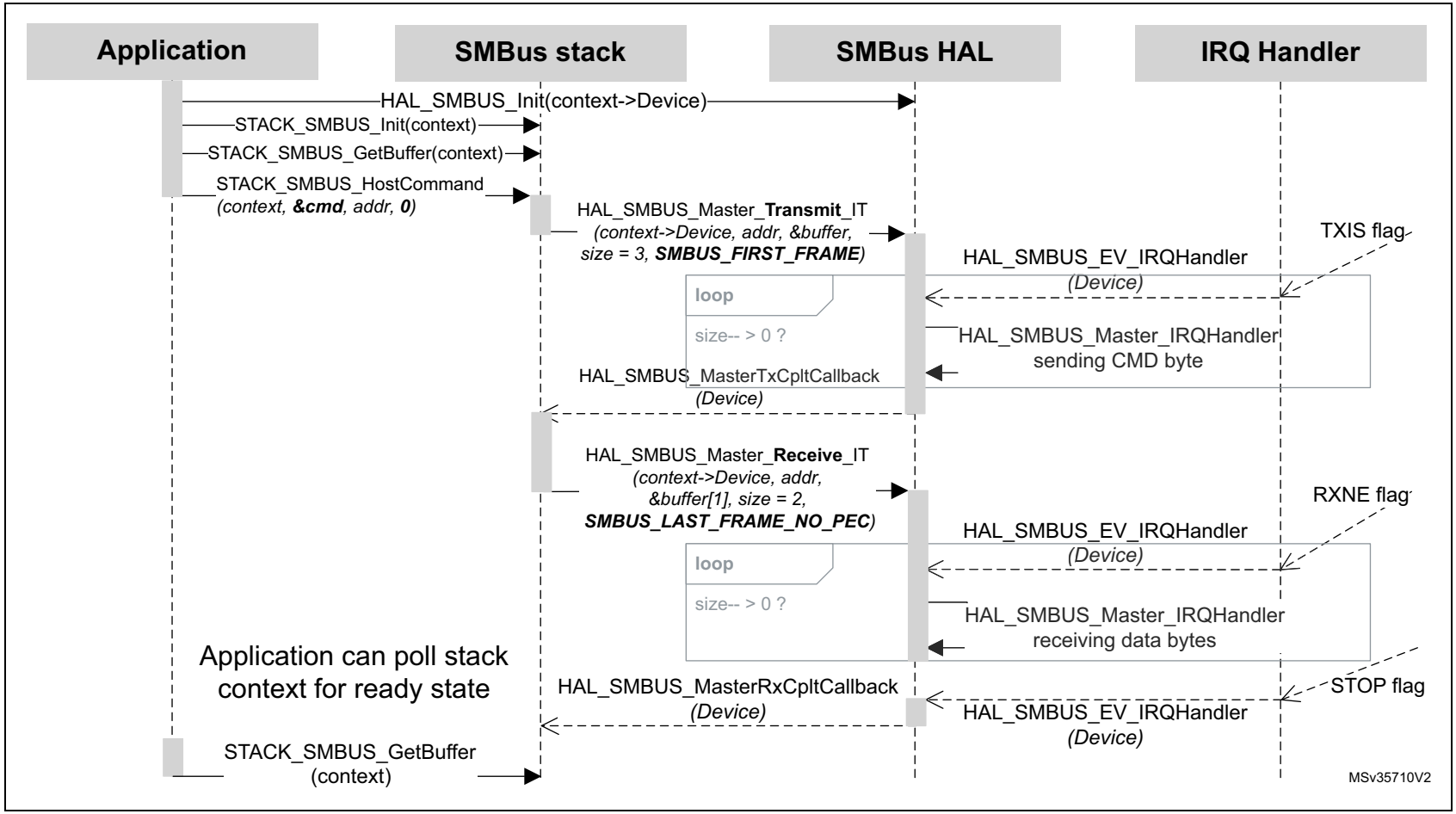


The Read word command descriptor has READ for cmnd_query, 1 for cmnd_master_Tx_size and 2 for cmnd_master_Rx_size. Write 32 has cmnd_master_Rx_size of 4. In the case of Write 64, it is equal to 8. The rest remains the same.

6.1.9 Process call

The process call is named in this way because a command sends data and waits for the slave to return a value dependent of that data. This process is a Write Word followed by a Read Word with a repeated start condition but without the command code.

Figure 13. Process call - sequence diagram

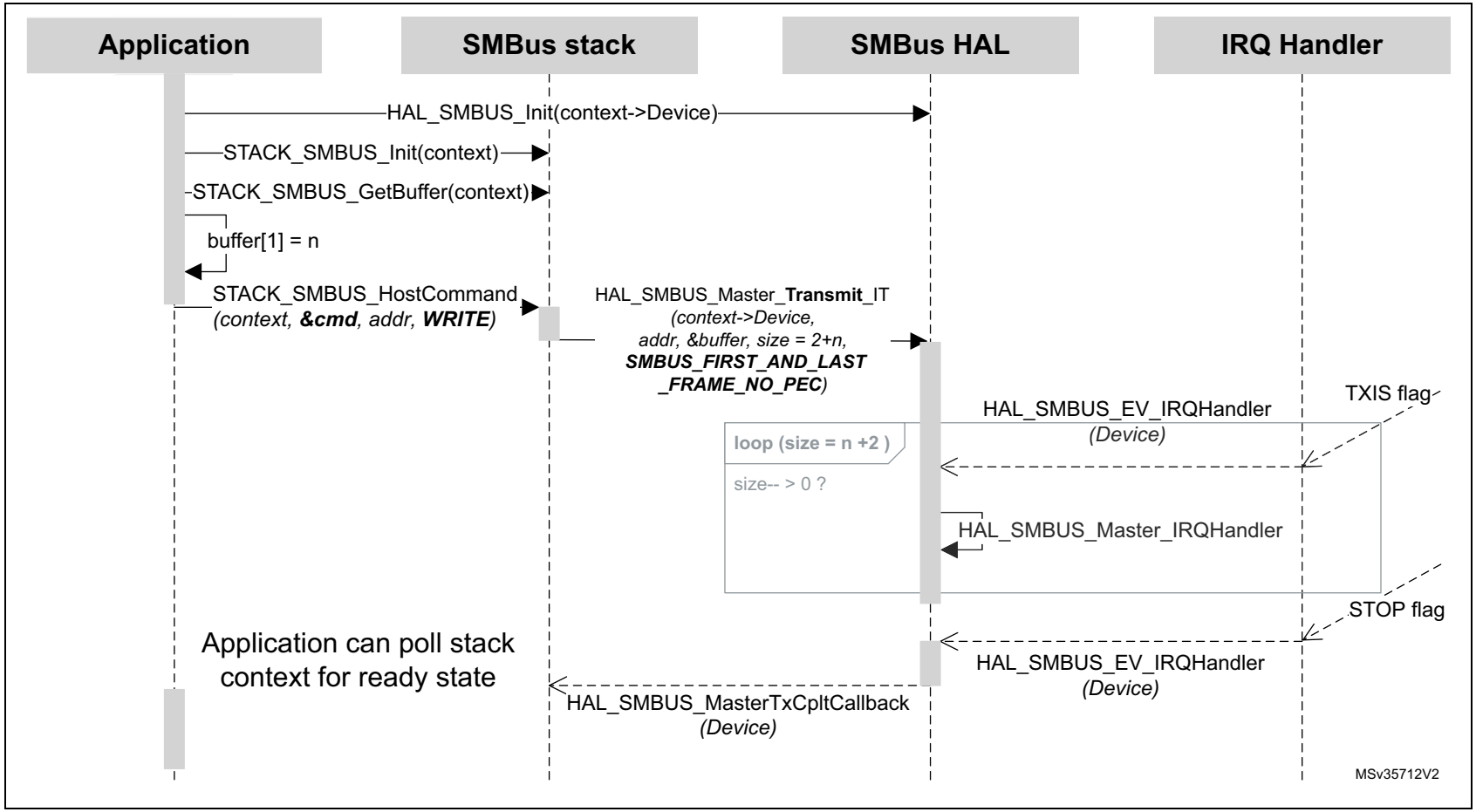


The process call command descriptor has PROCESS_CALL (0x80) for cmnd_query and then usually 3 for cmnd_master_Tx_size and 2 for cmnd_master_Rx_size. The sizes may be different.

6.1.10 Block write

The first byte of a Block write is the command code. It is followed by a byte count that determines how many more bytes follow the message.

Figure 14. Block write - sequence diagram



The block write command descriptor has BLOCK_WRITE for the cmd_query and the typical size (at least 3) for the cmnd_master_Tx_size. The cmnd_master_Rx_size remains 0.

6.1.11 Block read

The first byte of a Block read is the command code. It is followed by a repeated start condition, then by the byte count, which describes how many bytes follow the message from the slave.

Figure 15. Block read - sequence diagram - Part_1

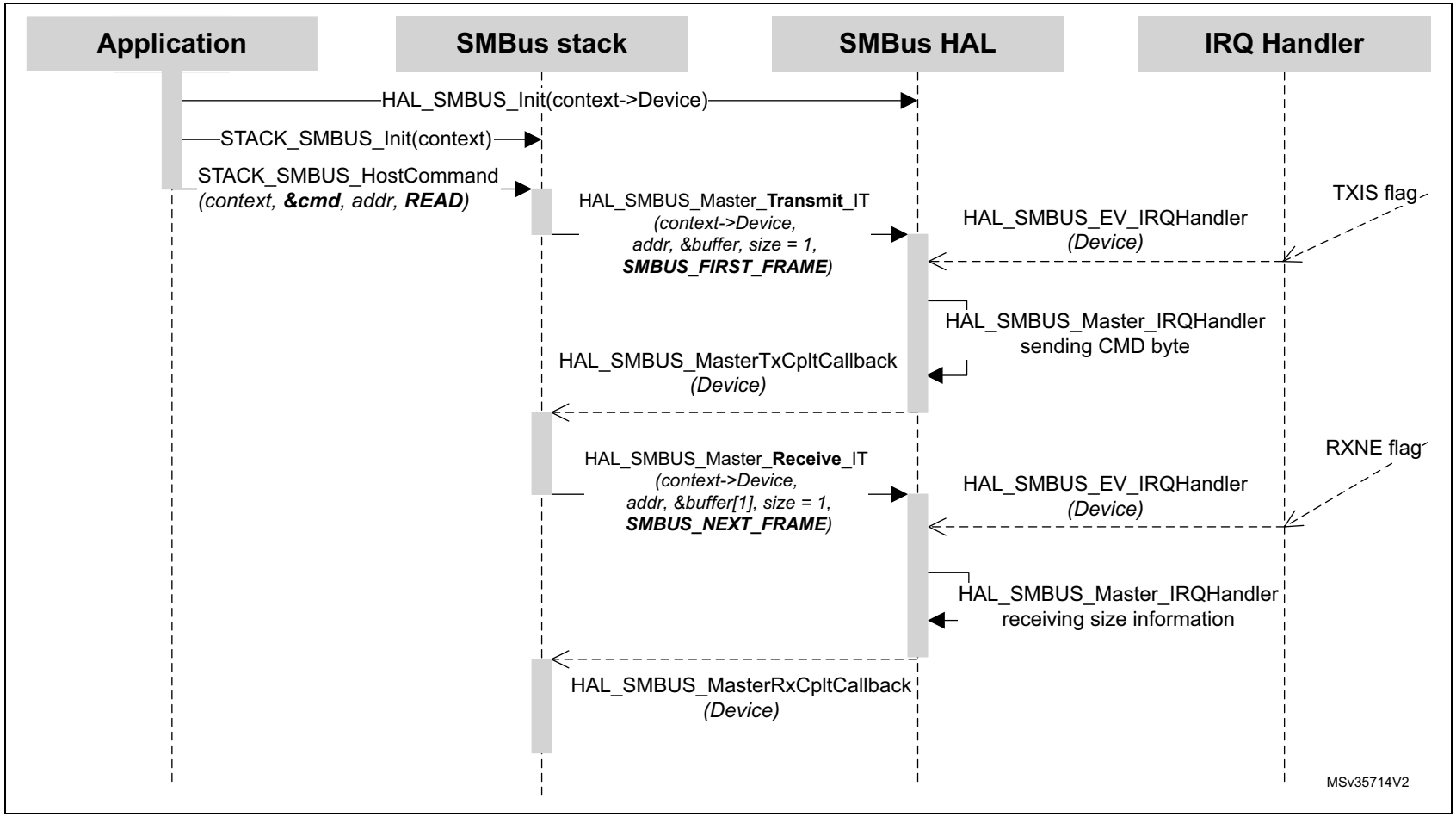
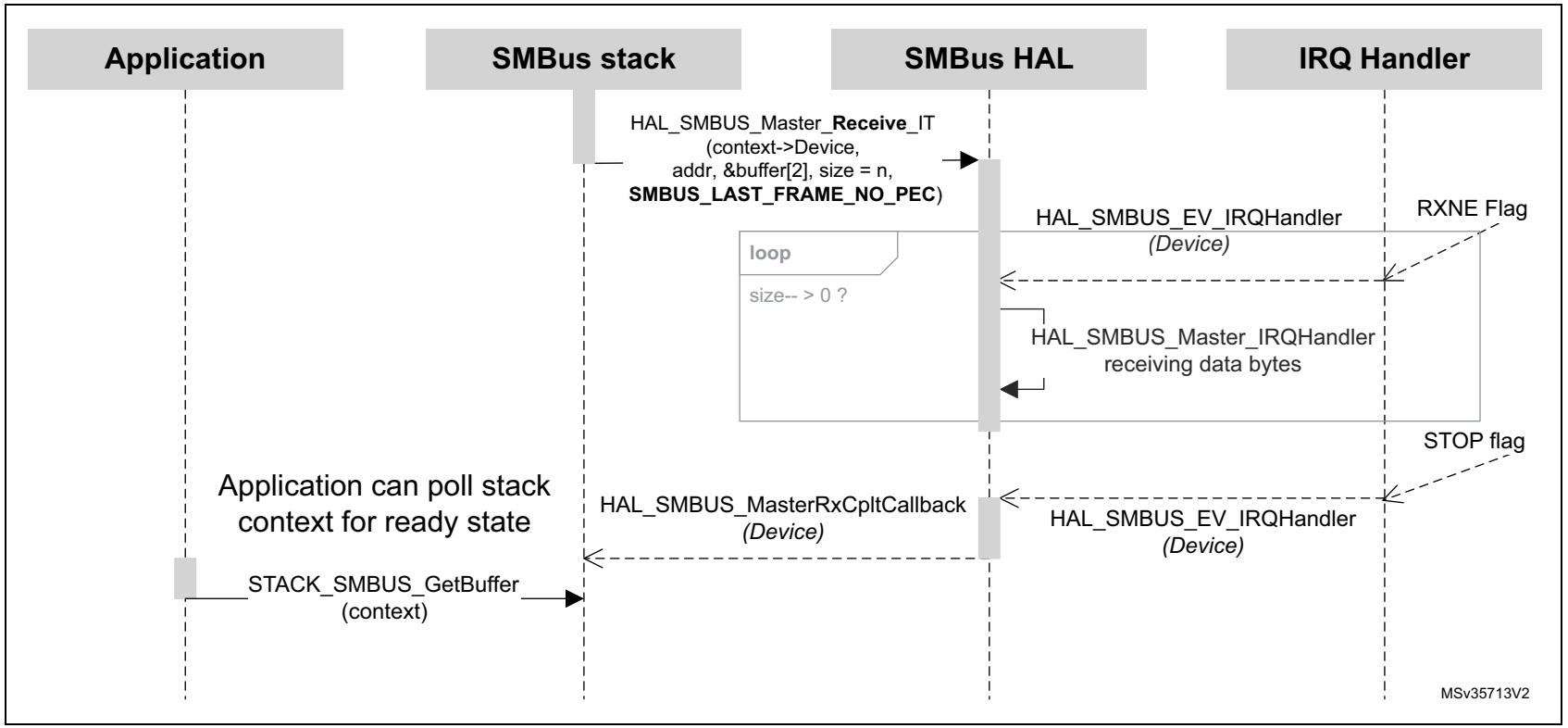


Figure 16. Block read - sequence diagram - Part_2



The Block read command descriptor has BLOCK_READ for the cmd_query and value 1 (the command code size) for the cmnd_master_Tx_size. The cmnd_master_Rx_size must be at least 2.

6.1.12 Block write – Block read process call

This process is a Block write followed by a Block read with a repeated start condition but sharing a common frame with the command code.

