Freescale Semiconductor
User's Guide

Document Number: VKSPRXUG
Rev. 0, 06/2008

Freescale Variable Key Security Protocol Receiver User's Guide

by: Ioseph Martínez and Christian Michel
Applications Engineering - RTAC Americas

Contents

1 Introduction

The software explained in this document allows implementation of a complete remote keyless entry (RKE) system. The variable key security protocol (VKSP) implements secure communication using one-way authentication with 128 bits of encryption. The VKSP software library is divided into two main blocks, transmitter and receiver. This document focuses only on the receiver part.

2 VKSP Overview

VKSP is a protocol intended mainly for RKE systems. It sends commands rather than information. The commands sent through the communications link are visible for any device monitoring the communications channel. VKSP carries two main tasks: the generation of authentic messages and the verification of received messages. These tasks are abstracted from the application layer.

A transmission frame is generated with every command that is sent. The transmission frame has two main parts: the data section and the message authentication code (MAC) section.

The data section is sent un-encrypted to the receiver and has the following information:

- Transmitter ID: This is a three-byte identifier of the transmitter device.
- Command: This is the one-byte command that indicates to the receiver what action to perform.
- Variable Key: This is a 4-byte value incremented with time and ensures that the messages received were not previously sent.
- Message authentication code: This 8-byte code is used to authenticate messages on the receiver side.

The message authentication code ensures that the data information is authentic. The MAC is generated automatically by the VKSP transmitter driver. To generate a MAC, the drivers use local keys generated internally and that cannot be accessed from external functions for security reasons. Figure 1 depicts how the transmission frame is generated.

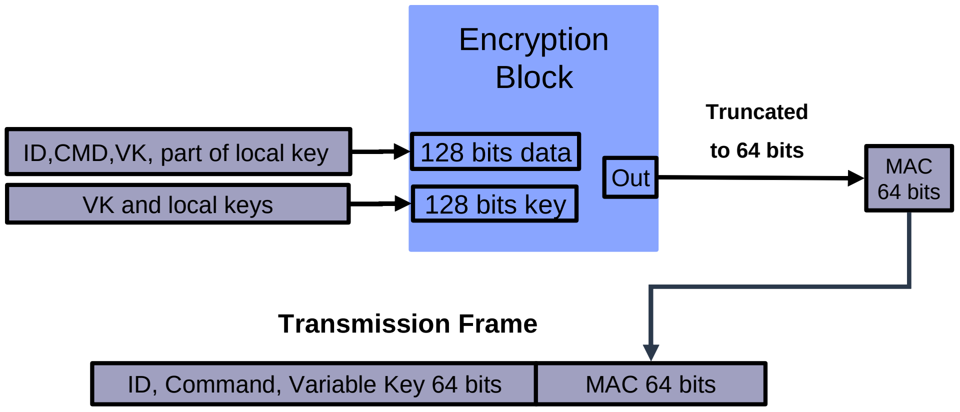


Figure 1. Transmission Frame Generation

The steps taken to verify the validity of a message are:

1. The ID data from the incoming message is extracted. The ID is looked-up in the receiver database. If the ID is found in the receiver database, a local key and a variable key associated with that ID are fetched.
2. The received variable key must be greater than the stored variable key. This step ensures that any re-transmitted frame (a frame that was already sent before) is not accepted.
3. The authentication is performed. A message authentication code (MAC) is generated from the received data, local data, and local keys. This generated MAC is compared to the received MAC from the received message. If these two MACs are equal the command is accepted.

The authentication process is illustrated in Figure 2.

