AES source code

Advanced Encryption Standard

The source code for the AES algorithm, also called Advanced Encryption Standard or the Rijndael algorithm. This source code is part of the mbed TLS library and represents the most current version in the trunk of the library.

The full algorithm of AES is further explained in AES algorithm (Wikipedia).

The code has a dependency on *config.h* in the aes.c source code file. You can remove this inclusion or just create a simple header file to define one or more of the configuration options that the AES source code has. In addition a dependency on *padlock.h* and *padlock.c* is present if you have **POLARSSL_PADLOCK_C** defined, and a dependency on *aesni.h* and *aesni.c* is present if you have **POLARSSL_AESNI_C** defined.

Full documentation on the AES source code can be found in the API documentation for the AES module.

You can also download it as part of the latest release of mbed TLS.

Header - aes.h

The aes.h header can also be found in the trunk on: aes.h.

```
/**
 * \file aes.h
 *
 * \brief   This file contains AES definitions and functions.
 *
 *          The Advanced Encryption Standard (AES) specifies a FIPS-approved
 *          cryptographic algorithm that can be used to protect electronic
 *          data.
 *
 *          The AES algorithm is a symmetric block cipher that can
 *          encrypt and decrypt information. For more information, see
 *          <em>FIPS Publication 197: Advanced Encryption Standard</em> and
 *          <em>ISO/IEC 18033-2:2006: Information technology -- Security
 *          techniques -- Encryption algorithms -- Part 2: Asymmetric
 *          ciphers</em>.
 *
 *          The AES-XTS block mode is standardized by NIST SP 800-38E
 *          <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38e.pdf>
 *          and described in detail by IEEE P1619
 *          <https://ieeexplore.ieee.org/servlet/opac?punumber=4375278>.
 */

/*  Copyright (C) 2006-2018, Arm Limited (or its affiliates), All Rights Reserved.
 *  SPDX-License-Identifier: Apache-2.0
 *
 *  Licensed under the Apache License, Version 2.0 (the "License"); you may
 *  not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
```

```
 *  distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
 *  WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 *
 *  This file is part of Mbed TLS (https://tls.mbed.org)
 */

#ifndef MBEDTLS_AES_H
#define MBEDTLS_AES_H

#if !defined(MBEDTLS_CONFIG_FILE)
#include "config.h"
#else
#include MBEDTLS_CONFIG_FILE
#endif

#include <stddef.h>
#include <stdint.h>

/* padlock.c and aesni.c rely on these values! */
#define MBEDTLS_AES_ENCRYPT     1 /**< AES encryption. */
#define MBEDTLS_AES_DECRYPT     0 /**< AES decryption. */

/* Error codes in range 0x0020-0x0022 */
#define MBEDTLS_ERR_AES_INVALID_KEY_LENGTH                -0x0020  /**< Invalid key length. */
#define MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH              -0x0022  /**< Invalid data input lengt
h. */

/* Error codes in range 0x0021-0x0025 */
#define MBEDTLS_ERR_AES_BAD_INPUT_DATA                    -0x0021  /**< Invalid input data. */

/* MBEDTLS_ERR_AES_FEATURE_UNAVAILABLE is deprecated and should not be used. */
#define MBEDTLS_ERR_AES_FEATURE_UNAVAILABLE               -0x0023  /**< Feature not available. Fo
r example, an unsupported AES key size. */

/* MBEDTLS_ERR_AES_HW_ACCEL_FAILED is deprecated and should not be used. */
#define MBEDTLS_ERR_AES_HW_ACCEL_FAILED                   -0x0025  /**< AES hardware accelerator
 failed. */

#if ( defined(__ARMCC_VERSION) || defined(_MSC_VER) ) && \
    !defined(inline) && !defined(__cplusplus)
#define inline __inline
#endif

#ifdef __cplusplus
extern "C" {
#endif

#if !defined(MBEDTLS_AES_ALT)
// Regular implementation
//

/**
 * \brief The AES context-type definition.