Figure 17. Block write - Block read process call - sequence diagram - Part_1

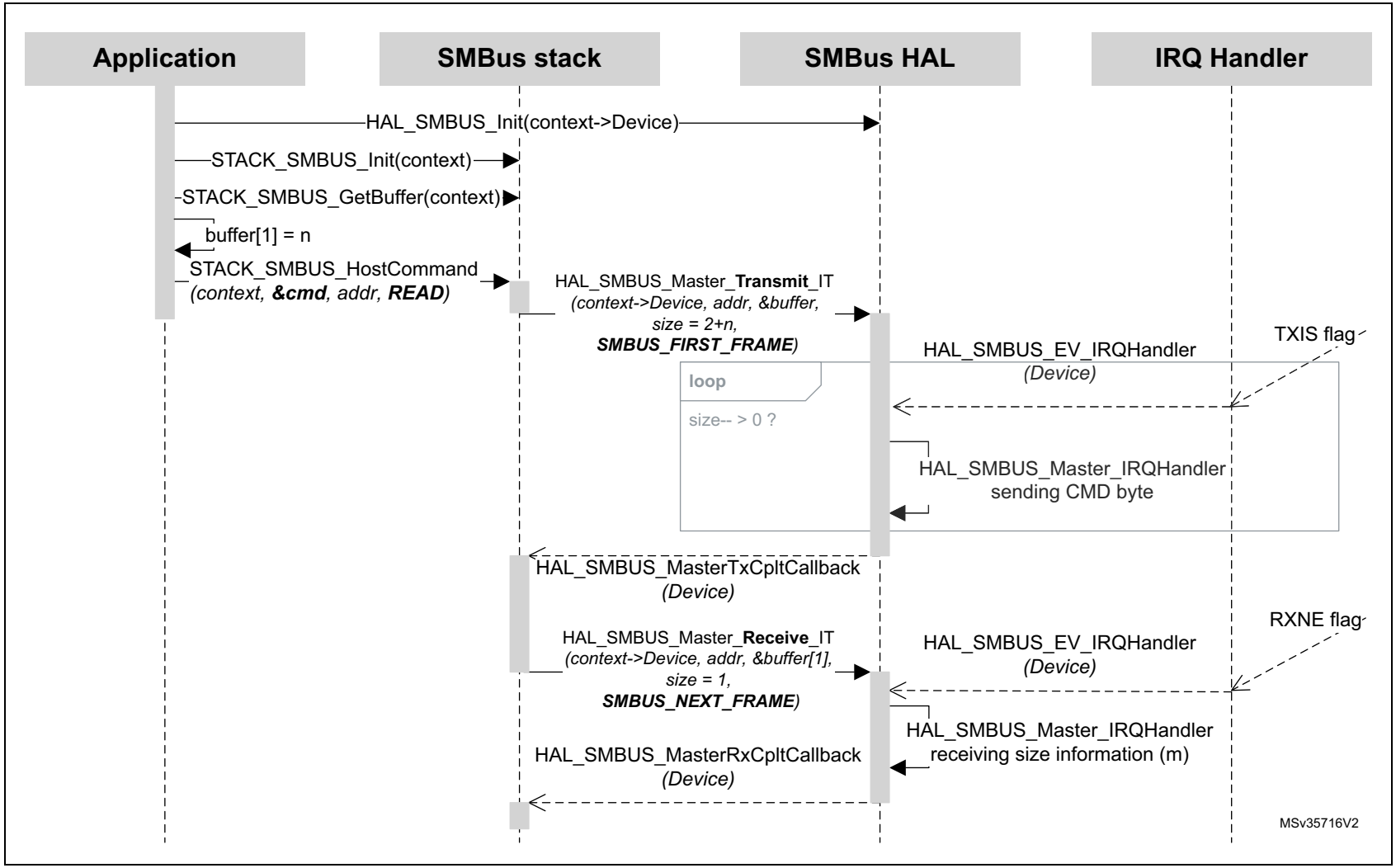
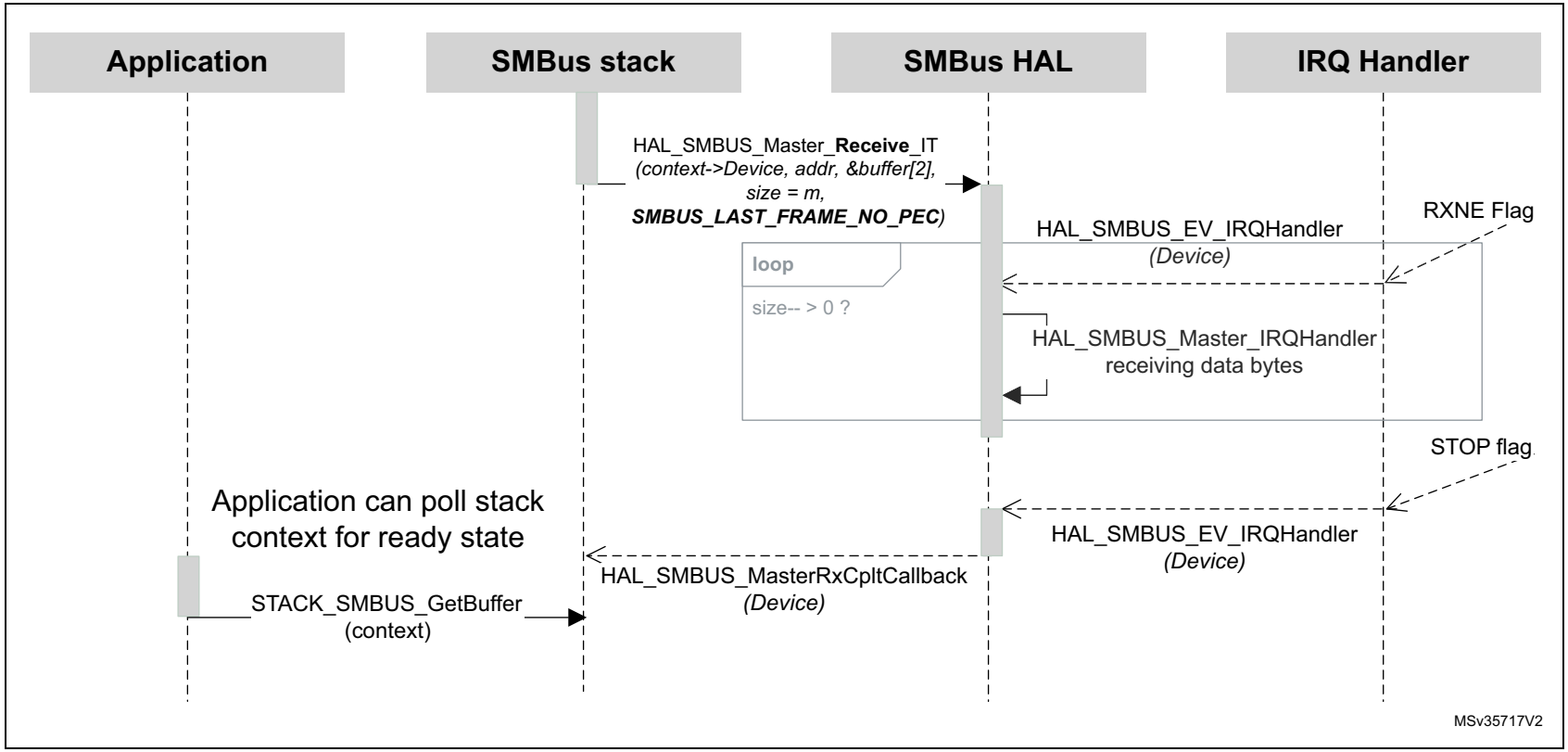


Figure 18. Block write - Block read process call - sequence diagram - Part_2



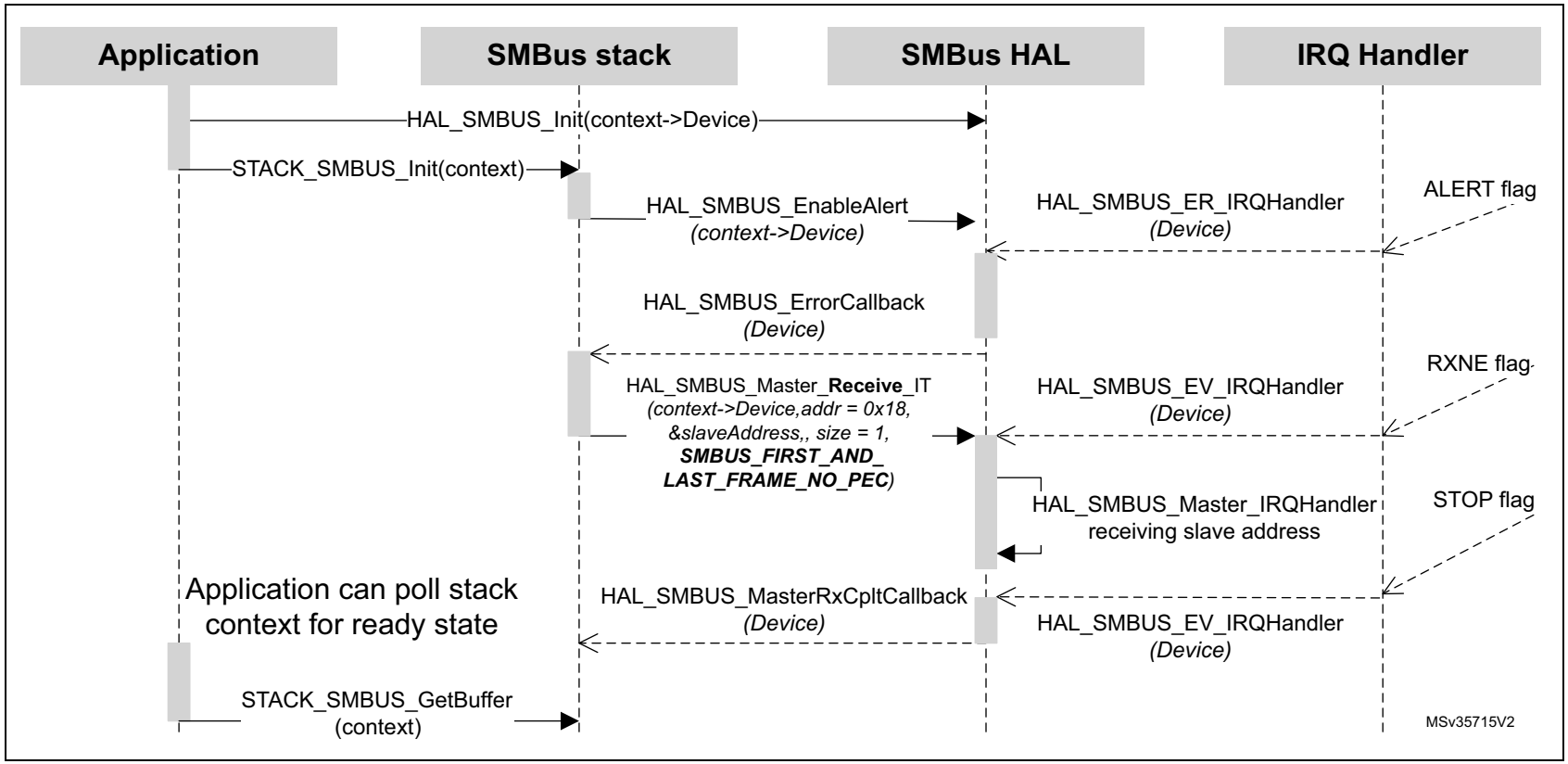
The Block write - Block read command descriptor has BLK_PRC_CALL for the cmd_query and value of at least 3 for the cmnd_master_Tx_size. The cmnd_master_Rx_size must be at least 2.

6.1.13 Alert response

The SMBALERT is a third wire and a signal besides the SMBCLK and the SMBDAT.

A slave device can signal the host through SMBALERT that it wants to talk. The host processes the interrupt and accesses all SMBALERT devices through the alert response address (ARA = 0x18).

Figure 19. SMBus alert treatment, master side - sequence diagram



The Alert response is sent by the stack automatically. Once the address of the signaling device is identified, it is stored in the AlertAddress, and the stack handle state is updated with a flag notifying about the alert processed - SMBUS_SMS_AlertAddress.

6.2 Device side

After initialization, the slave stack sets the driver to listen mode, awaiting for an incoming command. Almost all interactions with the application layer consist of callbacks, where an appropriate action is expected from the application.

These callbacks are put in place when the stack needs the application to make a decision or to provide data. With a typical command processing, the stack calls the three following most important callbacks:

1. *STACK_SMBUS_AddrAccpt* (Section 5.1.2). If the slave device uses only one fixed address, it is not necessary to implement this callback in the user application.
2. *STACK_SMBUS_LocateCommand* (Section 5.1.3). Already implemented in the stack, but the user may replace the default implementation with one simpler or more complex, depending on the application needs.
3. *STACK_SMBUS_ExecuteCommand* (Section 5.1.4). This is the actual command processing. The stack holds the information about the command code and inputs data in its handle. Typically the application grabs that available information and uses a switch-case branching to call a particular command function.

For more details about particular transaction type processing, please refer to the sequence diagrams.

The actions where the slave acts actively is the alert protocol and host notify protocol. To alert the SMBus master, simply call *STACK_SMBUS_SendAlert* (Section 5.3.2). The stack asserts the signal and treats automatically the alert response call from master, returning the OwnAddress configured in the stack handle.

The host notify protocol is initiated by calling *STACK_SMBUS_NotifyHost* (Section 5.3.1) function. The PMBus specifies ([2] and [3]) the contents of the two data bytes of the host notify. It is the responsibility of the application to prepare this data to the output buffer, before calling the function. The stack automatically uses either the OwnAddress attribute (if valid) or sends the ARP version (own device address not assigned yet).

6.2.1 Receive byte/Quick command read confusion

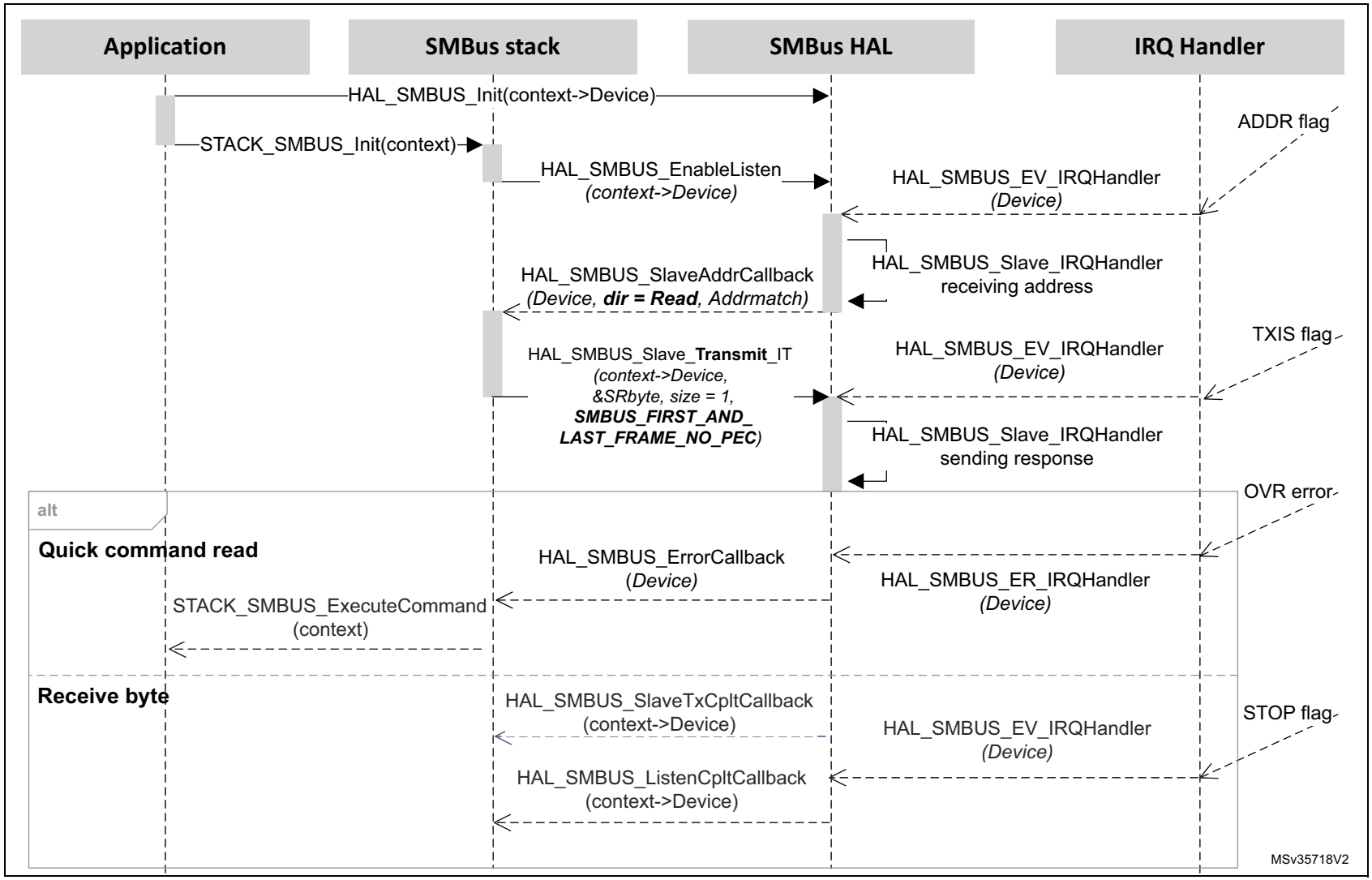
While most transaction types starts with a host writing command code, determining the subsequent action taken, the Quick command read and the Receive byte transactions start with a read frame. It is impossible for the slave to predict whether to expect a STOP condition right after an address (Quick command), or to transmit the prepared byte (Receive byte case). In some cases, this may lead to a situation where the slave device stalls the SDA line by forcing it low, preventing the master to assert a correct STOP condition. This type of situations lead to errors on both sides and to a stack reset after timeout.