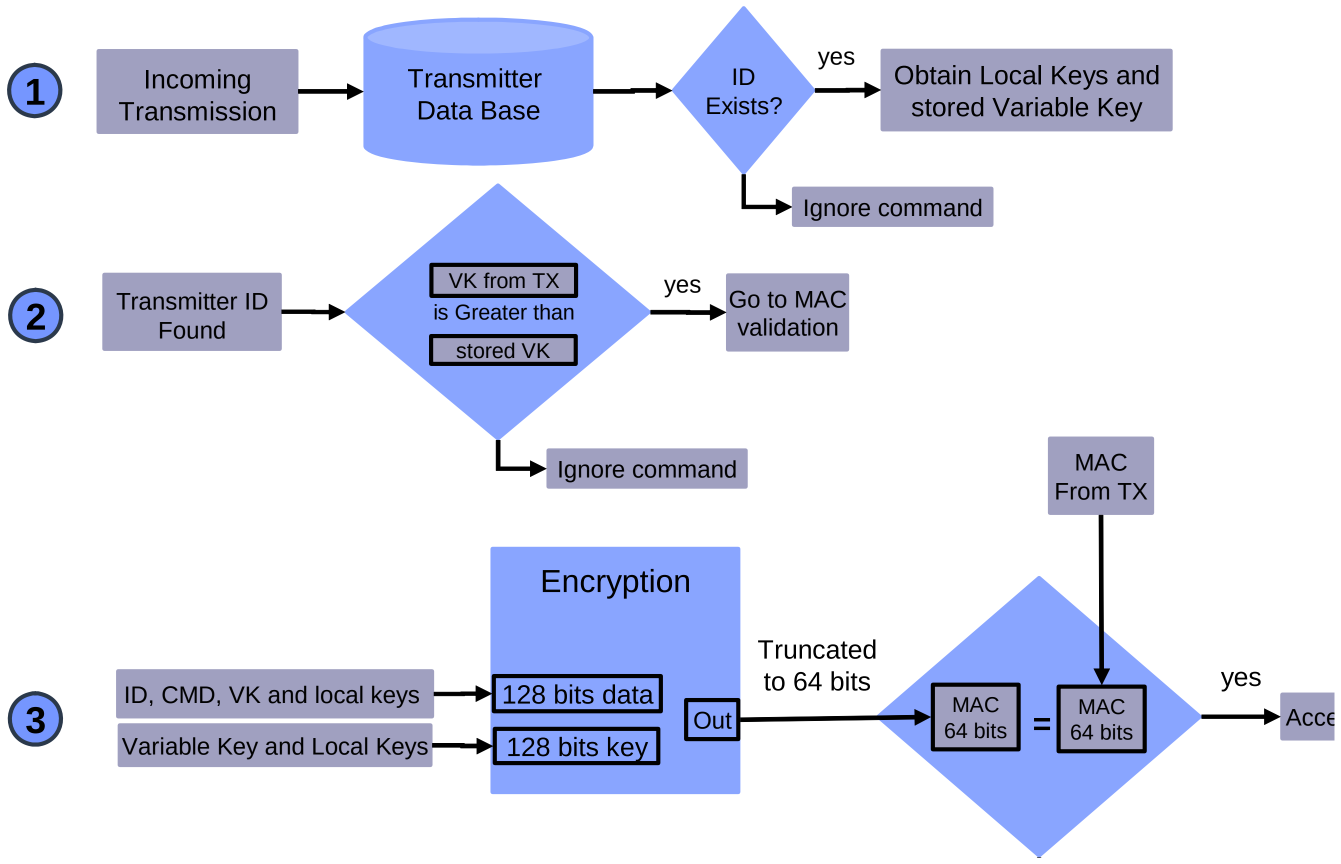


Figure 2. Normal Transmission Validation

2.1 Associating a Transmitter with a Receiver

A learning sequence is a process that must be performed each time a new transmitter is registered in the receiver system. For example, if a user of a garage door opener acquires a new transmitter key fob, it is necessary to register that key fob into the database of the receiver system. This is done by performing a learning sequence. When the receiver system receives the learning sequence, it will store the ID and local keys that pertain to that specific transmitter. Before accepting new learning sequences, the receiver system requires having a secure environment activated. The definition of a secure environment is defined by the user. A typical example of entering a secure environment is to have the final user press a button or a switch on the side of the receiver. A secure environment is set by changing the state of some input in the receiver, like a switch or a button.

Below is how a learning sequence is performed. These steps are also explained in Figure 3.

- The VKSP transmitter has a source of data from pseudo random number generator (PRNG) which is used to generate the local keys.
- Inside the integrated encoder, this pseudo random data is scrambled and two frames are generated. This information is called agreement info. The frames are depicted in Figure 4.
  - — Learning Frame 1: Contains a 3-byte header with ID information and a 1-byte field identifying the frame as a learning frame. Value 0xFE in this field indicates that this is learning frame number 1. The rest of the frame has agreement information, which is used in both the receiver

and transmitter to generate a common local key for that specific ID. The length of the learning frame body is 12 bytes.

— Learning Frame 2: Contains a 3-byte header with ID information and a 1-byte field with value 0xFF identifying the frame as learning frame number 2. The rest of the frame has agreement information (four bytes) and an 8-byte MAC.

- First, the receiver checks if there is a secure environment activated. This means the receiver must be in a mode that allows learning processes. For security reasons it is not possible to perform the learning process in the receiver unless it is in the proper mode.
- The integrated decoder receives the two frames. If the OEM key (the manufacturer key that must be shared by all authentic devices) is the same on the transmitter and receiver, the information is saved on a transmitter database for future use.

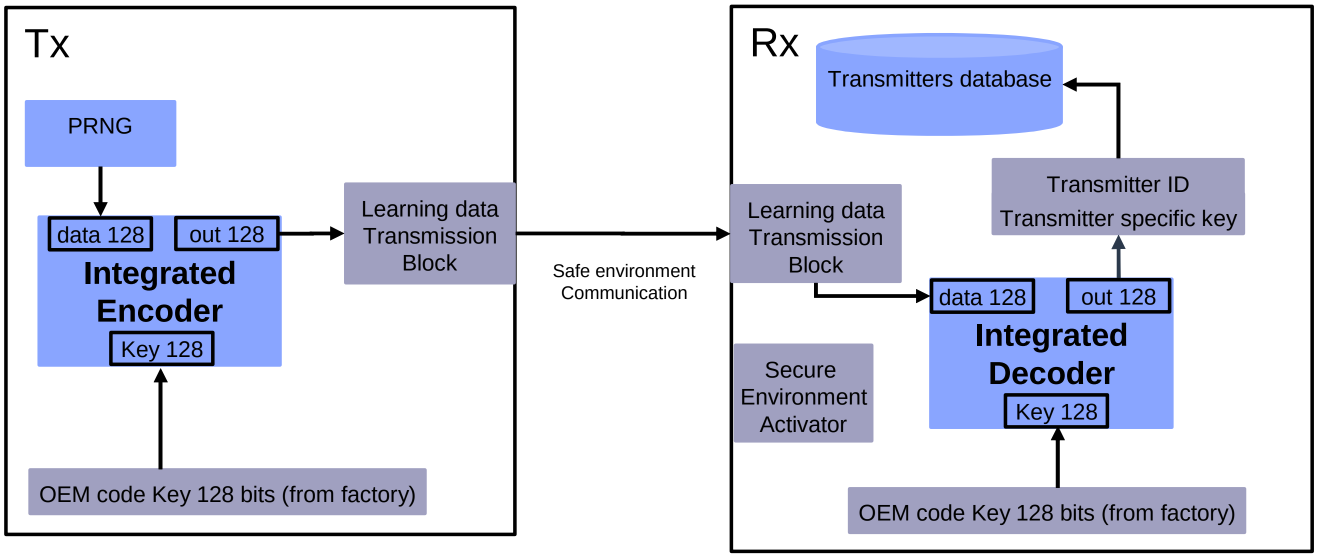

Tx
PRNG
data 128
out 128
Integrated Encoder
Key 128
Learning data Transmission Block
OEM code Key 128 bits (from factory)
Safe environment Communication
Rx
Transmitters database
Transmitter ID
Transmitter specific key
Learning data Transmission Block
data 128
out 128
Integrated Decoder
Key 128
Secure Environment Activator
OEM code Key 128 bits (from factory)

Figure 3. Learning Sequence

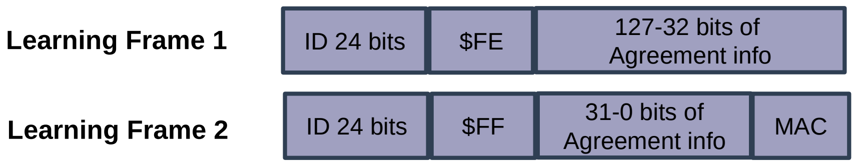


Figure 4. Learning Frames

3 VKSP Receiver Driver Overview

The VKSP receiver driver manages the VKSP frame validation. The routines are provided as state machines and can validate, configure, and retrieve information on processing results.

The driver is developed for the S08 family.