 */
typedef struct mbedtls_aes_context
{
    int nr;                     /*!< The number of rounds. */
    uint32_t *rk;               /*!< AES round keys. */
    uint32_t buf[68];           /*!< Unaligned data buffer. This buffer can
                                     hold 32 extra Bytes, which can be used for
                                     one of the following purposes:
                                     <ul><li>Alignment if VIA padlock is
                                             used.</li>
                                     <li>Simplifying key expansion in the 256-bit
                                         case by generating an extra round key.
                                         </li></ul> */
}
```

```
mbedtls_aes_context;

#if defined(MBEDTLS_CIPHER_MODE_XTS)
/**
 * \brief The AES XTS context-type definition.
 */
typedef struct mbedtls_aes_xts_context
{
    mbedtls_aes_context crypt; /*!< The AES context to use for AES block
                                    encryption or decryption. */
    mbedtls_aes_context tweak; /*!< The AES context used for tweak
                                    computation. */
} mbedtls_aes_xts_context;
#endif /* MBEDTLS_CIPHER_MODE_XTS */

#else  /* MBEDTLS_AES_ALT */
#include "aes_alt.h"
#endif /* MBEDTLS_AES_ALT */

/**
 * \brief          This function initializes the specified AES context.
 *
 *                 It must be the first API called before using
 *                 the context.
 *
 * \param ctx      The AES context to initialize.
 */
void mbedtls_aes_init( mbedtls_aes_context *ctx );

/**
 * \brief          This function releases and clears the specified AES context.
 *
 * \param ctx      The AES context to clear.
 */
void mbedtls_aes_free( mbedtls_aes_context *ctx );

#if defined(MBEDTLS_CIPHER_MODE_XTS)
/**
 * \brief          This function initializes the specified AES XTS context.
 *
 *                 It must be the first API called before using
 *                 the context.
 *
 * \param ctx      The AES XTS context to initialize.
 */
void mbedtls_aes_xts_init( mbedtls_aes_xts_context *ctx );

/**
 * \brief          This function releases and clears the specified AES XTS context.
 *
 * \param ctx      The AES XTS context to clear.
 */
void mbedtls_aes_xts_free( mbedtls_aes_xts_context *ctx );
#endif /* MBEDTLS_CIPHER_MODE_XTS */

/**
 * \brief          This function sets the encryption key.
 *
 * \param ctx      The AES context to which the key should be bound.
 * \param key      The encryption key.
 * \param keybits  The size of data passed in bits. Valid options are:
 *                 <ul><li>128 bits</li>
 *                 <li>192 bits</li>
 *                 <li>256 bits</li></ul>
 *
 * \return         \c 0 on success.
 * \return         #MBEDTLS_ERR_AES_INVALID_KEY_LENGTH on failure.
 */
int mbedtls_aes_setkey_enc( mbedtls_aes_context *ctx, const unsigned char *key,
```

```
                    unsigned int keybits );

/**
 * \brief          This function sets the decryption key.
 *
 * \param ctx      The AES context to which the key should be bound.
 * \param key      The decryption key.
 * \param keybits  The size of data passed. Valid options are:
 *                 <ul><li>128 bits</li>
 *                 <li>192 bits</li>
 *                 <li>256 bits</li></ul>
 *
 * \return         \c 0 on success.
 * \return         #MBEDTLS_ERR_AES_INVALID_KEY_LENGTH on failure.
 */
int mbedtls_aes_setkey_dec( mbedtls_aes_context *ctx, const unsigned char *key,
                    unsigned int keybits );

#if defined(MBEDTLS_CIPHER_MODE_XTS)
/**
 * \brief          This function prepares an XTS context for encryption and
 *                 sets the encryption key.
 *
 * \param ctx      The AES XTS context to which the key should be bound.
 * \param key      The encryption key. This is comprised of the XTS key1
 *                 concatenated with the XTS key2.
 * \param keybits  The size of \p key passed in bits. Valid options are:
 *                 <ul><li>256 bits (each of key1 and key2 is a 128-bit key)</li>
 *                 <li>512 bits (each of key1 and key2 is a 256-bit key)</li></ul>
 *
 * \return         \c 0 on success.
 * \return         #MBEDTLS_ERR_AES_INVALID_KEY_LENGTH on failure.