There are two flags introduced to the stack specifically to prevent this problem:

- SMBUS_SMS_RCV_BYTE_OFF, causing any direct read being treated as Quick command read. No attempt to transmit the data is done. This flag basically drops the Receive byte from the list of supported SMBus transactions.
- SMBUS_SMS_RCV_BYTE_LMT. This compromised solution forces the MSb of the Receive byte data byte to a logical one. The master may assert a STOP condition to the bus and complete the Quick command.

If none of these solutions are applied in the final product, the Receive byte works without any limitation. Though any received Quick command read causes an error state.

Figure 20. Receive byte processing - sequence diagram

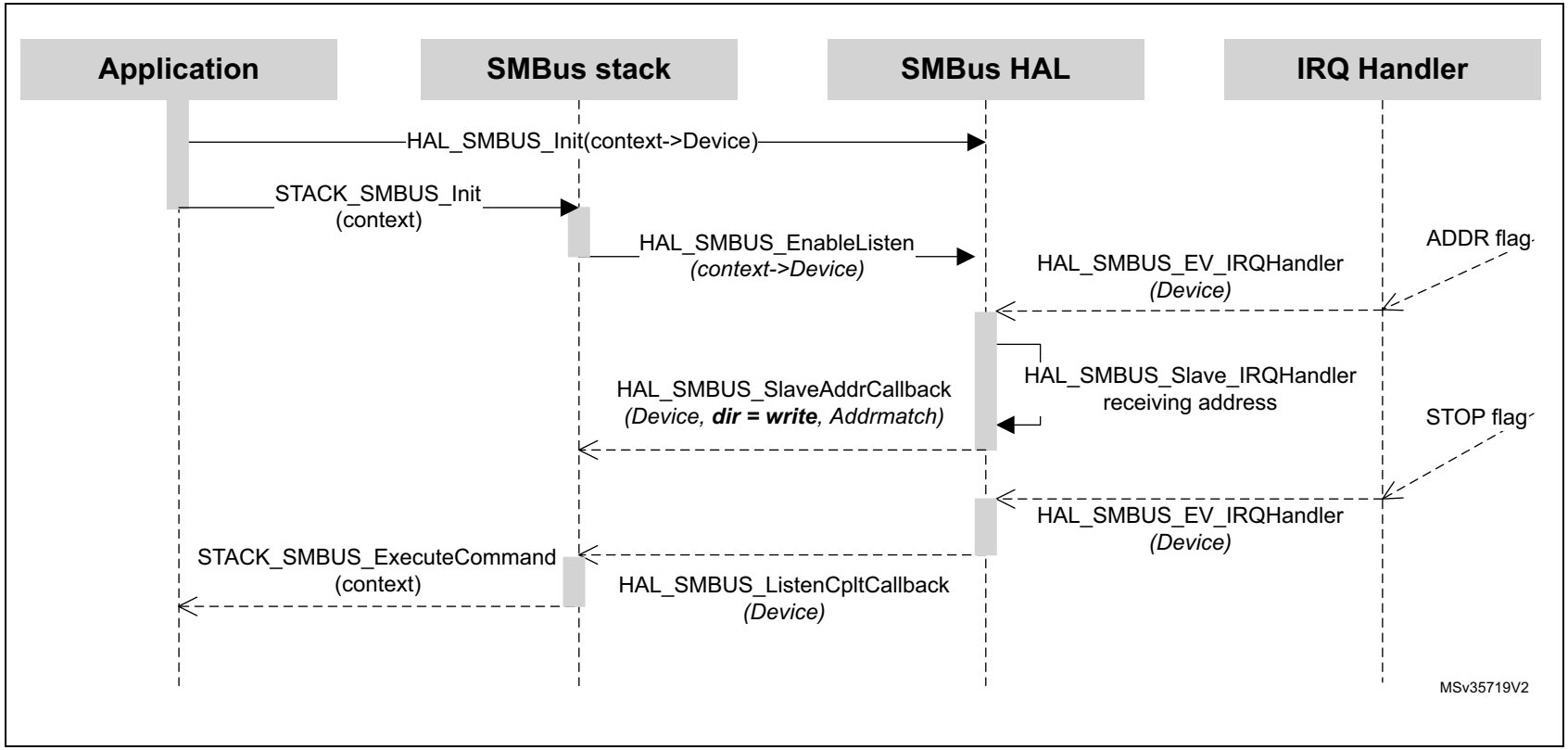


Depending on the frame received, two alternative conclusions are possible. In case of Quick command read, the HAL error code is translated to SMBUS_SMS_QUICK_CMD_R flag in the state machine and STACK_SMBUS_ExecuteCommand callback is executed. In case of Receive byte frame, the stack does not notify the application and handles the communication autonomously.

6.2.2 Quick command write

I n this case the stack attempts to receive additional data, but the STOP condition informs the slave that it was only a Quick command write.

Figure 21. Quick command write (slave side) - sequence diagram

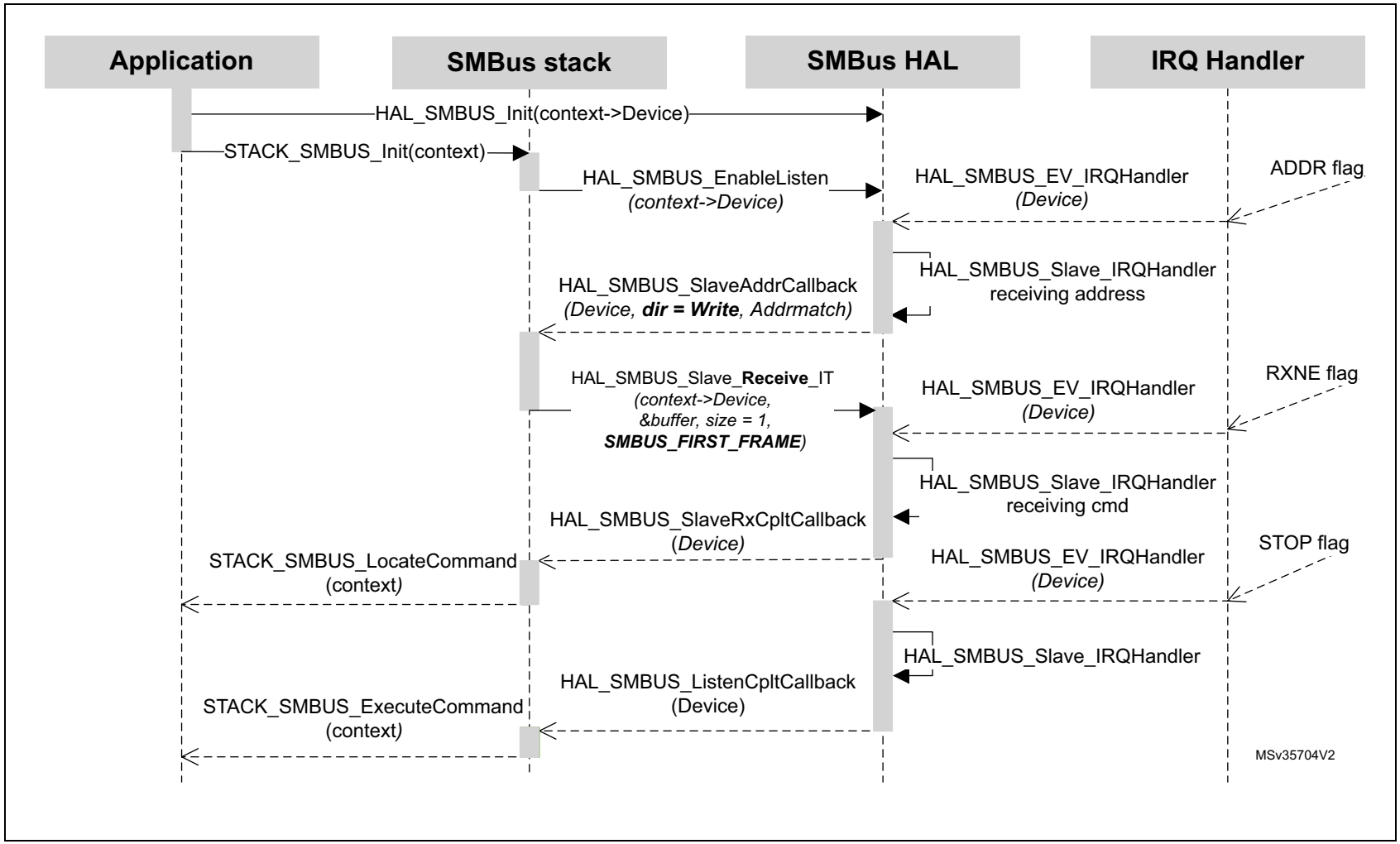


The *STACK_SMBUS_ExecuteCommand* callback finds the CurrentCommand attribute set to NULL, but *SMBUS_SMS_QUICK_CMD_W* is set as *HAL_SMBUS_ListenCpltCallback* finds the stack expecting more data.

6.2.3 Send byte

For a command with no data involved, it seems pointless to separate the *STACK_SMBUS_LocateCommand* and the *STACK_SMBUS_ExecuteCommand*. It is however done, mainly for timing purpose. In case of a PMBus group command, the delay between command code transmission and the actual STOP condition may be substantial.

Figure 22. Send byte (slave side) - sequence diagram



6.2.4 Write byte

The Write byte and Write word are very straightforward regarding the slave side processing.

Figure 23. Write byte (slave side) - sequence diagram - Part_1

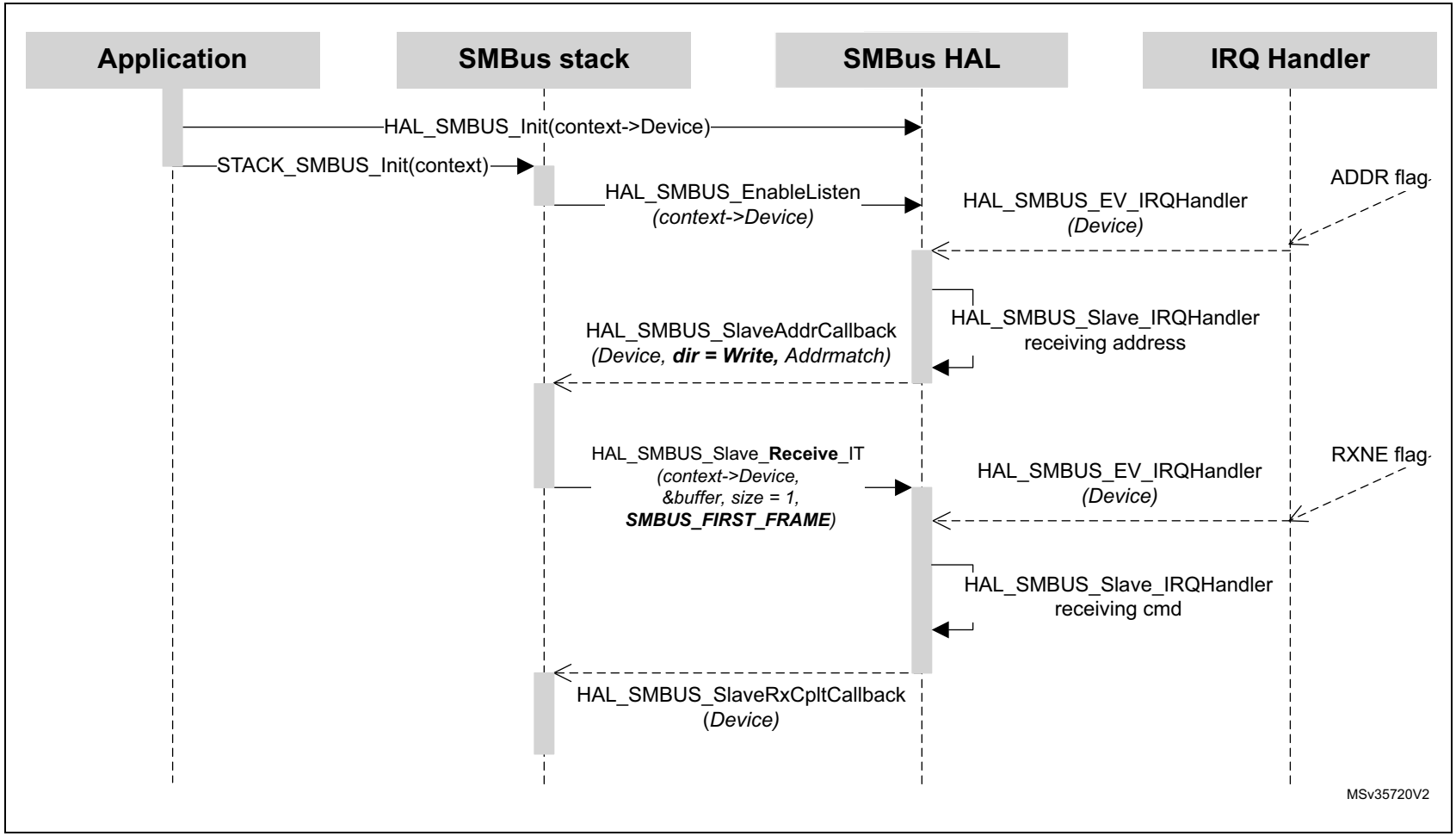
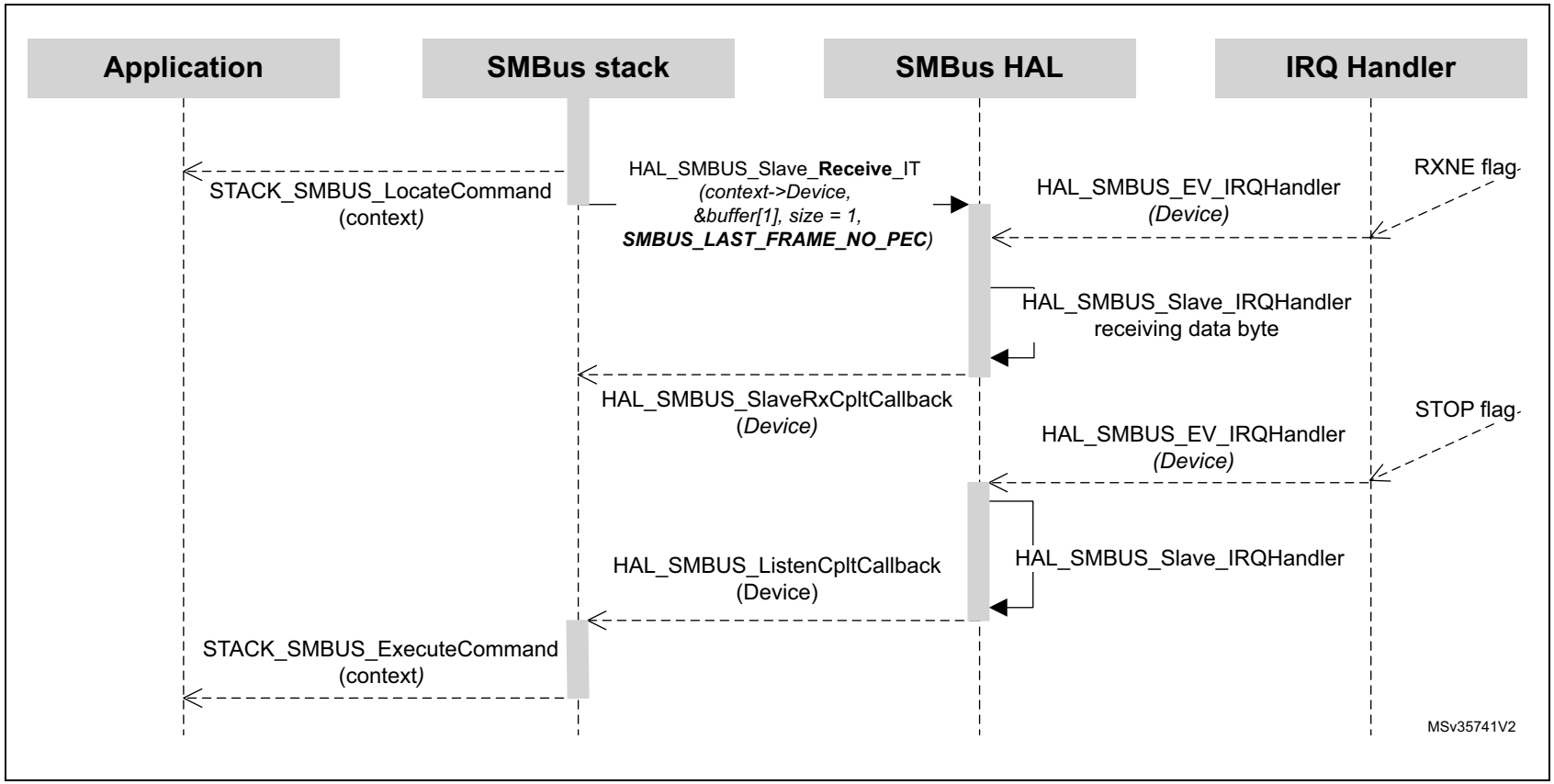