This library does not use any microcontroller hardware peripherals directly. The final application may use any drivers for encryption and nonvolatile memory access.

Figure 5 depicts the library interface and the communication with other modules.

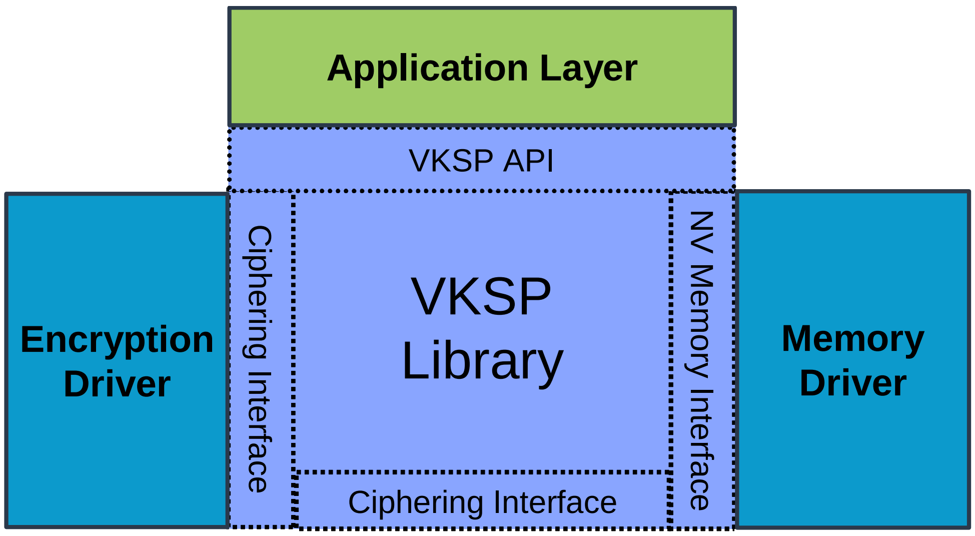


Figure 5. VKSP Receiver Block Interaction Diagram

The main block in Figure 5 is the VKSP Library block. The VKSP block interacts with other layers through an interface. These interfaces offer flexibility and allow the user to select different customized drivers.

- Ciphering Interface: used to encrypt data. Allows use of any encryption algorithm that complies with VKSP requirements.
- Nonvolatile Memory Interface: used to save nonvolatile data in memories such as flash or EEPROM.

The details about the requirements of each driver are explained in Section 5, "VKSP Receiver Application Program Interface."

The VKSP API allows the user application to initialize, validate any type of VKSP messages, and retrieve information about the received messages. The VKSP driver was designed as a state machine. It is recommended that interfaces implemented by the user be state machines as well, or at least non-blocking routines. It is possible to include blocking code in the interfaces, but it will stop the execution of the system until the routine is terminated.

4 Configuration of the VKSP Receiver Driver

There are two elements that are configurable by the user, which are presented in Table 1.

Table 1. Configuration Parameters

VKSPRX_MAX_KEYFOBS	Defines the maximum number of transmitters for the current system. The maximum possible depends on the memory interface and how it is implemented. For example: if a 512-byte memory sector is used and 19 bytes are required per transmitter, the maximum possible for that implementation is 25. Example of usage: `#define VKSPRX_MAX_KEYFOBS 8`

VKSPRX_OEM_KEY	This is the 16-byte manufacturer key which must be the same in all the devices which are intended to be used together. If a transmitter and a receiver have different OEM keys it is not possible to execute a successful learning sequence.
	
	Example of usage:
	
	`#define VKSPRX_OEM_KEY`
	`{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}`

5 VKSP Receiver Application Program Interface

Once the driver is configured and compiled, the user may use the application program interface (API) provided to integrate it in the user application. The following section details the VKSP API and the VKSP functions used to interface with the nonvolatile memory block and the encryption block.

5.1 Application Program Interface (API)

5.1.1 VkspRx_Init

Service Name:	VkspRx_Init

Syntax:	`void VkspRx_Init`
	`(`
	`    const VkspRx_ConfigType *CfgPtr`
	`)`

Parameters in:	CfgPtr: Pointer to the selected structure. This structure contains the necessary data to initialize the VKSP module. Data is taken from the configuration parameters.
Parameters out:	None
Return value:	Returns VKSPRX_STD_OFF if one of the parameters is out of range, VKSPRX_STD_ON if the initialization was successful.
Description:	This function initializes the VKSP module. The user can define the values of the structure fields by modifying the configuration parameters found in Vksp_rx_cfg.h.

5.1.2 VkspRx_StartProcessing

Service Name:	VkspRx_StartProcessing

Syntax:	`void VkspRx_StartProcessing`
	`(`
	`    VkspRx_MessageType *Message`
	`)`

Parameters in:	Message: 16-byte message received from the transmitting device.
Parameters out:	None
Return value:	None
Description:	This function starts the processing of a VKSP message. It will set the state of the driver to VKSPRX_BUSY and prepare the information so it can be processed by the VKSP receiver state machine.

5.1.3 VkspRx_ProcessMessage

Service Name:	VkspRx_ProcessMessage

Syntax:	`void VkspRx_ProcessMessage`
	`(`
	`    void`
	`)`

Parameters in:	None
Parameters out:	None
Return value:	None
Description:	This function implements the VKSP receiver state machine. It will process any incoming message triggered by the VkspRx_StartProcessing service, and it must be called cyclically by a higher level entity like a scheduler. This function will call the required interfaces to save data on nonvolatile memory and encrypt data for authentication purposes. It also terminates the job by calling VkspRx_JobEndNotification. Figure 6 and Figure 7 depict the state machine for this function.