 */
int mbedtls_aes_xts_setkey_enc( mbedtls_aes_xts_context *ctx,
                                const unsigned char *key,
                                unsigned int keybits );

/**
 * \brief          This function prepares an XTS context for decryption and
 *                 sets the decryption key.
 *
 * \param ctx      The AES XTS context to which the key should be bound.
 * \param key      The decryption key. This is comprised of the XTS key1
 *                 concatenated with the XTS key2.
 * \param keybits  The size of \p key passed in bits. Valid options are:
 *                 <ul><li>256 bits (each of key1 and key2 is a 128-bit key)</li>
 *                 <li>512 bits (each of key1 and key2 is a 256-bit key)</li></ul>
 *
 * \return         \c 0 on success.
 * \return         #MBEDTLS_ERR_AES_INVALID_KEY_LENGTH on failure.
 */
int mbedtls_aes_xts_setkey_dec( mbedtls_aes_xts_context *ctx,
                                const unsigned char *key,
                                unsigned int keybits );
#endif /* MBEDTLS_CIPHER_MODE_XTS */

/**
 * \brief          This function performs an AES single-block encryption or
 *                 decryption operation.
 *
 *                 It performs the operation defined in the \p mode parameter
 *                 (encrypt or decrypt), on the input data buffer defined in
 *                 the \p input parameter.
 *
 *                 mbedtls_aes_init(), and either mbedtls_aes_setkey_enc() or
 *                 mbedtls_aes_setkey_dec() must be called before the first
 *                 call to this API with the same context.
 *
 * \param ctx      The AES context to use for encryption or decryption.
```

```
 * \param mode     The AES operation: #MBEDTLS_AES_ENCRYPT or
 *                 #MBEDTLS_AES_DECRYPT.
 * \param input    The 16-Byte buffer holding the input data.
 * \param output   The 16-Byte buffer holding the output data.

 * \return         \c 0 on success.
 */
int mbedtls_aes_crypt_ecb( mbedtls_aes_context *ctx,
                    int mode,
                    const unsigned char input[16],
                    unsigned char output[16] );

#if defined(MBEDTLS_CIPHER_MODE_CBC)
/**
 * \brief  This function performs an AES-CBC encryption or decryption operation
 *         on full blocks.
 *
 *         It performs the operation defined in the \p mode
 *         parameter (encrypt/decrypt), on the input data buffer defined in
 *         the \p input parameter.
 *
 *         It can be called as many times as needed, until all the input
 *         data is processed. mbedtls_aes_init(), and either
 *         mbedtls_aes_setkey_enc() or mbedtls_aes_setkey_dec() must be called
 *         before the first call to this API with the same context.
 *
 * \note   This function operates on aligned blocks, that is, the input size
 *         must be a multiple of the AES block size of 16 Bytes.
 *
 * \note   Upon exit, the content of the IV is updated so that you can
 *         call the same function again on the next
 *         block(s) of data and get the same result as if it was
 *         encrypted in one call. This allows a "streaming" usage.
 *         If you need to retain the contents of the IV, you should
 *         either save it manually or use the cipher module instead.
 *
 *
 * \param ctx      The AES context to use for encryption or decryption.
 * \param mode     The AES operation: #MBEDTLS_AES_ENCRYPT or
 *                 #MBEDTLS_AES_DECRYPT.
 * \param length   The length of the input data in Bytes. This must be a
 *                 multiple of the block size (16 Bytes).
 * \param iv       Initialization vector (updated after use).
 * \param input    The buffer holding the input data.
 * \param output   The buffer holding the output data.
 *
 * \return         \c 0 on success.
 * \return         #MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH
 *                 on failure.
 */
int mbedtls_aes_crypt_cbc( mbedtls_aes_context *ctx,
                    int mode,
                    size_t length,
                    unsigned char iv[16],
                    const unsigned char *input,
                    unsigned char *output );
#endif /* MBEDTLS_CIPHER_MODE_CBC */

#if defined(MBEDTLS_CIPHER_MODE_XTS)
/**
 * \brief      This function performs an AES-XTS encryption or decryption
 *             operation for an entire XTS data unit.