Figure 24. Write byte (slave side) - sequence diagram - Part_2



6.2.5 Write word, Write 32 and Write 64

Figure 25. Write word (slave side) - sequence diagram - Part_1

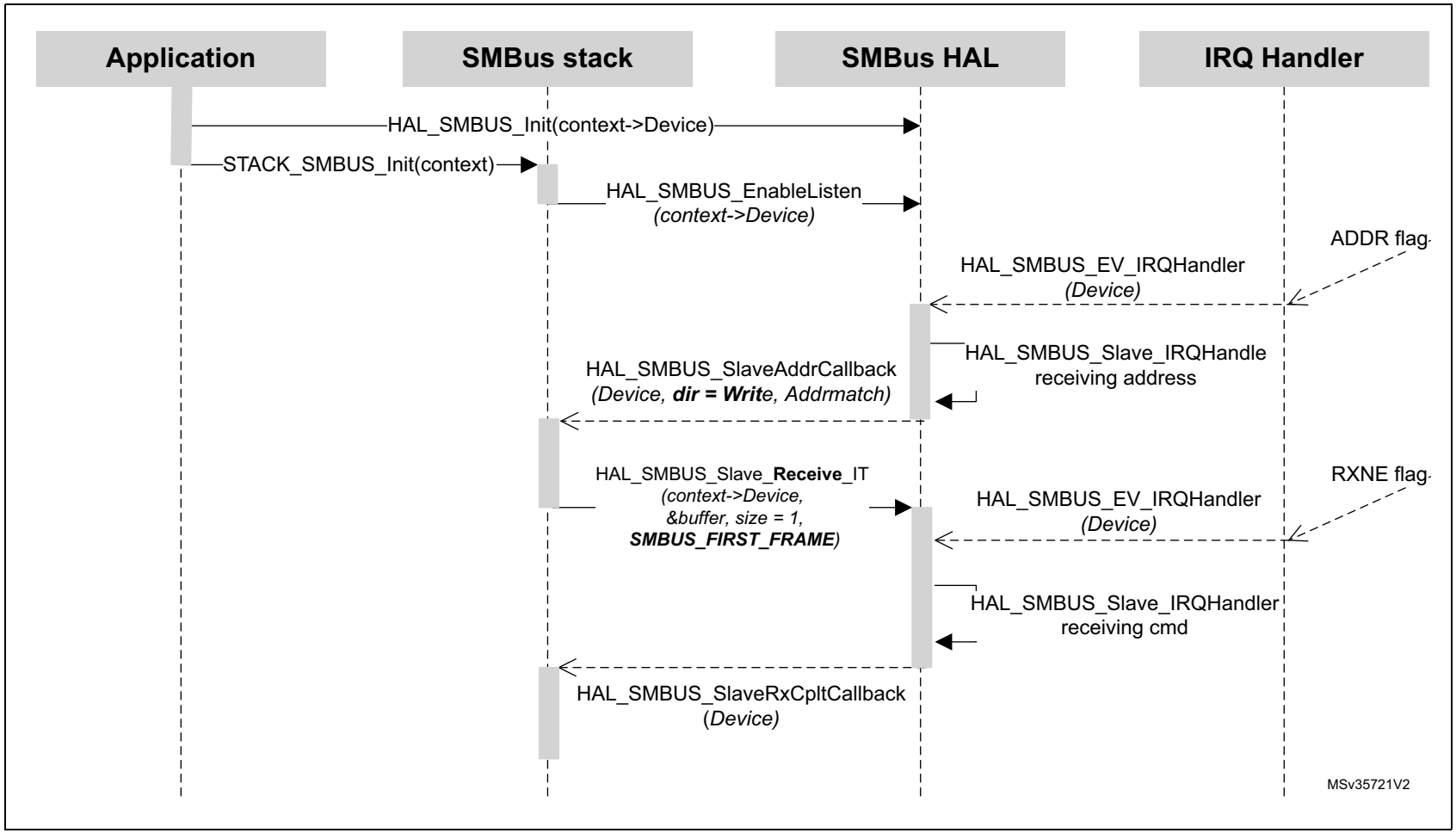
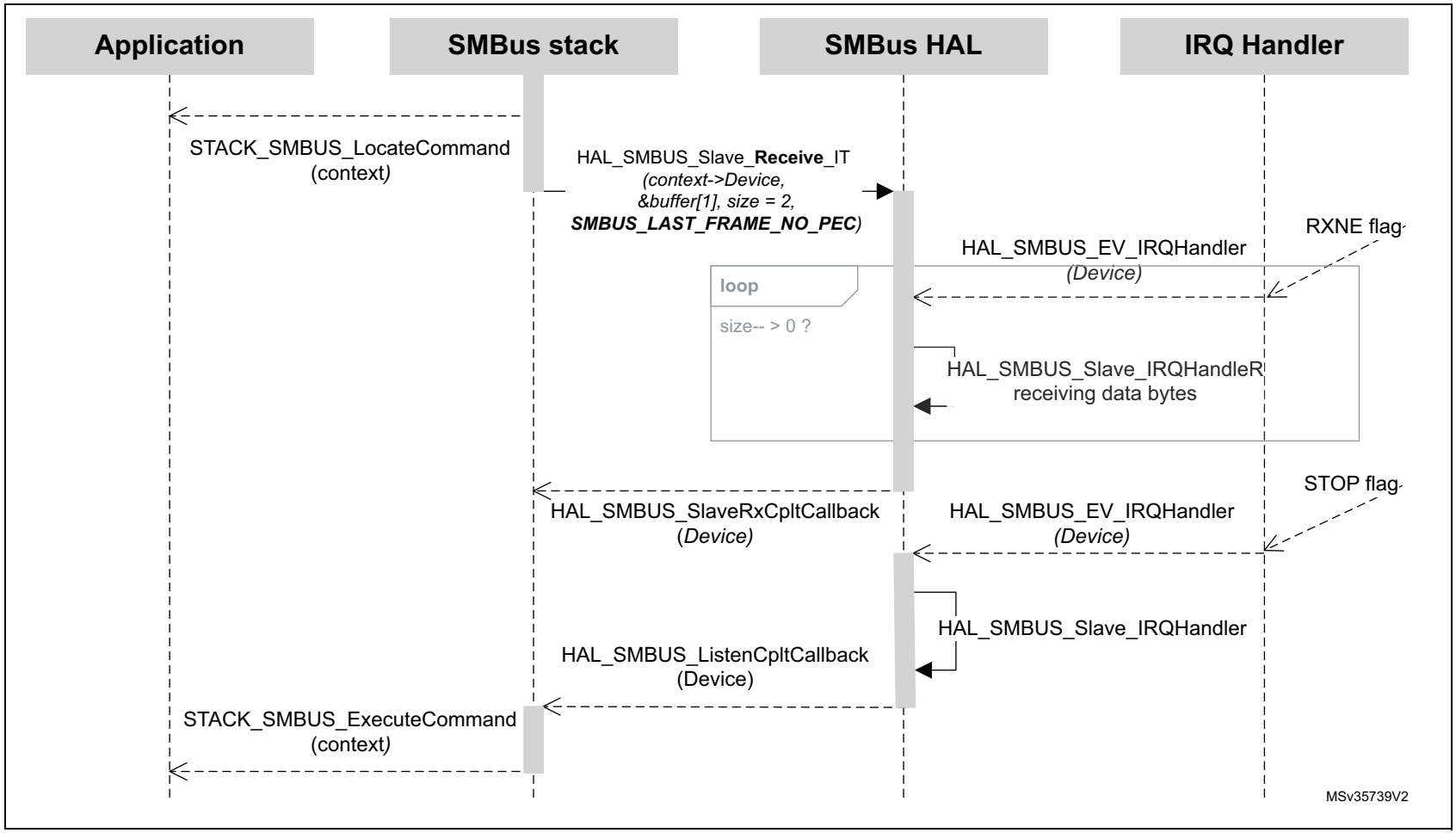


Figure 26. Write word (slave side) - sequence diagram - Part_2



The size parameter in the second HAL_SMBUS_Slave_Receive_IT call is four or height hen respectively Write 32 or Write 64 are identified.

6.2.6 Read byte

Figure 27. Read byte (slave side) - sequence diagram Part_1

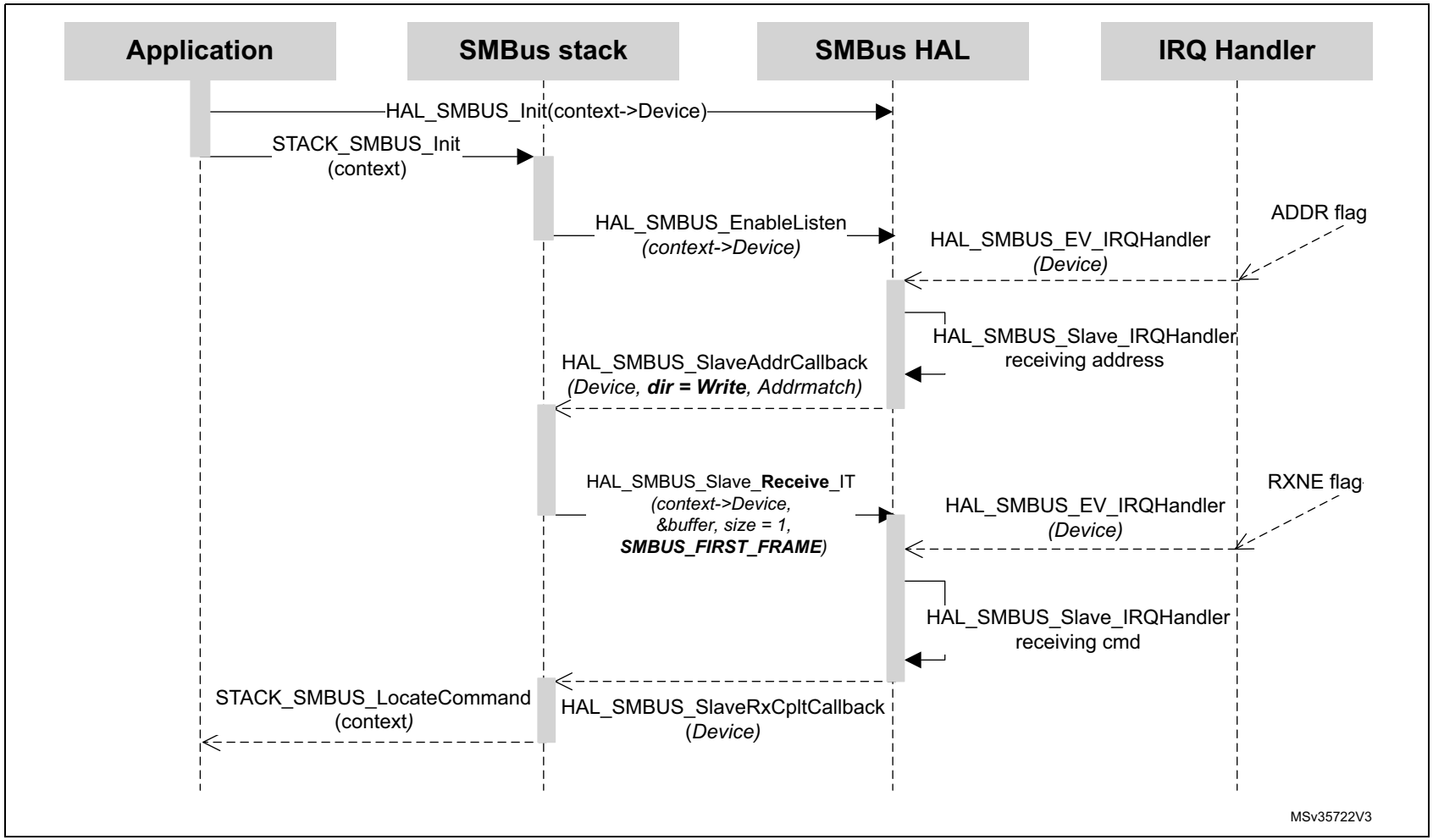
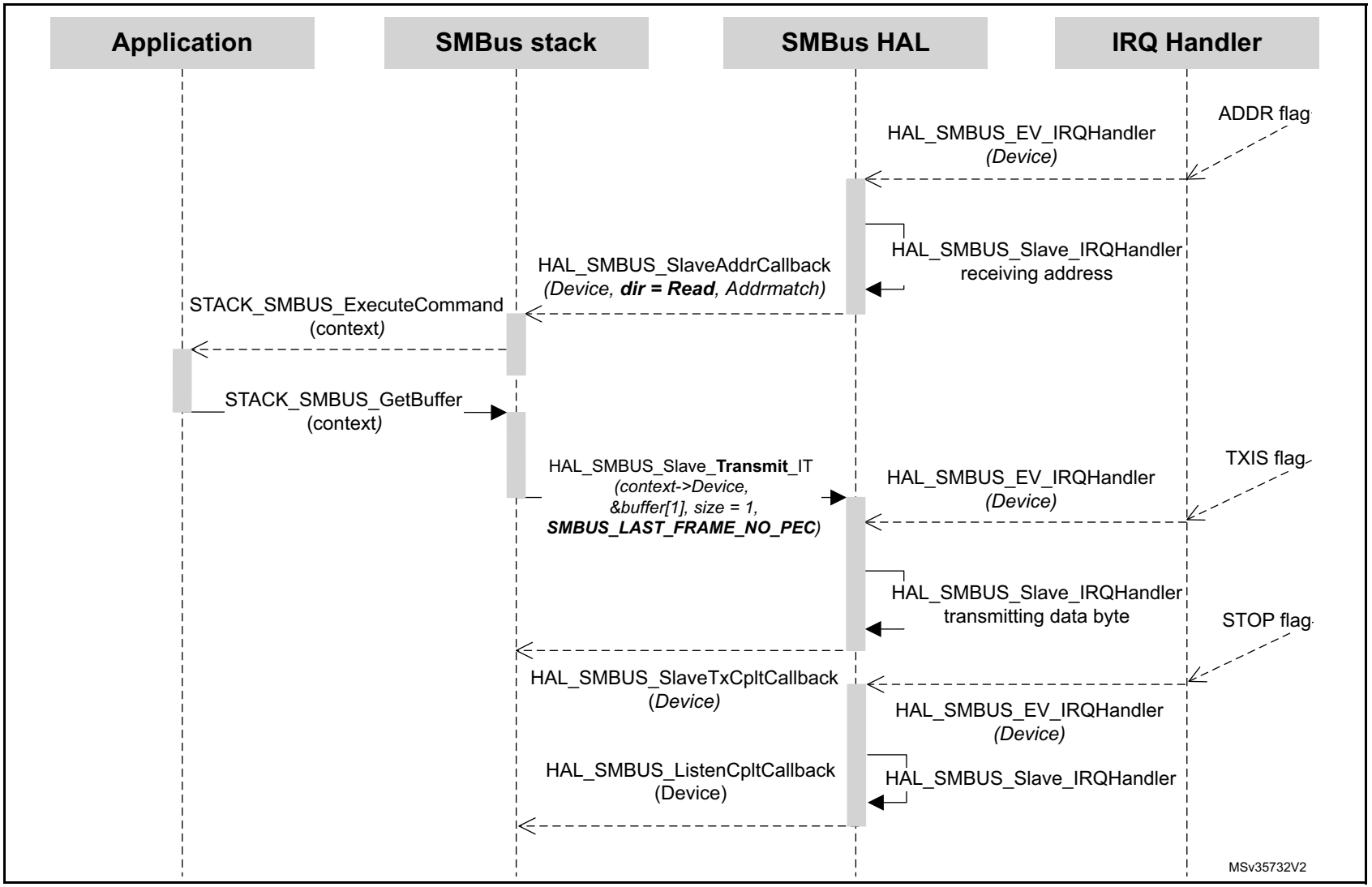


Figure 28. Read byte (slave side) - sequence diagram Part_2



With read class of commands, the timing is crucial. If command execution takes more than 15 or 20 ms, the communication may timeout.