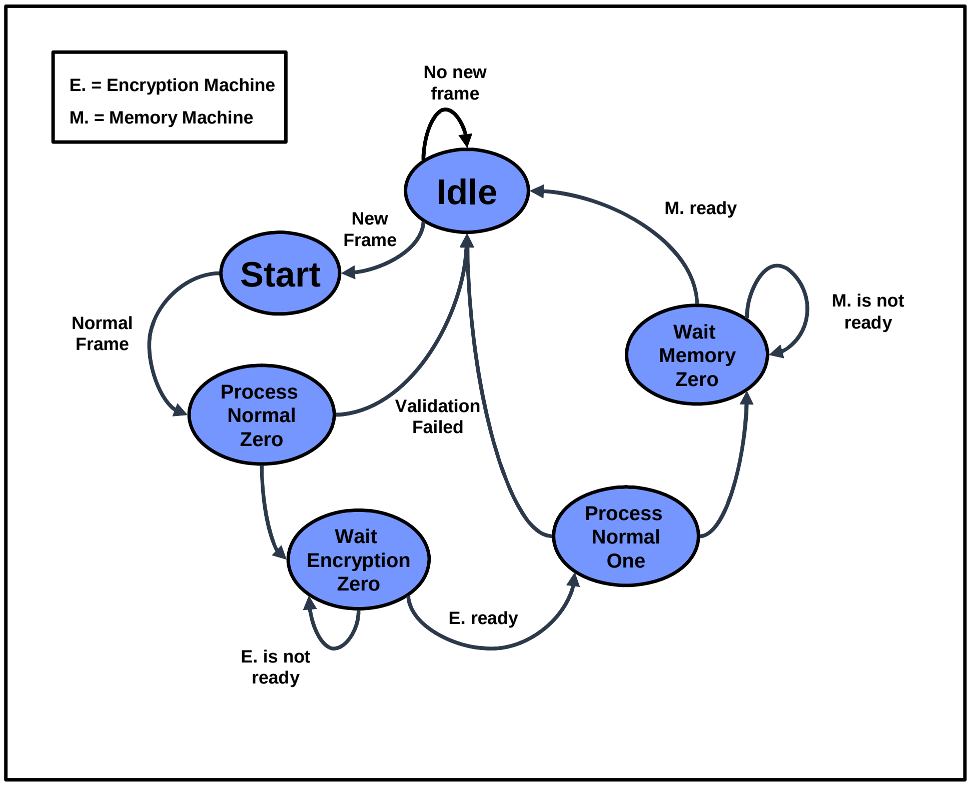


Figure 6. VkspRx_ProcessMessage State Machine (Normal Validation)

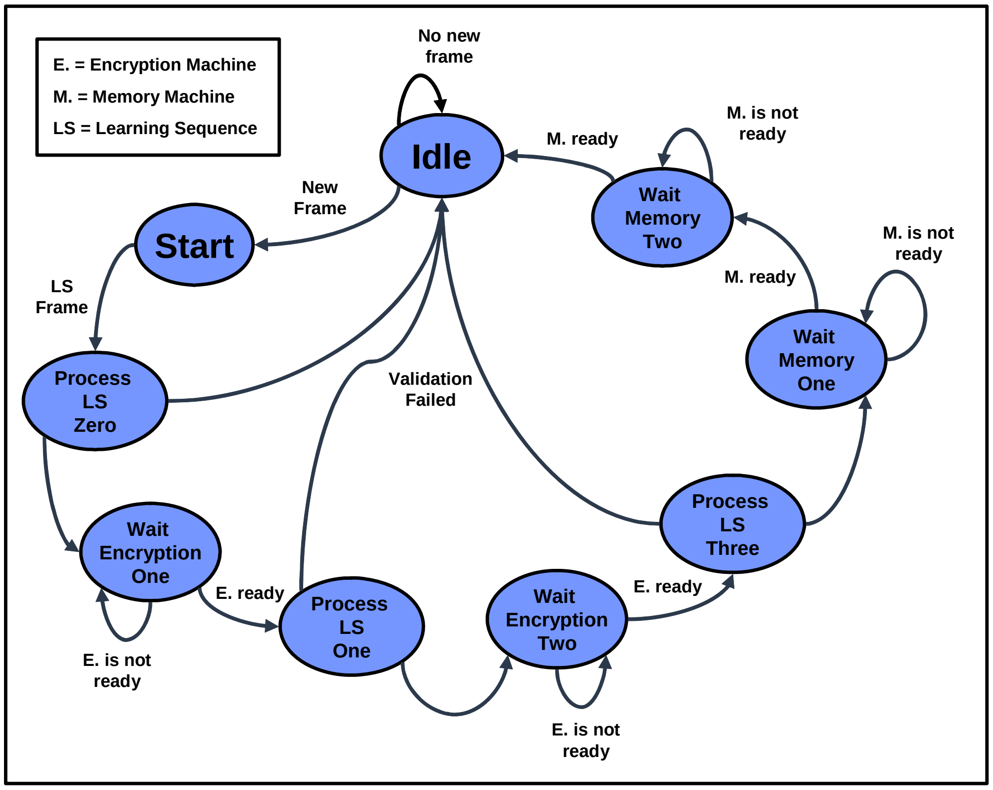


Figure 7. VkspRx_ProcessMessage State Machine (Learning Sequence Validation)

5.1.4 VkspRx_GetCommand

Service Name:	VkspRx_GetCommand

Syntax:	`VkspRx_CommandType VkspRx_GetCommand`
	`(`
	`    void`
	`)`

Parameters in:	None
Parameters out:	None
Return value:	Returns the received command.
Description:	This function returns the value of the last received command.

5.1.5 VkspRx_GetJobResult

Service Name:	VkspRx_GetJobResult

Syntax:	`VkspRx_JobResultType VkspRx_GetJobResult`
	`(`
	`    void`
	`)`

Parameters in:	None
Parameters out:	None
Return value:	Indicates the result of the last processed job.
Description:	This function returns the result of the last processed message. The possible results are:

VKSPRX_MSG_OK	0x00	The message received is valid.
VKSPRX_MSG_LS1	0x01	The message received is the first frame of a learning sequence.
VKSPRX_MSG_LS_OK	0x02	The two received frames from the learning sequence are valid.
VKSPRX_MSG_LS_MAC_FAILED	0x03	The learning sequence frames are not valid.
VKSPRX_MSG_NML_MAC_FAILED	0x04	The message received is not valid.
VKSPRX_MSG_INSECURE_ENVMT	0x05	There is not a secure environment to perform a learning sequence.
VKSPRX_MSG_INVALID_CNT	0x06	The Variable Key is not valid.
VKSPRX_MSG_INVALID_ID	0x07	The ID received is not registered in the memory.
VKSPRX_MSG_MEM_FULL	0x08	The memory is full; no more transmitters can be registered.
VKSPRX_MSG_MEMORYFAIL	0x0A	The VKSP core memory task was interrupted and could not be performed.