 *
 *             AES-XTS encrypts or decrypts blocks based on their location as
 *             defined by a data unit number. The data unit number must be
 *             provided by \p data_unit.
 *
 *             NIST SP 800-38E limits the maximum size of a data unit to 2^20
 *             AES blocks. If the data unit is larger than this, this function
```

```
 *             returns #MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH.
 *
 * \param ctx          The AES XTS context to use for AES XTS operations.
 * \param mode         The AES operation: #MBEDTLS_AES_ENCRYPT or
 *                     #MBEDTLS_AES_DECRYPT.
 * \param length       The length of a data unit in bytes. This can be any
 *                     length between 16 bytes and 2^24 bytes inclusive
 *                     (between 1 and 2^20 block cipher blocks).
 * \param data_unit    The address of the data unit encoded as an array of 16
 *                     bytes in little-endian format. For disk encryption, this
 *                     is typically the index of the block device sector that
 *                     contains the data.
 * \param input        The buffer holding the input data (which is an entire
 *                     data unit). This function reads \p length bytes from \p
 *                     input.
 * \param output       The buffer holding the output data (which is an entire
 *                     data unit). This function writes \p length bytes to \p
 *                     output.
 *
 * \return             \c 0 on success.
 * \return             #MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH if \p length is
 *                     smaller than an AES block in size (16 bytes) or if \p
 *                     length is larger than 2^20 blocks (16 MiB).
 */
int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
                           int mode,
                           size_t length,
                           const unsigned char data_unit[16],
                           const unsigned char *input,
                           unsigned char *output );
#endif /* MBEDTLS_CIPHER_MODE_XTS */

#if defined(MBEDTLS_CIPHER_MODE_CFB)
/**
 * \brief This function performs an AES-CFB128 encryption or decryption
 *        operation.
 *
 *        It performs the operation defined in the \p mode
 *        parameter (encrypt or decrypt), on the input data buffer
 *        defined in the \p input parameter.
 *
 *        For CFB, you must set up the context with mbedtls_aes_setkey_enc(),
 *        regardless of whether you are performing an encryption or decryption
 *        operation, that is, regardless of the \p mode parameter. This is
 *        because CFB mode uses the same key schedule for encryption and
 *        decryption.
 *
 * \note  Upon exit, the content of the IV is updated so that you can
 *        call the same function again on the next
 *        block(s) of data and get the same result as if it was
 *        encrypted in one call. This allows a "streaming" usage.
 *        If you need to retain the contents of the
 *        IV, you must either save it manually or use the cipher
 *        module instead.
 *
 *
 * \param ctx      The AES context to use for encryption or decryption.
 * \param mode     The AES operation: #MBEDTLS_AES_ENCRYPT or
 *                 #MBEDTLS_AES_DECRYPT.
 * \param length   The length of the input data.
 * \param iv_off   The offset in IV (updated after use).
 * \param iv       The initialization vector (updated after use).
 * \param input    The buffer holding the input data.
 * \param output   The buffer holding the output data.
 *
 * \return         \c 0 on success.
 */
int mbedtls_aes_crypt_cfb128( mbedtls_aes_context *ctx,
                      int mode,
```

```
                    size_t length,
                    size_t *iv_off,
                    unsigned char iv[16],
                    const unsigned char *input,
                    unsigned char *output );

/**
 * \brief This function performs an AES-CFB8 encryption or decryption
 *        operation.
 *
 *        It performs the operation defined in the \p mode
 *        parameter (encrypt/decrypt), on the input data buffer defined
 *        in the \p input parameter.
 *
 *        Due to the nature of CFB, you must use the same key schedule for
 *        both encryption and decryption operations. Therefore, you must
 *        use the context initialized with mbedtls_aes_setkey_enc() for
 *        both #MBEDTLS_AES_ENCRYPT and #MBEDTLS_AES_DECRYPT.
 *
 * \note  Upon exit, the content of the IV is updated so that you can
 *        call the same function again on the next
 *        block(s) of data and get the same result as if it was
 *        encrypted in one call. This allows a "streaming" usage.
 *        If you need to retain the contents of the
 *        IV, you should either save it manually or use the cipher
 *        module instead.
 *
 *
 * \param ctx      The AES context to use for encryption or decryption.