6.2.7 Read word, Read 32 and Read 64

Figure 29. Read word (slave side) - sequence diagram - Part_1

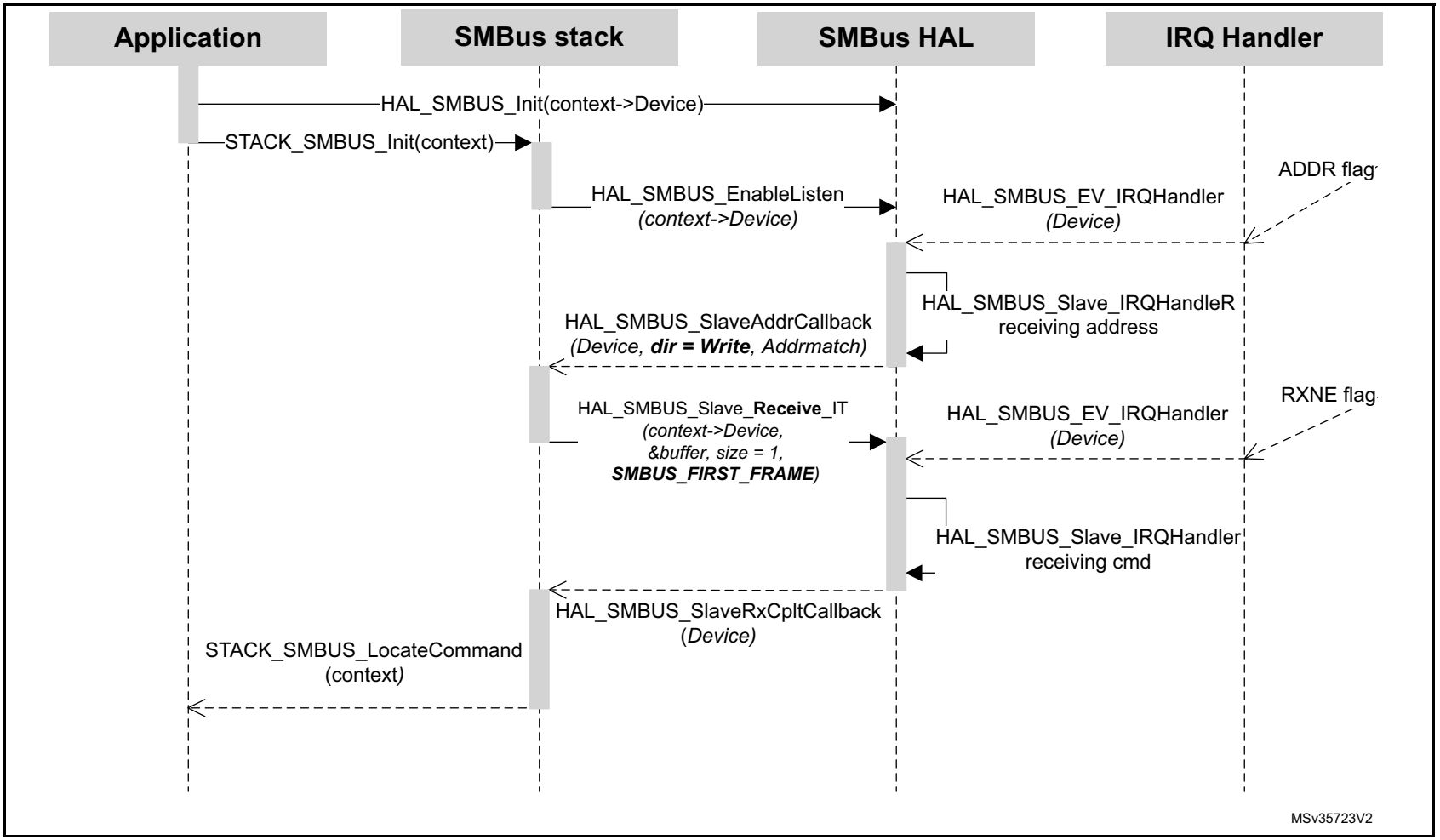
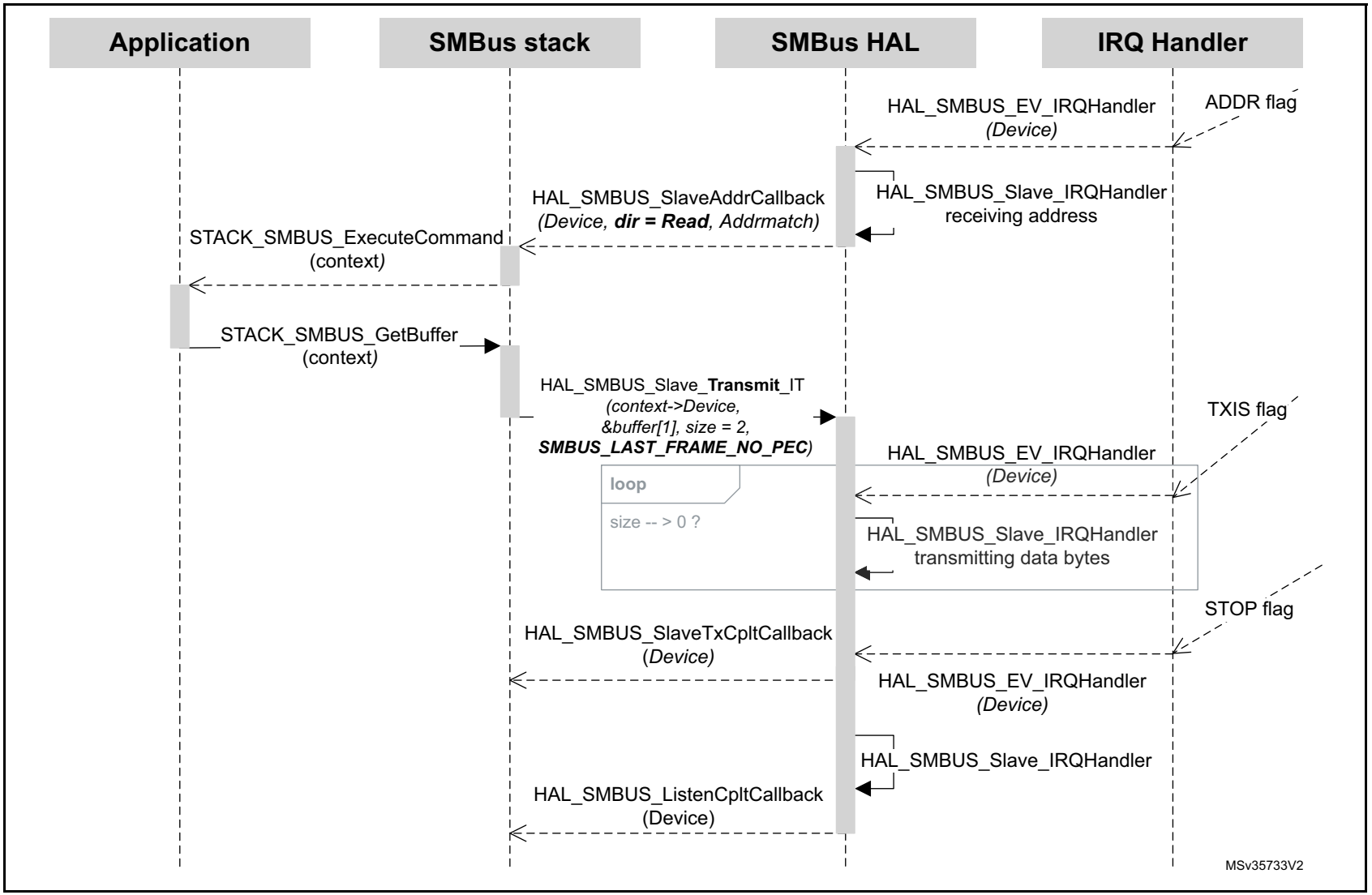


Figure 30. Read word (slave side) - sequence diagram - Part_2



The size parameter in the HAL_SMBUS_Slave_Transmit_IT call is four or height when respectively Read 32 or Read 64 are identified.

6.2.8 Process call

Figure 31. Process call (slave side) - sequence diagram - Part_1

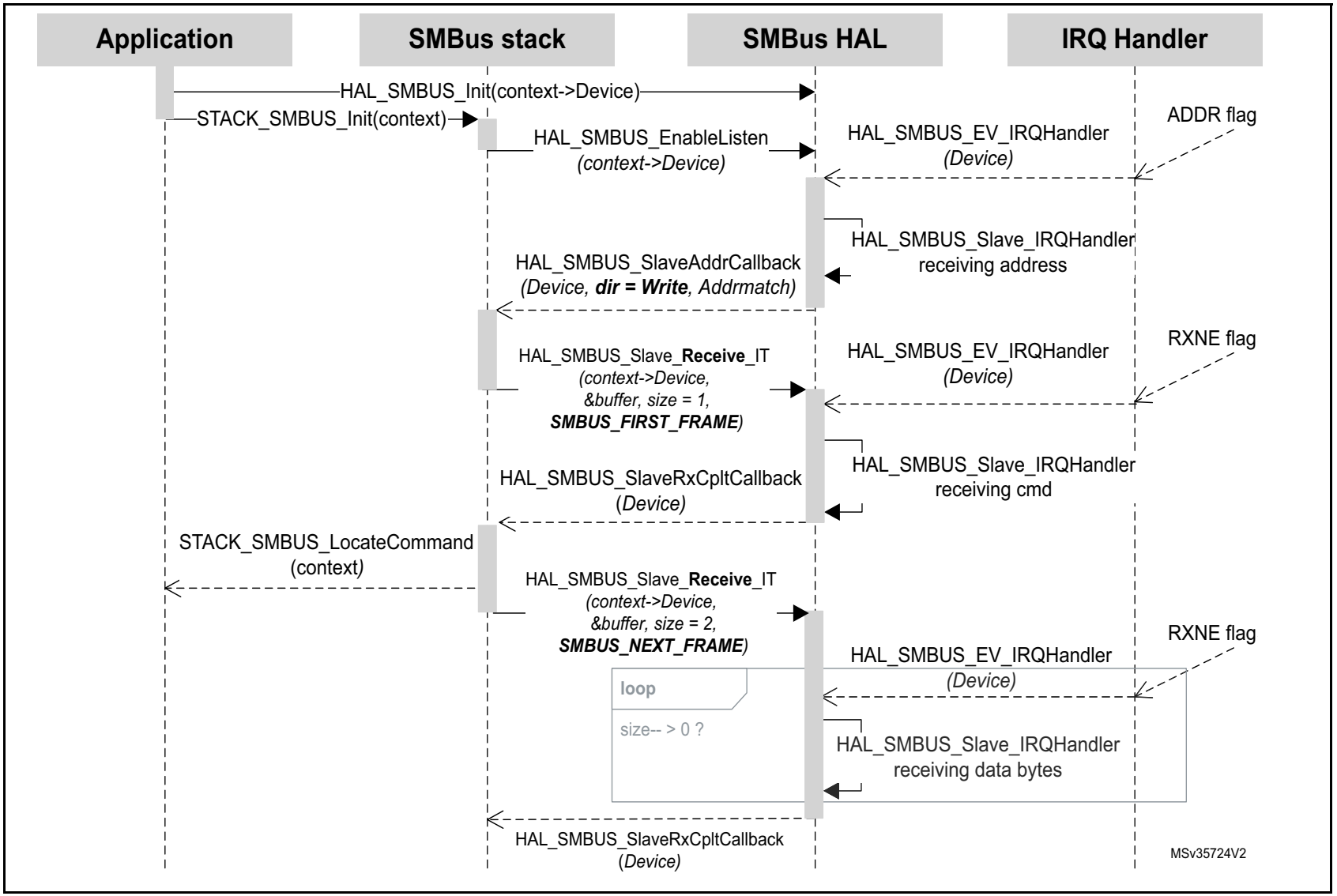
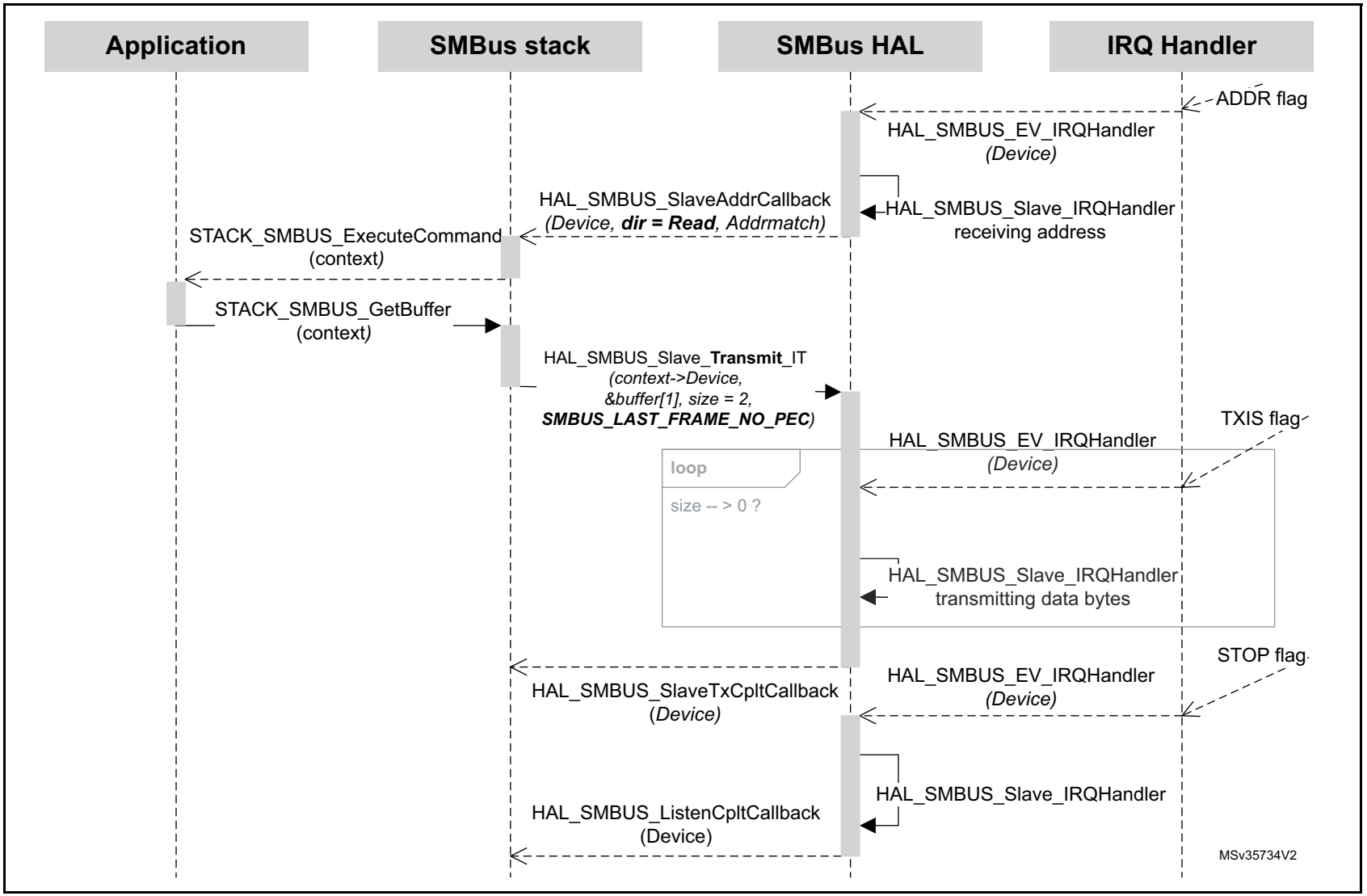


Figure 32. Process call (slave side) - sequence diagram - Part_2



The process call execution bears the same execution timing restriction as the other read commands do.