5.2 VKSP Core Interfaces

The VKSP core interfaces are used by the VKSP state machine. These functions must be implemented by the user. The description for each function represents the features that the user has to implement.

5.2.1 VkspRx_SecureEnvironment

Service Name:	VkspRx_SecureEnvironment
Syntax:	`VkspRx_SecureEnvironment (     void )`
Parameters in:	None
Parameters out:	None
Return value:	Returns VKSPRX_STD_OFF if there is not a secure environment. VKSPRX_STD_ON if there is a secure environment.
Description:	This function tells the VKSP driver if there is a secure environment or not. (The implementation of this function is left open for the user to meet the needs of his application.) The return result tells the VKSP state machine how to proceed. If the result is positive, it is possible to perform a learning sequence; otherwise it is not. A common implementation would be to check the state of a pin.

5.2.2 VkspRx_JobEndNotification

Service Name:	VkspRx_JobEndNotification
Syntax:	`VkspRx_JobEndNotification (     void )`
Parameters in:	None
Parameters out:	None
Return value:	None
Description:	This function is called after the termination of a processing job by the service VkspRx_ProcessMessage. The user should insert the action to be performed inside the body of this function.

5.2.3 VkspRx_NotificationVar

Service Name:	VkspRx_NotificationVar
Syntax:	`VkspRx_NotificationVarType VkspRx_NotificationVar;`
Parameters in:	N/A
Parameters out:	N/A
Return value:	N/A
Description:	This is an interface variable. When an encryption or a memory job is about to be processed the value of this variable is set to VKSPRX_NOTIFICATION_BUSY by the VKSP state machine. User must make sure that in the implementation of the interface, this variable is set to VKSPRX_NOTIFICATION_DONE when the current encryption or memory job is finished. A memory job can be cancelled by changing this variable to VKSPRX_NOTIFICATION_CANCEL value while the VKSP state machine is waiting for the done notification. This variable is monitored by the VKSP state machine after calling the following services: VkspRx_SaveLcode, VkspRx_SaveCounter and VkspRx_StartEncryption.

5.3 Memory Interface

The VKSP memory interface functions are called by the VKSP state machine. They are located in the MemIF.c file with the example code provided. This code can be changed to fit the requirements of the final application. The description for each function represents the features that the user has to implement.

5.3.1 VkspRx_SaveLcode

Service Name:	VkspRx_SaveLcode

Syntax:	`VkspRx_LCodeId VkspRx_SaveLcode`
	`(`
	`    VkspRx_DataType *Source`
	`)`

Parameters in:	Source: Pointer to a 19-byte array to be saved in memory.
Parameters out:	None
Return value:	None
Description:	This is the interface function to back up nonvolatile values. There are several possible implementations and they are up to the user, depending on the memory driver used. This function must return the index of the register used to save the learning code. The index is used later with the interface function VkspRx_ReadLcode as argument to retrieve a specific learning code. The leraning code is saved as shown in Figure 8.

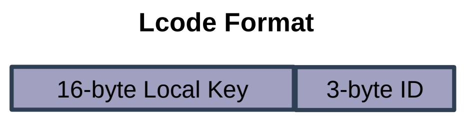


Figure 8. Learning Code Format

5.3.2 VkspRx_ReadLcode

Service Name:	VkspRx_ReadLcode

Syntax:	`void VkspRx_ReadLcode`
	`(`
	`    VkspRx_LCodeId LcodeId,`
	`    VkspRx_DataType **Target`
	`)`

Parameters in:	LCodeId: Index of learning code to be read. This value is usually obtained by calling VkspRx_SaveLcode or VkspRx_SearchID. Target: Address of the 19-byte array where the data should be stored.
Parameters out:	None
Return value:	None
Description:	This is the interface function to read backed-up nonvolatile values of the learning codes. It should be non-blocking and implemented by the user. The most common approach would be to read directly from RAM or EEPROM.

5.3.3 VkspRx_SaveCounter

Service Name:	VkspRx_SaveCounter

Syntax:	`void VkspRx_SaveCounter`
	`(`
	`    VkspRx_CounterId CounterId,`
	`    VkspRx_DataType *Source`
	`)`

Parameters in:	CounterId: Index of counter to be saved. This is the same index used for the learning codes. This value is usually obtained by calling VkspRx_SaveLcode or VkspRx_SearchID. Source: Pointer to a 4-byte array to save to memory.
Parameters out:	None
Return value:	None
Description:	This is the interface function to back up nonvolatile values. There are several possible implementations and they are up to the user, depending on the used memory driver. This function must overwrite the old value for the specific CounterId used and replace it with the received Source parameter.

5.3.4 VkspRx_ReadCounter

Service Name:	VkspRx_ReadCounter

Syntax:	`void VkspRx_ReadCounter`
	`(`
	`    VkspRx_CounterId CounterId,`
	`    VkspRx_DataType *Source`
	`)`

Parameters in:	CounterId: Index of the counter to be read. It is the same index used for the learning codes. This value is usually obtained by using VkspRx_SaveLcode or VkspRx_SearchID. Source: This is the address of the 4-byte array where the data should be read.
Parameters out:	None
Return value:	None
Description:	This is the interface function to read backed-up nonvolatile values of the counters. It should be non-blocking and implemented by the user. The most common approach would be to read directly from RAM or EEPROM.

5.3.5 VkspRx_SearchID

Service Name:	VkspRx_SearchID

Syntax:	`VkspRx_LCodeId VkspRx_SearchID`
	`(`
	`    VkspRx_DataType *ID`
	`)`

Parameters in:	ID: Pointer to a 3-byte array that indicates what VKSP ID to search for.
Parameters out:	None
Return value:	Returns the index of the register which corresponds to the input ID.
Description:	This function will be used to search for the Transmitter ID in the memory database. If the ID is not found it must return MEMIF_BLANK. The function must return the newest learning code that matches with the input ID. For example: if two learning sequences are received from the same transmitter ID, the last one received is the one that must be used.

5.4 Encryption Interface

The VKSP encryption interface functions are called by the VKSP state machine. They are located in the CipherIF.c file with the example code provided. This code can be changed to fit the requirements of the final application. The description for each function represents the features that the user has to implement.