 * \param mode     The AES operation: #MBEDTLS_AES_ENCRYPT or
 *                 #MBEDTLS_AES_DECRYPT
 * \param length   The length of the input data.
 * \param iv       The initialization vector (updated after use).
 * \param input    The buffer holding the input data.
 * \param output   The buffer holding the output data.
 *
 * \return         \c 0 on success.
 */
int mbedtls_aes_crypt_cfb8( mbedtls_aes_context *ctx,
                    int mode,
                    size_t length,
                    unsigned char iv[16],
                    const unsigned char *input,
                    unsigned char *output );
#endif /*MBEDTLS_CIPHER_MODE_CFB */

#if defined(MBEDTLS_CIPHER_MODE_OFB)
/**
 * \brief       This function performs an AES-OFB (Output Feedback Mode)
 *              encryption or decryption operation.
 *
 *              For OFB, you must set up the context with
 *              mbedtls_aes_setkey_enc(), regardless of whether you are
 *              performing an encryption or decryption operation. This is
 *              because OFB mode uses the same key schedule for encryption and
 *              decryption.
 *
 *              The OFB operation is identical for encryption or decryption,
 *              therefore no operation mode needs to be specified.
 *
 * \note        Upon exit, the content of iv, the Initialisation Vector, is
 *              updated so that you can call the same function again on the next
 *              block(s) of data and get the same result as if it was encrypted
 *              in one call. This allows a "streaming" usage, by initialising
 *              iv_off to 0 before the first call, and preserving its value
 *              between calls.
 *
 *              For non-streaming use, the iv should be initialised on each call
 *              to a unique value, and iv_off set to 0 on each call.
```

```
 *
 *              If you need to retain the contents of the initialisation vector,
 *              you must either save it manually or use the cipher module
 *              instead.
 *
 * \warning     For the OFB mode, the initialisation vector must be unique
 *              every encryption operation. Reuse of an initialisation vector
 *              will compromise security.
 *
 * \param ctx      The AES context to use for encryption or decryption.
 * \param length   The length of the input data.
 * \param iv_off   The offset in IV (updated after use).
 * \param iv       The initialization vector (updated after use).
 * \param input    The buffer holding the input data.
 * \param output   The buffer holding the output data.
 *
 * \return         \c 0 on success.
 */
int mbedtls_aes_crypt_ofb( mbedtls_aes_context *ctx,
                       size_t length,
                       size_t *iv_off,
                       unsigned char iv[16],
                       const unsigned char *input,
                       unsigned char *output );

#endif /* MBEDTLS_CIPHER_MODE_OFB */

#if defined(MBEDTLS_CIPHER_MODE_CTR)
/**
 * \brief      This function performs an AES-CTR encryption or decryption
 *             operation.
 *
 *             This function performs the operation defined in the \p mode
 *             parameter (encrypt/decrypt), on the input data buffer
 *             defined in the \p input parameter.
 *
 *             Due to the nature of CTR, you must use the same key schedule
 *             for both encryption and decryption operations. Therefore, you
 *             must use the context initialized with mbedtls_aes_setkey_enc()
 *             for both #MBEDTLS_AES_ENCRYPT and #MBEDTLS_AES_DECRYPT.
 *
 * \warning    You must never reuse a nonce value with the same key. Doing so
 *             would void the encryption for the two messages encrypted with
 *             the same nonce and key.
 *
 *             There are two common strategies for managing nonces with CTR:
 *
 *             1. You can handle everything as a single message processed over
 *             successive calls to this function. In that case, you want to
 *             set \p nonce_counter and \p nc_off to 0 for the first call, and
 *             then preserve the values of \p nonce_counter, \p nc_off and \p
 *             stream_block across calls to this function as they will be
 *             updated by this function.
 *
 *             With this strategy, you must not encrypt more than 2**128
 *             blocks of data with the same key.
 *
 *             2. You can encrypt separate messages by dividing the \p
 *             nonce_counter buffer in two areas: the first one used for a
 *             per-message nonce, handled by yourself, and the second one
 *             updated by this function internally.