6.2.9 Block write

Figure 33. Block write (slave side) - sequence diagram - Part_1

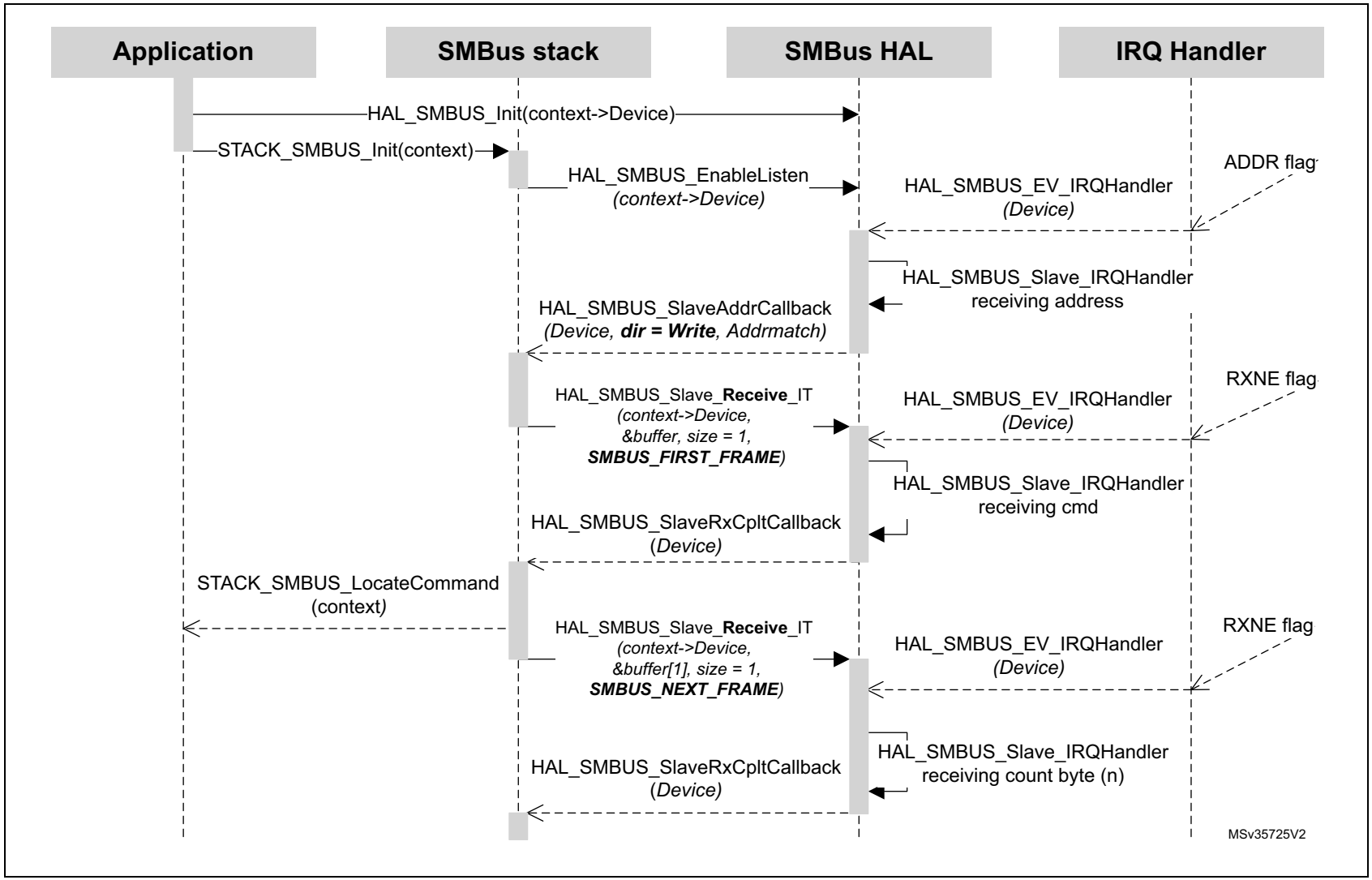
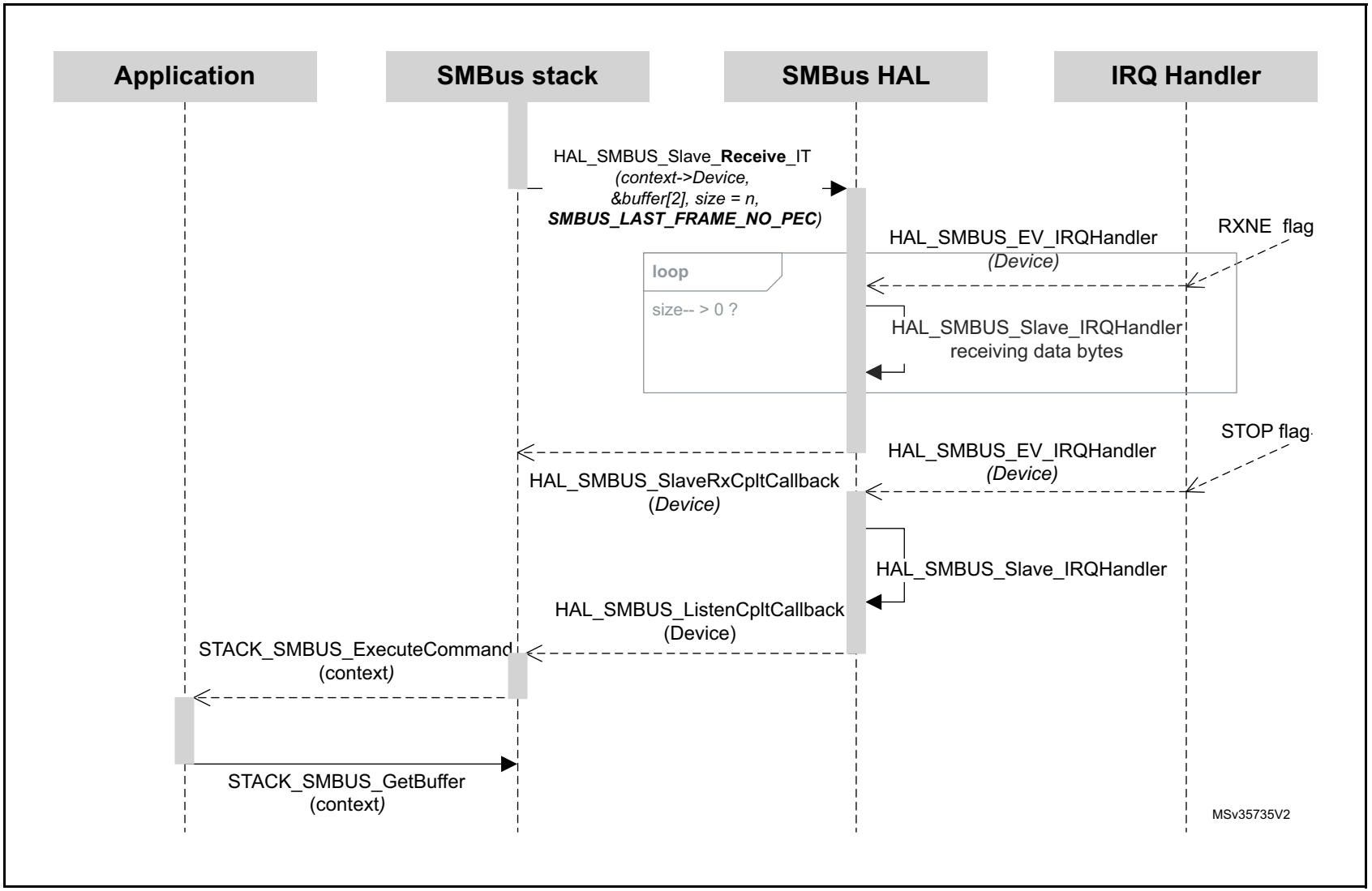


Figure 34. Block write (slave side) - sequence diagram - Part_2



6.2.10 Block read

Figure 35. Block read (slave side) - sequence diagram - Part_1

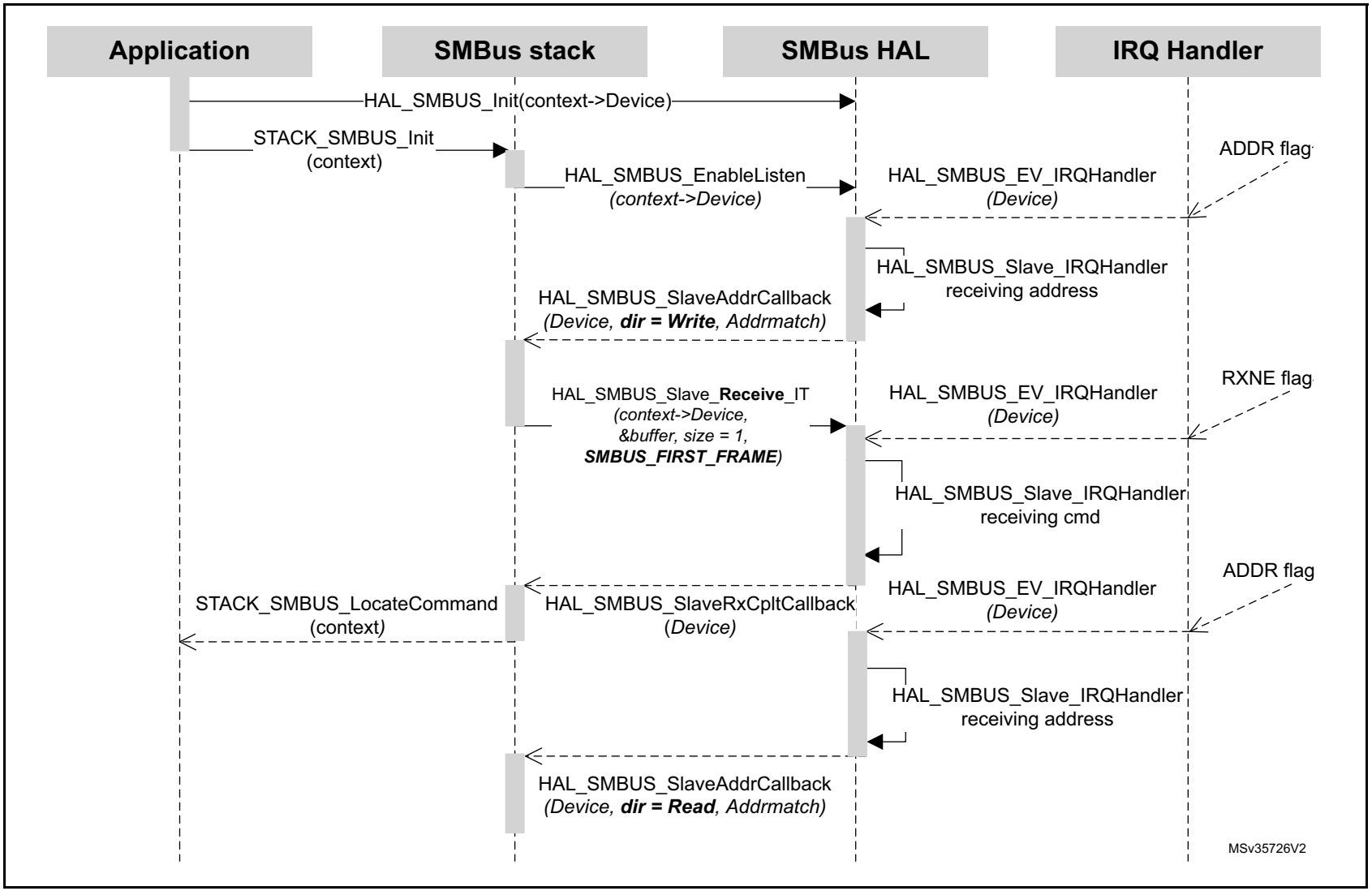
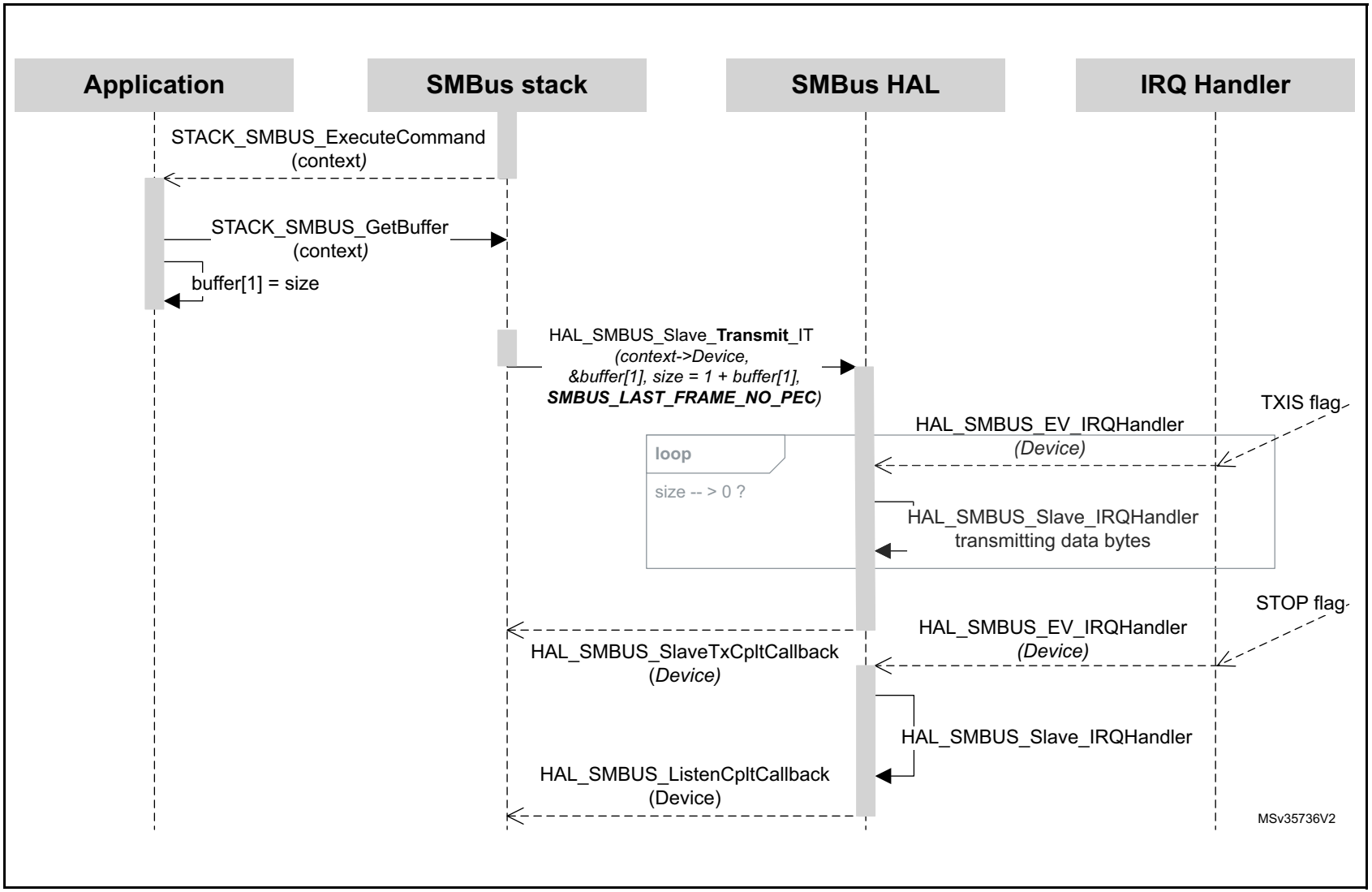


Figure 36. Block read (slave side) - sequence diagram - Part_2



The Block read bears the same execution timing restriction as other read commands do. The data size has to be set to the location returned by GetBuffer.