5.4.1 VkspRx_StartEncryption

Service Name:	VkspRx_StartEncryption

Syntax:	`VkspRx_StdType VkspRx_StartEncryption`
	`(`
	`    VkspRx_DataType *Data,`
	`    VkspRx_DataType *Key,`
	`    VkspRx_DataType *Result`
	`)`

Parameters in:	Data: 16-byte array containing the data to be encrypted. Key: 16-byte array containing the key to be used for encryption.
Parameters out:	Result: 16-byte array that will hold the result of the encryption.
Return value:	Indicates if the encryption module is ready to perform an encryption job.
Description:	This is the interface function that starts the encryption job for the VKSP core state machine. Depending on the implementation of the encryption state machine, it could be designed to be shared with different devices. If that is the case, the interface must indicate to the VKSP core state machine if the encryption machine is ready to accept an encryption job.

5.4.2 VkspRx_Encrypt

Service Name:	VkspRx_Encrypt

Syntax:	`void VkspRx_Encrypt`
	`(`
	`    void`
	`)`

Parameters in:	None
Parameters out:	None
Return value:	None
Description:	This interface function is called cyclically to verify if the encryption job is finished. When the current job is terminated this interface function must set VkspRx_NotificationVar to VKSPRX_ENCRYPTION_DONE**.** This flag will be monitored by the VKSP core state machine.

6 VKSP Receiver Driver Implementation

The following sections explain how to set up an RKE receiver application from the beginning. The application is designed in a layered approach that facilitates migration to different microcontrollers.

6.1 RKE Receiver Overview

To build an RKE receiver application, it is necessary to implement other modules aside from the VKSP receiver core. It is important to understand what functionalities are handled by the VKSP receiver core and what functionalities are not.

What the VKSP receiver core driver does:

- Validate ID, variable key, and MAC from incoming messages.
- Provide results of the processing jobs.
- Request access to the memory and to the encryption driver.

What the VKSP receiver core driver does not do:

- Encrypt.
- Use MCU resources directly.
- Determine how the messages are received by the physical layer.
- Write into nonvolatile memory

The present VKSP receiver driver and the RKE application were designed using a software architecture with the purpose of having a layered software design and easing the migration to different platforms.

The software architecture used divides software into different layers. From a high level perspective, the software can be described as follows:

- Hardware Abstraction Layer: Provides drivers to control MCU peripherals needed for the RKE application, such as timers, GPIO control, and SPI.
- Hardware Independent Layer: Provides drivers dependent on the electronic module used but independent of the hardware used. Examples of these drivers are: The echo receiver driver, the signal abstraction driver and the VKSP driver.
- Services Layer: Drivers contained in this layer are often used by other layers. Examples: scheduler, MCU configuration, etc.
- Application Layer: This layer implements the application, then initializes and integrates the rest of the drivers.

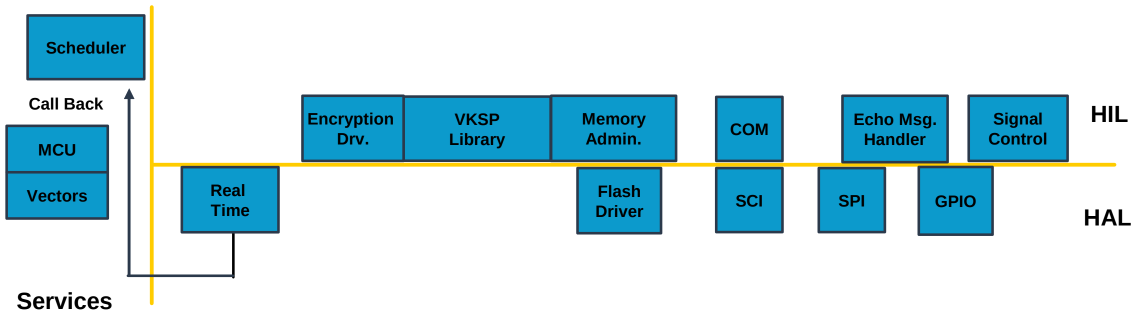


Figure 9. RKE Receiver Layer Structure

6.2 Components Used

The receiver device provides the following functionality:

- Waits for messages that are received through the RF link
- Validates messages
- Proivides results from received messages
- Performs actions based on the received messages

Below is a brief description of the software components and their functionality.

6.2.1 HAL Drivers

- Real Time: This driver configures the Real Time Clock or Real Time Interrupt peripheral, depending on the MCU used. It implements a callback and an interrupt flag.
- Timer: This driver implements the software to control the TPM peripheral. It implements a callback and an interrupt flag. It is also possible to set the interrupt frequency based on the BUS frequency.
- Flash Driver: This driver implements control functions to write and erase flash memory. The functions are copied and executed from RAM.
- GPIO: Abstracts the microcontroller pins to virtual ports. It also implements a function to set the ports into low power modes.
- SPI: Implements simple control for the SPI peripheral. This driver implements interrupts with callbacks when working as a slave device.
- SCI: Configures the SCI peripheral. The baud rate is selectable in the configuration parameters.

6.2.2 HIL Drivers

- Signals: Abstracts the virtual ports to named signals from the electronic module. For example, it allows access to signals as buttons or LEDs. It uses the GPIO driver.
- EchoMsgHandler: This handler uses the signals and the SPI modules to control the echo transceiver. It is specifically done to receive fixed length messages. Length is configurable by the user.
- COM: This driver allows the application layer to access the SCI driver.
- AesSt: Implements the AES encryption module using state machines. This driver does not use any of the HAL drivers.
- VKSP Receiver: Driver that implements the VKSP for the receiver. Requires some files to interface with other modules, like the flash and the encryption module. It implements state machines.

6.2.3 Services Drivers

- Common: Contains declarations of common types used by all other modules.
- MCU: Implements miscellaneous routines and macros like initialization, access to assembler instructions, and public parameters such as the bus frequency.
- Scheduler: Implements the usage of timer drivers, allowing the division of processing power into time slices. Inside those slices different application tasks can be added.
- Vectors: Defines the interrupt locations in program memory.

6.2.4 Application

This layer integrates the HIL and services drivers implementing the following features:

- Initializes the MCU, Signals, Memory, VKSP, EchoMsgHandler, Communications and Scheduler drivers.
- Enables/disables the interrupts.
- Services the state machines for VKSP and encryption drivers.
- Checks for messages ready in the EchoMsgHandler.
- Outputs results from processed jobs.