6.2.11 Block write – Block read process call

Figure 37. Block write - Block read process (slave side) - sequence diagram - Part_1

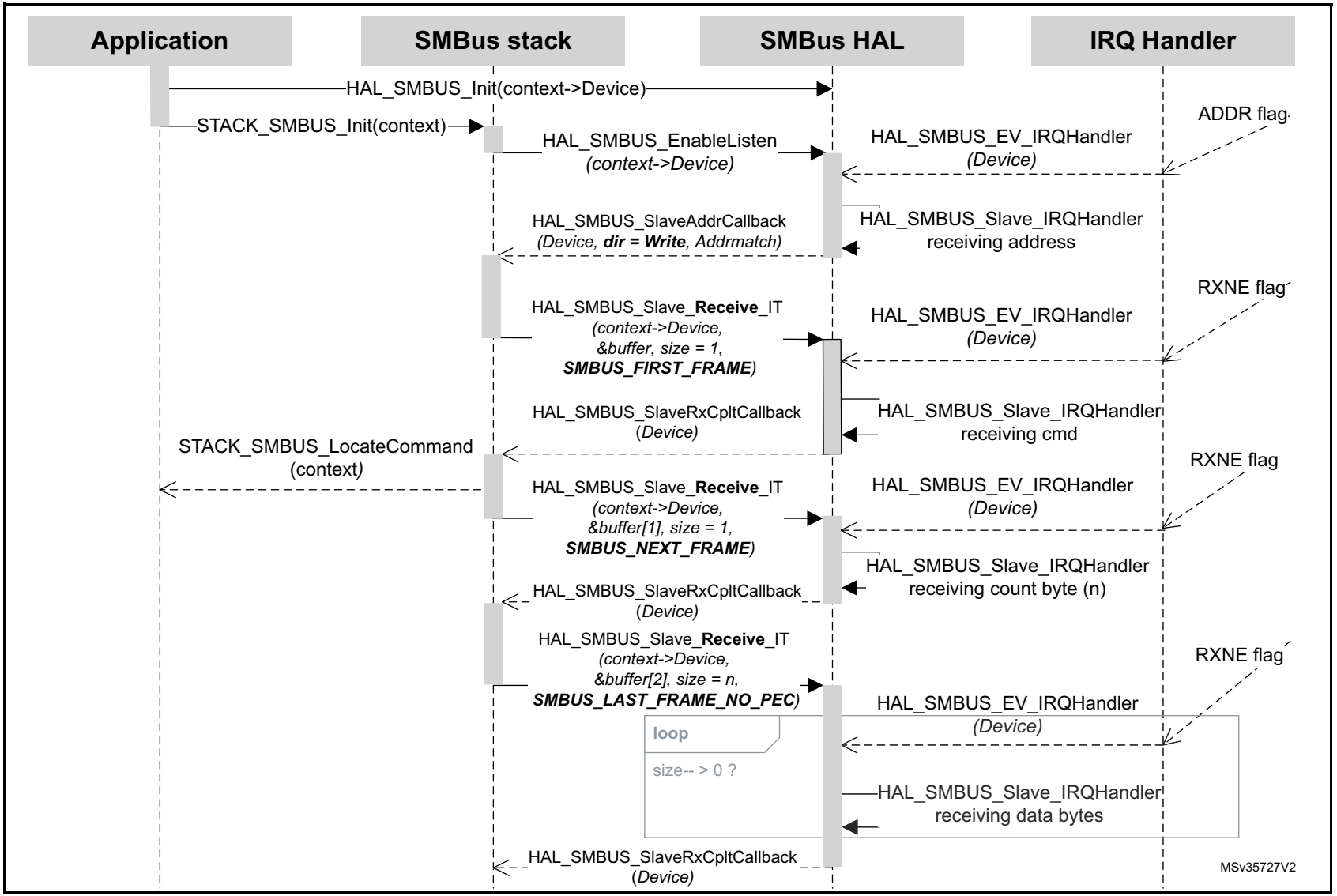
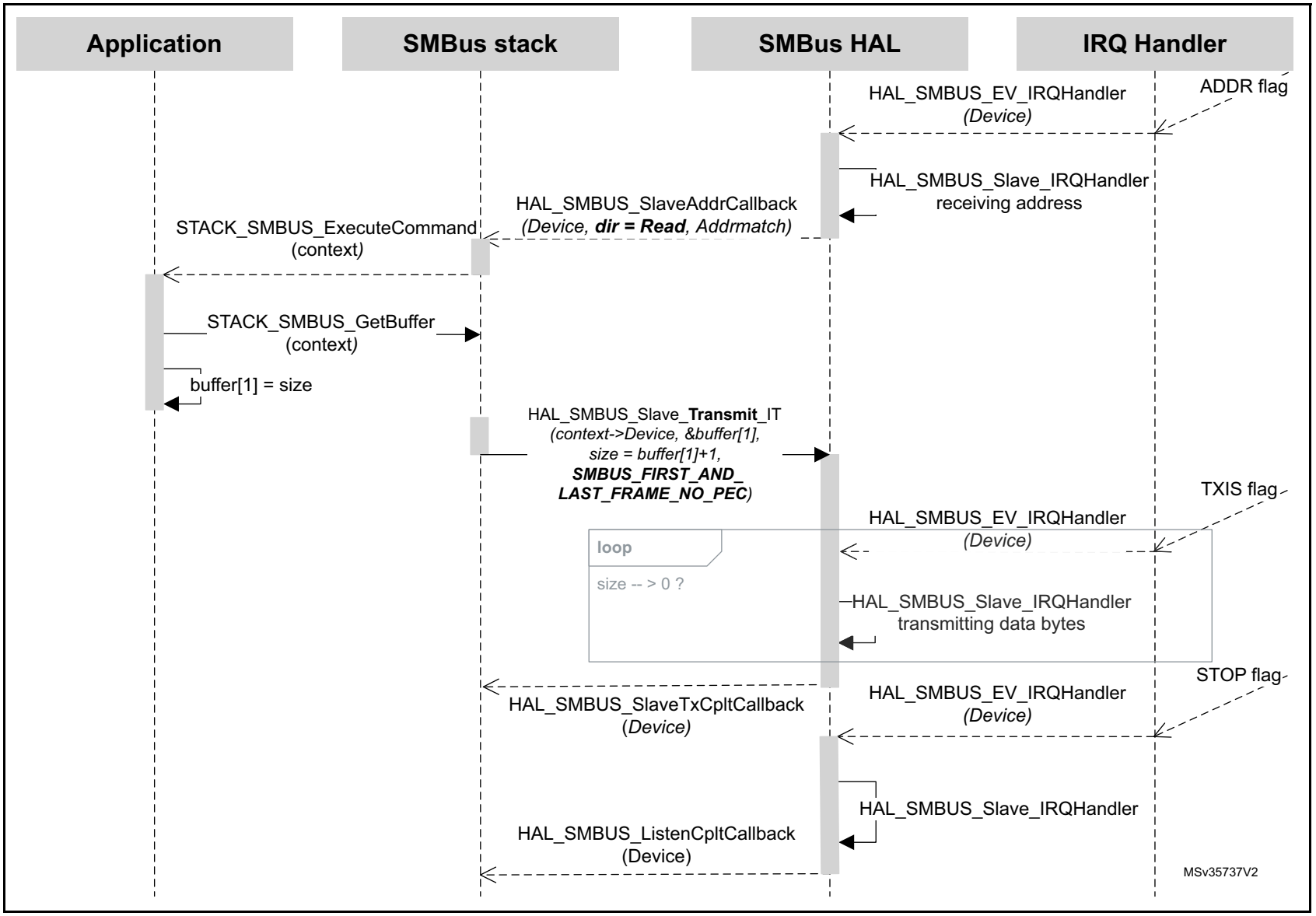


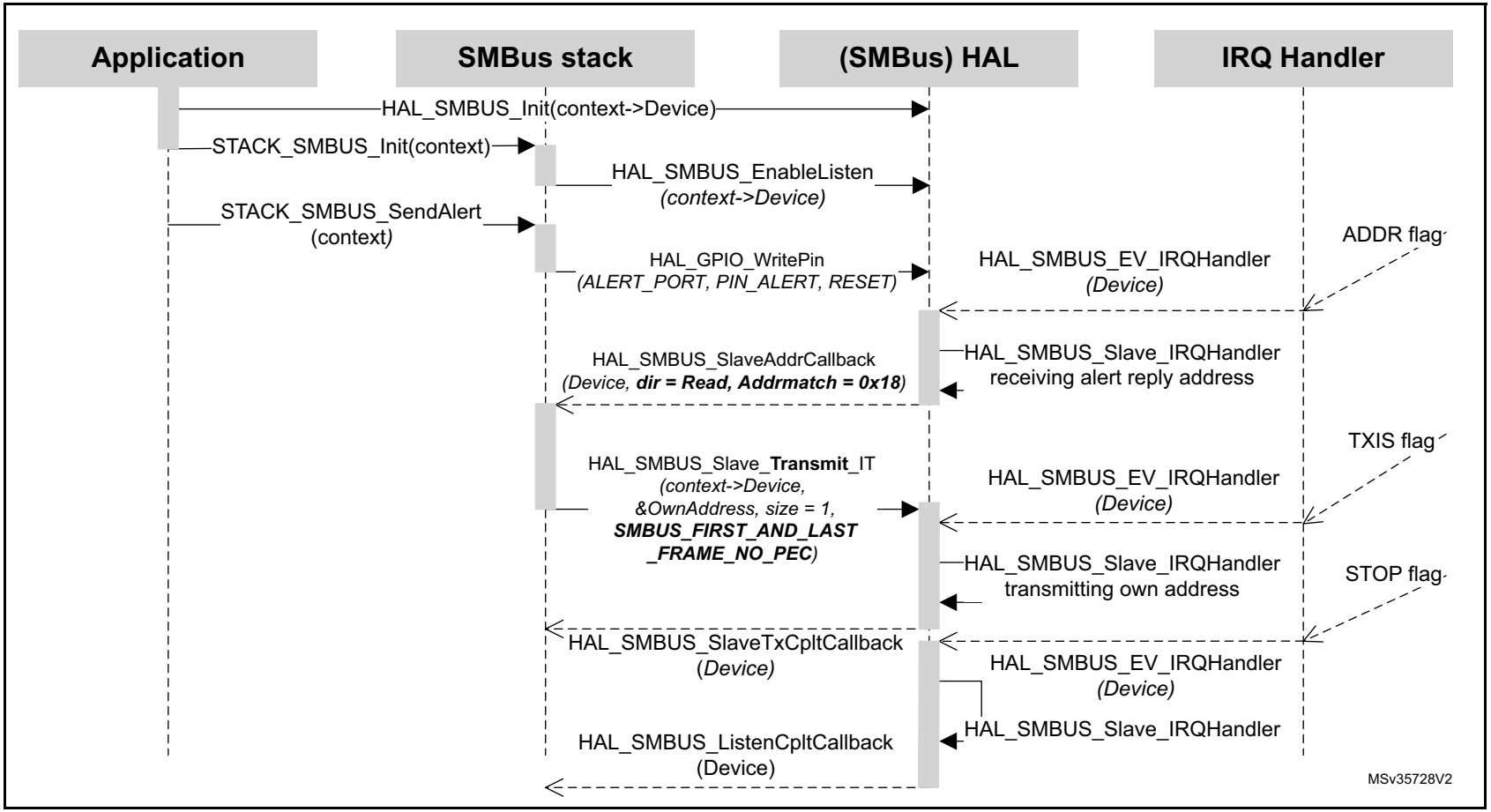
Figure 38. Block write - Block read process (slave side) - sequence diagram - Part_2



6.2.12 SMBus alert

The alert signal treatment is mostly automated in the stack. The application only needs to make sure the OwnAddress attribute is correctly configured in the stack handle.

Figure 39. Alert signal processing (slave side) - sequence diagram



To prevent responding to an ARA command initiated by other device signaling, the stack checks that the device currently issues the alert signal. It is then recommended for the application not to de-assert this signal.

6.3 Address resolution protocol (ARP)

The ARP enumerates the bus, assigns addresses to all devices needing one, or confirms addresses of devices with address preset. This stack implements a part of the ARP, though some choices and actions, particularly on the host side, are left to be done by the application.

6.3.1 Host side

The crucial ARP host structure, the device list, is left on application discretion to be implemented. It may be partially filled with fixed addresses for specific purposes, partially persistent.

The host initiates the ARP automatically on reset, or waits for the ARP host notify command (option implemented in the example). An implementation of an ARP host notify into *STACK_SMBUS_ExecuteCommand* (*Section 5.1.4*) is application choice.

Upon reception, the application should initiate the ARP by calling the sequence of ARP commands using the *STACK_SMBUS_HostCommand* (*Section 5.2.1*), and using the templates from the PMBUS_COMMANDS_ARP table collecting specific ARP commands (*Chapter 5.4*).

6.3.2 Device side

The device must have the unique device identifier ARP-UDID initialized in the stack handle and one of the device type - operating modes selected. The device type and behavior is selected compile-time in the present stack version, using DEV_PSA and DEV_DIS define configuration options.

The PeripheralMode item of the SMBUS HAL driver initialization structure has to be initialized to setup proper reaction upon the SMBus default device address (0xC2), preferably by setting the init.PeripheralMode to value SMBUS_PERIPHERAL_MODE_SMBUS_SLAVE_ARP.

It is also important for the application to correctly initialize the ARP_AR and ARP_AV flags in the stack state machine (*Section 4.2*).

If the application needs the bus host to assign an address to the device it runs, the application must call *STACK_SMBUS_NotifyHost* (*Section 5.3.1*). If the OwnAddress field in the stack handle structure is not assigned yet (meaning it equals zero), the stack sends the ARP notify host frame.

Note: As the SMBus has no mechanism to prevent bus collision, the application should check the stack handle state machine to see whether the notify host was processed correctly.

The responses to ARP command are then implemented in the stack functions *STACK_SMBUS_LocateCommand* (*Section 5.1.3*) and *STACK_SMBUS_ExecuteCommand* (*Section 5.1.4*), which, in the default state, replace the regular ones each time the address is the default device address.

If the application needs to redefine the *STACK_SMBUS_LocateCommand*, it must then keep the ARP part in the beginning in order for the ARP to work.

Once the ARP is complete and an address is assigned, the stack handle attribute OwnAddress and the ARP_AR and ARP_AV flags are set.

6.4 Known limitation

Particularly in the case of the ARP, there may be a situation when a device firmware realizes that it must not respond to a read request. Even though the address decoding finds the acceptable address on the input, sometimes the firmware wants to quit the communication, avoiding to stretch the clock or transmitting any data.

This is not the case in the current implementation, even when there is still room for improvement. It is particularly problem of Address Match Callback function, when IGNORED flag is set in the stack handle state machine.

7 PMBus support

7.1 Command support

The PMBus stack implementation includes a command table based on reference *[3]*. The command implementations are not supplied by the stack.

Different versions of the specification used for the command table can be selected in compile-time using *defines.* More details about *defines* are found in *Section 9.2*.

7.2 PMBus stack API extensions

7.2.1 STACK_PMBUS_HostCommandGroup

Parameters:

pStackContext (*Section 4*).

pCommand - pointer to the command descriptor of the command to be transmitted, NULL for quick command.

address - slave address to be used in the transmission, 16-bit unsigned integer.

last - parameter set to 1 indicates that STOP condition should conclude this transmission 8-bit unsigned type

Returns the HAL_StatusTypeDef response code.

Group commands are transmitted using the *STACK_PMBUS_HostCommandGroup* function. For each command in the group, this function has to be called. In between the commands, a call to the *STACK_SMBUS_GetBuffer* (*Section 5.5.2*) can be done to modify the contents of the output buffer.

It is mandatory to make this GetBuffer call (and test it returns valid address) even when the output must remain the same. It ensures synchronicity of the output.

The last call of *STACK_PMBUS_HostCommandGroup* must have its last parameter set to indicate that the host has to send a STOP condition to the bus.

7.2.2 Extended command

This feature of the PMBus enables a support of more commands than what can be selected by the 8-bit command code. This implementation enables transmission and reception of extended commands. However, many actions have to be carried out by the application.

Host side

A set of predefined *st_command* structures (extended_read_byte, extended_read_word, extended_write_byte and extended_write_word) should be used with the regular *STACK_SMBUS_HostCommand (Section 5.2.1)* function. *STACK_SMBUS_GetBuffer* (*Section 5.5.2*) sets the command code directly to the output buffer.

Device side

The device receiving the extended command must have:

- the regular *STACK_SMBUS_LocateCommand (Section 5.1.3)* function redefined in the PMBUS stack source file
- a function that chooses the extended_read_byte template regardless of the selected command table the stack uses. The reason is that all the extended commands share the same base command code.

Extended reads are handled as a process call by the stack.

After reception of the real command code, the command implementation *SMBUS_STACK_ExecuteCommand* callback (*Section 5.1.4*) is called. At this point, the application assists the stack in determining the actual structure of the transmission through the following steps:

- No stack handle modification is required for *extended read byte*.
- For *extended read word,* modify the value of CurrentCommand pointer in the actual stack to the predefined extended_read_word structure. The stack transmits two data bytes in response.
- *For Extended write byte,* modifying the CurrentCommand pointer is still necessary but not sufficient. The application has to:
  - secure it receives the callback again once the reception is complete, by clearing *SMBUS_SMS_RESPONSE_READY* flag in the stack state machine (*Section 4.2*).
  - order reception of an extra byte (call directly the HAL driver function *HAL_SMBUS_Slave_Receive_IT*).
  - take note that it has already been called for this command once in order not to request another byte from the master again and to perform the actual command action when callback is called the second time for the same command.
- *Extended write word* differs from *extended write byte* only in number of bytes requested from the driver.

7.2.3 Zone commands

This feature was added in the PMBus specification version 1.3 to address the need of a master to issue a command to several slaves at once. For a write command, this feature allows synchronization of the slaves actions. For a read command, it eases the gathering of the information from the system.

To get a complete description of the Zone system, please refer to references *[2]* and *[3]*. To enable a Zone support, select PMBus version 1.3 when configuring the stack in the STM32CubeMX or define the conditional flag PMBUS13, described in *Section 9.2*.

This stack contains provisions to help in the implementation of the Zone commands and to allow basic testing with the SMBus example. This specification also recognizes the separate notions of Read Zones and Write Zones, as there are often different slaves in the data acquisition and control jobs.