6.2.5 Linker Parameters

The VKSP library requires saving nonvolatile data. This storage space must be reserved. This is done by creating a specific section for this purpose in the linker parameter file. The memory interface is open and implementation is up to the user, so the user must ensure that this storage space is reserved and not used for program code.

6.3 Memory Interface Implementation

Memory interface for VKSP is open and flexible so it can be adapted to user needs.

Each VKSP transmitter used in the application requires space to store two variables in the receiver system:

- Learning code (19 bytes)
- Variable key (4 bytes)

The learning code (Lcode) is a variable that includes information about the ID and local keys of the device. Each Lcode is related to a variable key, in the same way that relational databases have related tables with an index field.

Depending on the nonvolatile memory capabilities, different strategies may be used to save into nonvolatile space. If the receiver microcontroller has internal EEPROM it is possible to reserve space for each variable; when the variable key needs to be updated, the memory location is erased and programmed individually.

If the microcontroller only has flash memory as a nonvolatile resource, the strategy to follow is quite more complex. In this case, the memory driver associates index numbers to each copy of the variable key and learning code in order to keep track of the newest values.

These two approaches are explained in the following sections.

6.3.1 EEPROM Memory Model

The following section explains how to implement a method to save the values of the Lcodes and the variable keys on EEPROM memory. This method is reliable and simple. The drawback is that it has a limited lifetime, because every time the values are updated an erase action must be performed. This method can be used as well with EEPROM emulation in order to have increased memory endurance.

The size of the memory sector used in this example is four bytes. For each variable it is necessary to allocate a space that is a multiple of the sector size. For each learning code a 5-sector space is reserved, but for each counter one sector is enough. There is a parameter in the VKSP configuration file called

VKSPRX_MAXKEYFOBS that indicates the maximum number of transmitters for the application. Use that parameter to calculate the size for the storage of the learning codes and the variables keys.

Figure 10 describes the allocation of variables in EEPROM memory. To save or retrieve information, an index is used to locate the address in memory that must be erased and written over.

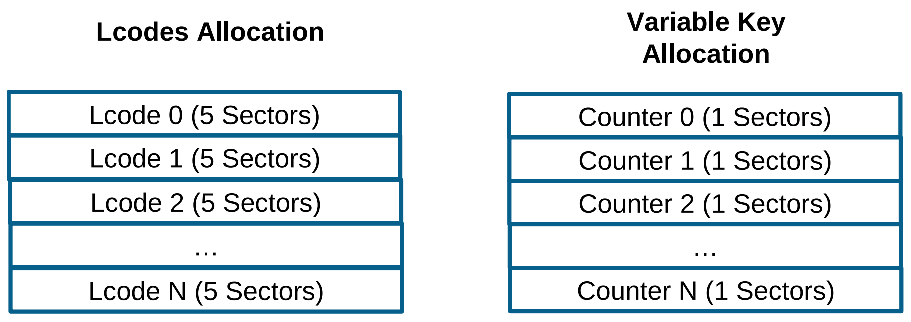


Figure 10. EEPROM Memory Model

6.3.2 Flash Memory Model

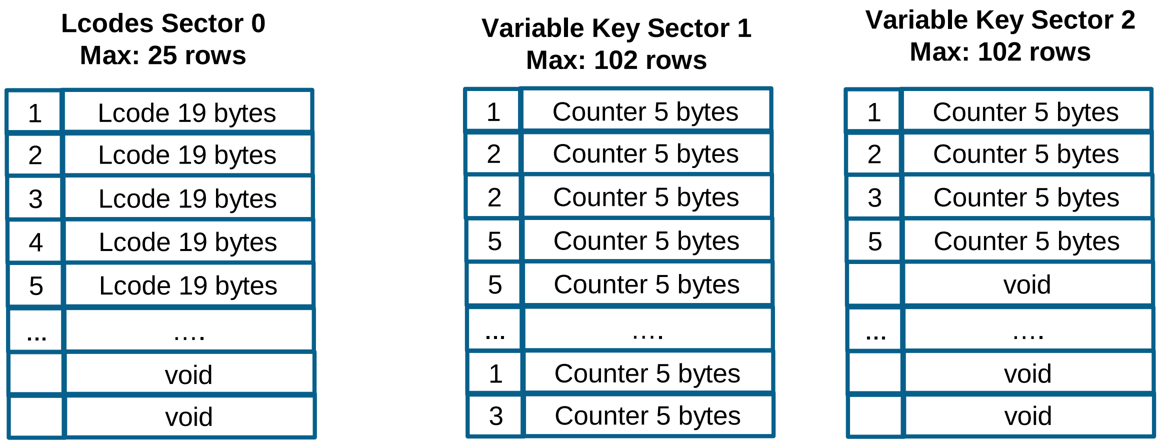
The following section explains how to implement a method to save the values of the learning codes and the variable keys in flash memory. This method provides endurance but requires more processing.

The size of the memory sector used in this example is 512 bytes, a typical value for a flash sector. In this case, one sector of memory is used for the storage of learning codes and two sectors are used for the variable keys. Also, for each learning code and variable key, a 1-byte index is added. This index makes it possible to identify them in the flash database. This is useful for recovering the latest values after a power-up event.

The index must be administered by the user code in the memory interface. For example, each time a successful learning sequence is received, the function VkspRx_SaveLcode will be called. This function will receive the Lcode as an argument, and the code within the function must store it in memory and assign a new index for that Lcode. If the memory is full, depending on the implementation, the function VkspRx_SaveLcode can return the value indicating that memory is full or can replace the oldest Lcode with the newest one and continue operating normally.

The learning codes are saved into memory and assigned an index in a sequential manner, as new transmitters are registered into the database. Each time a successful normal transmission is received, a variable key with the transmitter index number is stored in a new location in the variable key memory sector.

This is depicted in Figure 11. If there are several variable keys with the same index, the last inserted is the valid one. When one variable key sector gets full, the other one is filled with the last received values and marked as the active sector. Once this task is done, it is possible to erase the sector that is full.