ZONE_WRITE and READ_ZONE new addresses are introduced, allow the addressing of all the slaves supporting Zone control at once.

The following sections describe simple wrappers of the STACK_SMBUS_HostCommand defined to ease the use of the Zone feature.

Zone configuration and associated data structure

The Zone configuration is a regular command addressed to a single device and implemented in STACK_PMBUS_MasterZoneConfig wrapper. Its goal is to configure the device to a specific Read Zone or to a specific Write Zone.

STACK_PMBUS_MasterZoneConfig

Parameters:

pStackContext - stack context (*Section 4*).

address - used in the command call zone, pointer to the SMBUS_ZoneStateTypeDef structure with configuration.

Returns HAL_StatusTypeDef as a regular host command.

A structure is defined to help on the use, convey and store of the Zone state.

```
typedef struct
{
   uint8_t readZone;
   uint8_t writeZone;
   uint8_t activeReadZone;
   uint8_t activeWriteZone;
} SMBUS_ZoneStateTypeDef;
```

This structure is part of the SMBus handle structure for each slave on the bus, and is used to identify the commands intended for itself.

Zone active command

The Zone active command is sent to a ZONE_WRITE address with a ZONE_ACTIVE command code. It is implemented in the stack wrapper function STACK_PMBUS_MasterZoneActive.

STACK_PMBUS_MasterZoneActive

Parameters:

pStackContext - stack context (*Section 4*).

zone - pointer to the SMBUS_ZoneStateTypeDef structure with configuration.

Returns HAL_StatusTypeDef as a regular host command.

Each slave on the bus has to check if the new active Zone is identical to the Zone currently configured. After confirmation, the slave is prepared to receive further Zone commands.

Zone write command

Any command can be sent to the Zone *write recipient address,* as long as it does not involve any reading (not even as a Block read or a process call). The only difference is in the address value. All the slaves within the active Zone receive and execute the command.

Due to the physical properties of the bus, the ACK nature is OR. The master cannot have then an immediate acknowledge from all the slaves. One acknowledge is enough. The wrapper STACK_PMBUS_MasterZoneWrite is used to transmit the command.

STACK_PMBUS_MasterZoneWrite

Parameters:

pStackContext - stack context (*Section 4*).

pCommand - pointer to the command sent to the Zone. NULL for quick command.

Returns HAL_StatusTypeDef as a regular host command.

Zone read command

The Zone read capability is not mandatory as it requires a special arbitration and a repeated start capability. It is however supported in both Master and Slave modes in its basic form by executing the STACK_PMBUS_MasterReadZoneStatus function wrapper.

STACK_PMBUS_MasterReadZoneStatus

Parameters:

pStackContext - stack context (*Section 4*).

ccode - control code.

mask - sent with the read request.

Returns HAL_StatusTypeDef as a regular host command.

8 Configuration

The STM32CubeMX GUI tool and the software package generate the correct configuration. The steps are detailed in the *Section 8.1: The software package*.

The manual configuration of the stack is also possible. The steps are described in the *Section 8.2: Manual configuration*.

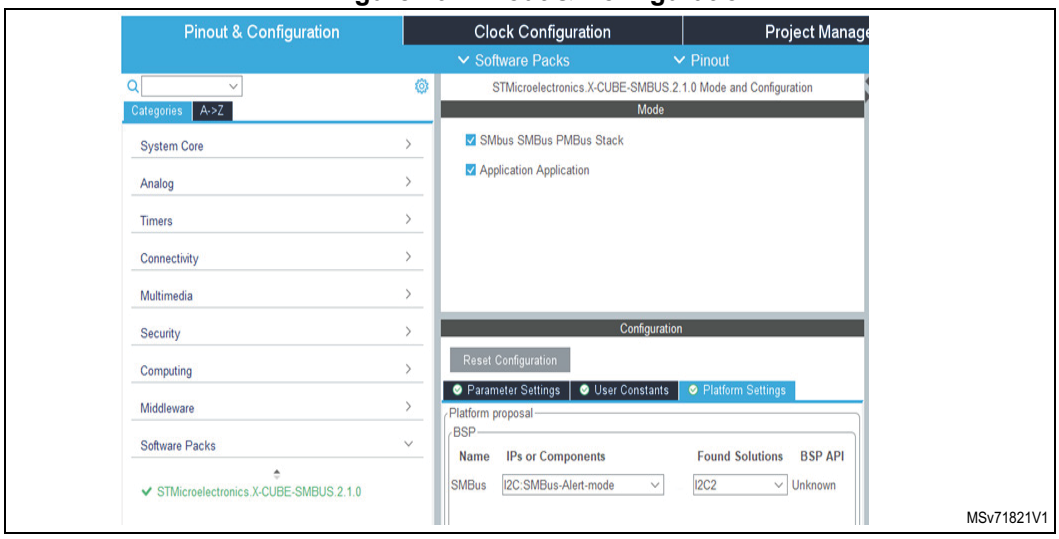
8.1 The software package

The CubeMX supports the usage of plug-ins, allowing the GUI to:

- configure the software, which is delivered with the software package.
- project that configuration into the generated project.

After installing the X-CUBE-SMBUS software package, it can be added to the “Pinout & Configuration” tab. This tab is accessible from the “Software packs” dropdown menu.

Figure 40. Pinout & Configuration



8.1.1 Configuring the interface in the HAL

To configure the interface in the HAL, search for “I2C” under the “Connectivity” category:

- Select the suitable I2C instance and all the known parameters, such as the own address, the bus speed and the role (master/slave).
- Select the available GPIOs. Also for Alert if needed.
- Enable all the interrupts in the “NVIC settings” tab. The stack operates exclusively in the interrupt mode, and it expects both event and error interrupts that were enabled in the “NVIC Settings” tab from the I2C HAL configuration.
- For ARP and general call support, the “General Call Address Detection” HAL slave setting must be enabled.
- For PMBus 1.3 support of the Zone commands, the “Dual Address Acknowledged” HAL slave setting must be enabled. The recommended settings are 0x40 for the secondary slave address, and 0x05 for the secondary address mask.

8.1.2 Configuring the stack

Tick both checkboxes in the software pack upper “Mode” window.

Make sure that the stack configuration matches the HAL configuration. To obtain functional results, the settings for Master/slave, PEC, and alert must be aligned between the stack and the HAL before proceeding to project generation. If the assert option is enabled in the project generation, the generated code checks the settings alignment for basic parameters.

8.1.3 Generating the project

After following these steps, it is now possible to generate the project.

The Slave configured device calls by default the STACK_SMBUS_ExecuteCommand() function for each command received. This is the most likely place to start the actual command implementation.

The Master is expected to call STACK_SMBUS_HostCommand() to initiate a communication.

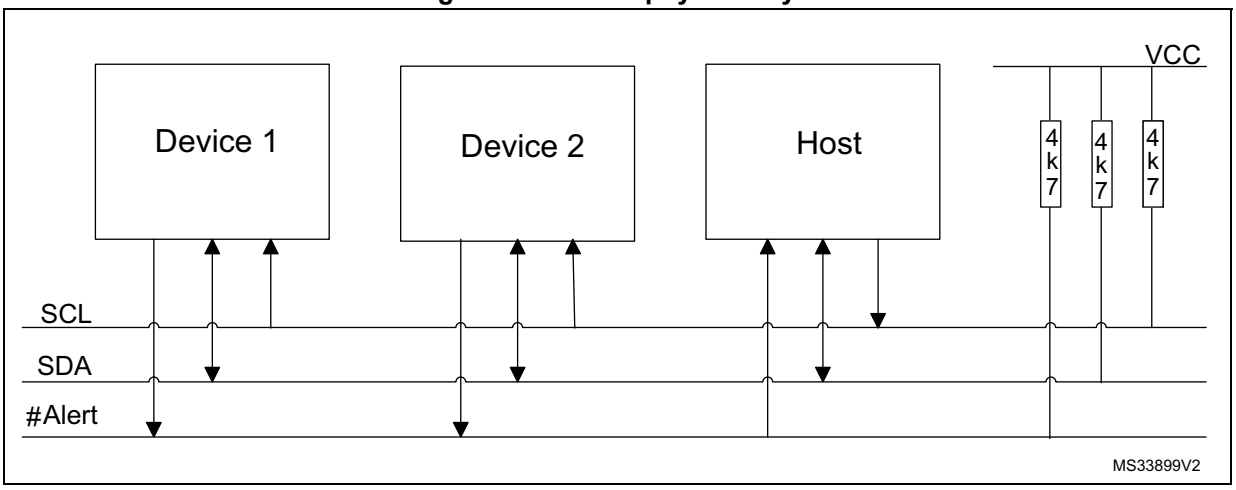
8.2 Manual configuration

8.2.1 HW initialization

The stack follows the usual STM32Cube software structure. It is necessary to add the I2C I/O configuration to the MSP_init function with the following steps:

1. Enable the clock domain for the I2C.
2. Properly configure the ports and GPIO alternate function registers, including the alert pin, in case it is used in the application. It is recommended to use an external pull-up as too many internal pull-ups in devices may stall the bus.
3. Configure and enable the interrupt.

Figure 41. SMBus physical layer



8.2.2 Registering the IRQs

Interrupts are not treated on the stack level. The HAL driver is configured directly. For example, STM32F3 and STM32L4 series have separate interrupt vectors for error and event interrupts, while STM32F0 series and STM32L0 series treat both events and errors in single interrupt vector. The recommended solution is to call first an event handling, following by an error handling in the interrupt service routine.

It is important to correctly assign the vectors to particular instances of HAL driver handles. The stack makes no attempt to automate the resource assignment in this case for performance reasons.

8.2.3 Initializing the driver

For details, please refer to the SMBUS HAL driver documentation in the source files.

8.2.4 Initializing the stack

The objective of the stack initialization is to register the stack handle instance and to link it with a driver instance and thus a physical interface. Allocate a variable of SMBUS_StackHandle type and set the following attributes:

- CMD_Table – initialize with pointer to a set of supported commands structured
- CMD_TableSize – set number of records in the supported command stable
- Device – initialize with pointer to the driver handle
- APR-UDID – initialize the UDID if ARP code is included.

Additional attributes may need to be initialized:

- SRByte – if the device configured supports Receive byte transaction
- OwnAddress – ifdevice with PSA – persistent address. This must be initialized.
- StateMachine – SMBUS_SMS_RCV_BYTE_OFF, SMBUS_SMS_RCV_BYTE_LMT and SMBUS_SMS_PEC_ACTIVE flags must be initialized if used. Same for the ARP_AR and ARP_AV flags, if ARP is used (See *Section 4.2* for details).

The initialization is concluded by calling the function *STACK_SMBUS_Init* (*Section 5.5.1*) with pointer to the prepared *SMBUS_StackHandle* structure as parameter.

8.2.5 Optimization

The example code is not optimized. The primary goal is to provide a solution that is easy to understand, integrate, and maintain. Portability to wide range of STM32 products was also a high priority.

For products with limited amount of available program memory, we suggest to undefine conditional define flags such as ARP or PMBUS13 to drop support of unused features from the built code. It is also possible to define other such flags to limit the functionality to host only or device only, if the complete, full featured, implementation is not necessary.

Further optimizations would probably require complete rewriting of the stack, probably using the low level HAL library.

8.2.6 Conclusion

After enabling the interrupts, the SMBus device and the stack are operational.

9 Project example

An example code is supplied with this application note, demonstrating the usage of all the SMBus and PMBus features.

The example is developed and tested with Discovery boards of STM32F0, STM32F3, STM32L0, and STM32L4 series. Other devices such as STM32H7 and STM32WB series are tested with Nucleo boards. This section describes the STM32F3 series version as all others are similar but with only one SMBus interface, configured in the example, on different pin numbers. The readme file in each particular project describes the differences.

9.1 Hardware setup

Two boards are needed to get a properly working example: one as host and the other as device.

The SCL and SDA pins of the two boards are connected with wires (for example PB6 and PB7 pins for the STM32F3 series Discovery boards). Refer to the readme file provided with the examples to see the pins used on other boards. If the host side does not have internal pull-up configured, external pull-up resistors are needed. It is not recommended to have internal pull-up configured on more devices. To test the alert capability, interconnect the alert pins.

9.2 Configuration

Various features and working modes of the example firmware can be configured using the following preprocessor defines:

- **SMB1** – define to configure I2C1 to SMBus (Port B)
- **SMB2** – define to configure I2C2 to SMBus (Port A, available in selected examples only)
- **ARP** – define to enable the automatic address resolution protocol execution in the code startup
- **DEV_DIS** – device is discoverable – reacts to general ARP calls on default address
- **DEV_PSA** – device has a persistent address.
- **HOST1** – define to configure I2C1 as the bus host. If ARP is defined as well, the host waits for “ARP notify host” command from a device. Otherwise, the host immediately attempts to send loop of test commands to the device on address that goes first in the device list (address 0x1A by default).
- **TEST2** - replaces the first test command with a PMBus group command test.
- **TEST3** - replaces the initial 4 test cases with PMBus extended command tests.
- **TEST4** - replaces the default test group with a set of actual PMBus commands.
- **TEST5** - switch to test the Zone commands.
- **PMBUS12** - includes commands newly added to the PMBus specification in V1.2.
- **PMBUS13** - includes new features such as Zone operations from PMBUS 1.3 and SMBUS 3.0.
- **ALERT** - enables an alert signal. It must be defined both in master and slave.
- **USE_PEC** - switch to enable packet error checking. Implicitly enabled with ARP.

9.3 Functioning of the example

The host-configured Discovery board flashes a LED light while transmitting commands. The device receiving the commands flashes a LED in the frame frequency.

The host mounted user buttons (usually with blue cap) is available to cycle between commands (test cases).

The device mounted user button may be used to send an alert or a notify.

10 Revision history

Table 4. Document revision history

Date	Revision	Changes
17-Oct-2014	1	Initial release.
6-Jun-2017	2	Updated: – *Introduction* – *Section 1: Definitions* – *Section 2: Resources* – *Section 3: Features* – *Section 3.1: SMBus features* – *Section 3.2: PMBus features* – *Section 4: SMBus stack context* – *Section 4.1: The context structure* – *Section 4.1.9: SlaveAddress* – *Section 5.1.2: STACK_SMBUS_AddrAccpt* – *Section 5.1.3: STACK_SMBUS_LocateCommand* – *Section 5.1.4: STACK_SMBUS_ExecuteCommand* – *Section 5.2.1: STACK_SMBUS_HostCommand* – *Section 5.2.2: STACK_SMBUS_HostRead* – *Section 5.3.1: STACK_SMBUS_NotifyHost* – *Section 5.3.2: STACK_SMBUS_SendAlert* – *Section 5.4: ARP specific commands* – *Section 5.4.1: STACK_SMBUS_LocateCommandARP* – *Section 5.4.2: STACK_SMBUS_ExecuteCommandARP* – *Section 5.5.1: STACK_SMBUS_Init* – *Section 5.5.2: STACK_SMBUS_GetBuffer* – *Section 5.7.2: STACK_SMBUS_ReadyIfNoAlert* – *Section 6.1.6: Write word, Write 32 and Write 64* – *Section 6.1.8: Read word, Read 32 and Read 64* – *Section 6.2.5: Write word, Write 32 and Write 64* – *Section 6.2.7: Read word, Read 32 and Read 64* – *Section 6.3.2: Device side* – *Section 6.4: Known limitation* – *Section 7.1: Command support* – *Section 8.2: Register the IRQs* – *Section 9.1: HW setup* – *Section 9.2: Configuration* – *Figure 1: X-CUBE-SMBUS system architecture* – *Figure 2: Simplified state transition diagram* – *Table 2: StateMachine attribute bits* Added: – *Section 4.1.7: TheZone* – *Section 7.2.3: Zone commands*

Table 4. Document revision history (continued)

Date	Revision	Changes
5-Mar-2019	3	Updated: – Title of the document – Introduction – Section 1: General information – Context replaced by handle in most part of the document – Section 3.1: SMBus features – Table 2: StateMachine attribute bits – Section 8: Configuration – Section 9: Project example Added: – Section 5.3.3: STACK_SMBUS_ExtendCommand – Section 8.2.5: Optimization
07-Mar-2023	4	Updated: – *Section : Introduction* – *Section 3: Features supported by the stack* – *Section 4.1.1: StateMachine* – *Section 4.1.2: Device* – *Section 4.1.7: TheZone* – *Section 4.1.13: Buffer* – *Table 2: StateMachine attribute bits* – *Figure 2: Simplified state transition diagram* – *Table 3: Quick command write - sequence diagram* – *Section 7.2.3: Zone commands* – *Section 8: Configuration* – *Section 8.2: Manual configuration* Added software package and configuration information: – *Section 8.1: The software package*