Lcodes Sector 0
Max: 25 rows

1	Lcode 19 bytes
2	Lcode 19 bytes
3	Lcode 19 bytes
4	Lcode 19 bytes
5	Lcode 19 bytes
...	….
	void
	void

Variable Key Sector 1
Max: 102 rows

1	Counter 5 bytes
2	Counter 5 bytes
2	Counter 5 bytes
5	Counter 5 bytes
5	Counter 5 bytes
...	….
1	Counter 5 bytes
3	Counter 5 bytes

Variable Key Sector 2
Max: 102 rows

1	Counter 5 bytes
2	Counter 5 bytes
3	Counter 5 bytes
5	Counter 5 bytes
	void
...	….
	void
	void

Figure 11. Flash Memory Model

There are several variations in this implementation to improve code size or execution time. For example, it is possible to keep a local copy of the most recent variable key counter in RAM for faster access, instead of looking for the values in the database.

6.4 Encryption Interface Implementation

VKSP allows the user to choose an encryption protocol for the generation of MACs. The call to the encryption driver must be located in the encryption interface. The interface requires the usage of a 128-bit block size encryption algorithm with a 128-bit key. Some encryption algorithms that do not comply with this requirement can be adapted so they can be used. Figure 12 depicts the adaptation of a hypothetical algorithm.

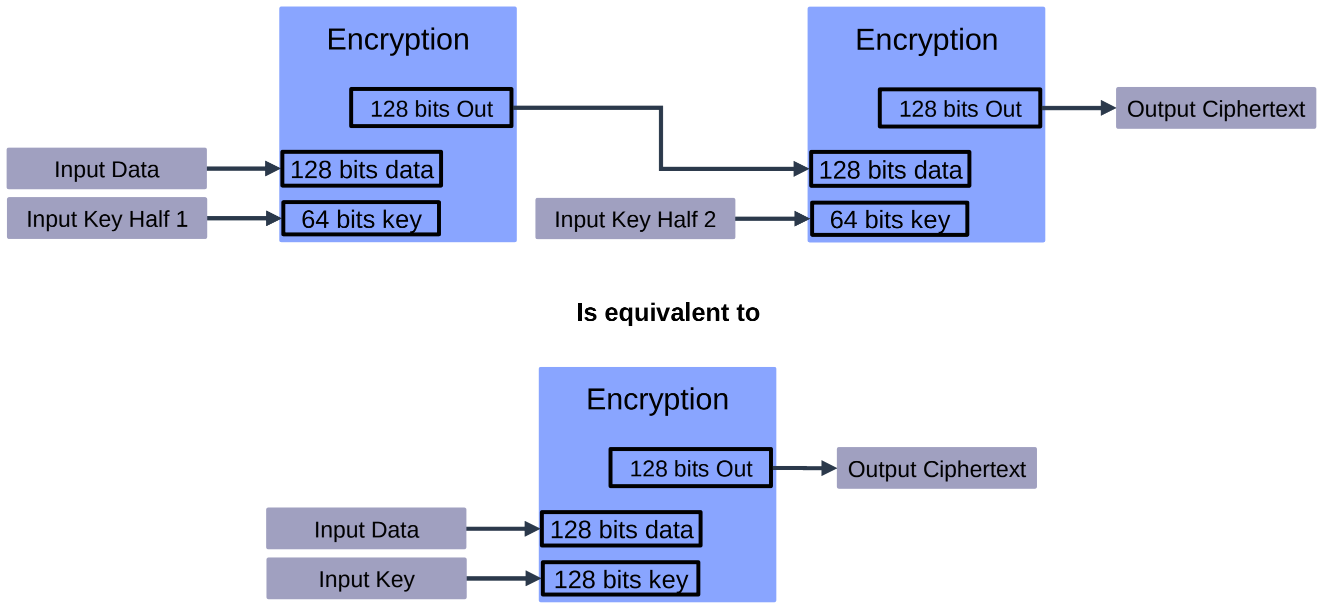


Figure 12. Encryption Adaptation to VKSP Requirements

In the application provided, the Advanced Encryption Standard (AES) was chosen because it complies directly with the input/output requirements.

7 Step-by-Step Setup Guide of a VKSP Receiver Application

This section indicates how to set up a receiver application. Before implementing the steps below you must have:

- A driver to save and read values into nonvolatile memory
- An encryption driver
- A scheduler.

In the following steps the term “if required” is used, this means it is possible to change the code provided in the sample application if the user requires it.

1. Add the following files to your project: the vksp_rx.obj core file, the VKSP configuration files (vksp_rx_cfg.h, vksp_rx_cfg.c) and the interface files (MemIf.c, CipherIf.c).
2. Modify the configuration parameters as needed. (Refer to Section 4, “Configuration of the VKSP Receiver Driver.”)
3. Modify code in the encryption interface if required. In VkspRx_StartEncryption add the code that prepares the data to be encrypted. In VkspRx_Encrypt insert the call to the encryption state machine used. When the task is done the code must set the notification variable to VKSPRX_NOTIFICATION_DONE.
4. Modify code in the memory interface, if required. When a save task is finished, the user must ensure that the notification variable is set to VKSPRX_NOTIFICATION_DONE.
   a) VkspRx_SaveCounter: The user must write code to save the 4-byte counter or variable key in memory.
   b) VkspRx_SaveLcode: The user must write code to save the 19-byte Lcode in memory.
   c) VkspRx_ReadCounter: Insert code that returns the address of the first byte where the requested counter is located.
   d) VkspRx_ReadLcode: Insert code that returns the address of the first byte where the requested Lcode is located.
   e) VkspRx_SearchID: Insert code to search for a match with the received 3-byte ID within the database. The ID in the learning code array is located at byte number 16, 17, and 18. If there is more than one match, the newest register must be returned.
5. Implement the function VkspRx_SecureEnvironment. This function will be called by the VKSP receiver core if a learning sequence arrives. The user code will tell the VKSP receiver core if there is a secure environment or not.
6. Implement the function VkspRx_JobEndNotification. This function will be called by the VKSP receiver core after processing a VKSP frame. It is expected you will have code here that will react to the received commands when they are authentic.
7. In the main application: Initialize nonvolatile memory.
8. Initialize the VKSP driver by calling VkspRx_Init.
9. Make sure VkspRx_ProcessMessage and VkspRx_Encrypt are called periodically by the scheduler. VkspRx_Encrypt must be called at a higher rate than VkspRx_ProcessMessage because the encryption is used by the VKSP driver.

10. Every time a frame is received by the RF link, trigger the start of the processing by calling VkspRx_StartProcessing.

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor Hong Kong Ltd.
Technical Information Center
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
+800 2666 8080
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: VKSPRXUG
Rev. 0
06/2008