 *
 *             For example, you might reserve the first 12 bytes for the
 *             per-message nonce, and the last 4 bytes for internal use. In that
 *             case, before calling this function on a new message you need to
 *             set the first 12 bytes of \p nonce_counter to your chosen nonce
 *             value, the last 4 to 0, and \p nc_off to 0 (which will cause \p
 *             stream_block to be ignored). That way, you can encrypt at most
 *             2**96 messages of up to 2**32 blocks each with the same key.
```

```
 *
 *             The per-message nonce (or information sufficient to reconstruct
 *             it) needs to be communicated with the ciphertext and must be unique.
 *             The recommended way to ensure uniqueness is to use a message
 *             counter. An alternative is to generate random nonces, but this
 *             limits the number of messages that can be securely encrypted:
 *             for example, with 96-bit random nonces, you should not encrypt
 *             more than 2**32 messages with the same key.
 *
 *             Note that for both stategies, sizes are measured in blocks and
 *             that an AES block is 16 bytes.
 *
 * \warning    Upon return, \p stream_block contains sensitive data. Its
 *             content must not be written to insecure storage and should be
 *             securely discarded as soon as it's no longer needed.
 *
 * \param ctx              The AES context to use for encryption or decryption.
 * \param length           The length of the input data.
 * \param nc_off           The offset in the current \p stream_block, for
 *                         resuming within the current cipher stream. The
 *                         offset pointer should be 0 at the start of a stream.
 * \param nonce_counter    The 128-bit nonce and counter.
 * \param stream_block     The saved stream block for resuming. This is
 *                         overwritten by the function.
 * \param input            The buffer holding the input data.
 * \param output           The buffer holding the output data.
 *
 * \return                 \c 0 on success.
 */
int mbedtls_aes_crypt_ctr( mbedtls_aes_context *ctx,
                       size_t length,
                       size_t *nc_off,
                       unsigned char nonce_counter[16],
                       unsigned char stream_block[16],
                       const unsigned char *input,
                       unsigned char *output );
#endif /* MBEDTLS_CIPHER_MODE_CTR */

/**
 * \brief          Internal AES block encryption function. This is only
 *                 exposed to allow overriding it using
 *                 \c MBEDTLS_AES_ENCRYPT_ALT.
 *
 * \param ctx      The AES context to use for encryption.
 * \param input    The plaintext block.
 * \param output   The output (ciphertext) block.
 *
 * \return         \c 0 on success.
 */
int mbedtls_internal_aes_encrypt( mbedtls_aes_context *ctx,
                                  const unsigned char input[16],
                                  unsigned char output[16] );

/**
 * \brief          Internal AES block decryption function. This is only
 *                 exposed to allow overriding it using see
 *                 \c MBEDTLS_AES_DECRYPT_ALT.
 *
 * \param ctx      The AES context to use for decryption.
 * \param input    The ciphertext block.
 * \param output   The output (plaintext) block.
 *
 * \return         \c 0 on success.
 */
int mbedtls_internal_aes_decrypt( mbedtls_aes_context *ctx,
                                  const unsigned char input[16],
                                  unsigned char output[16] );

#if !defined(MBEDTLS_DEPRECATED_REMOVED)
```

```
#if defined(MBEDTLS_DEPRECATED_WARNING)
#define MBEDTLS_DEPRECATED      __attribute__((deprecated))
#else
#define MBEDTLS_DEPRECATED
#endif
/**
 * \brief          Deprecated internal AES block encryption function
 *                 without return value.
 *
 * \deprecated     Superseded by mbedtls_aes_encrypt_ext() in 2.5.0.
 *
 * \param ctx      The AES context to use for encryption.
 * \param input    Plaintext block.
 * \param output   Output (ciphertext) block.
 */
MBEDTLS_DEPRECATED void mbedtls_aes_encrypt( mbedtls_aes_context *ctx,
                                             const unsigned char input[16],
                                             unsigned char output[16] );

/**
 * \brief          Deprecated internal AES block decryption function
 *                 without return value.
 *
 * \deprecated     Superseded by mbedtls_aes_decrypt_ext() in 2.5.0.
 *
 * \param ctx      The AES context to use for decryption.
 * \param input    Ciphertext block.
 * \param output   Output (plaintext) block.
 */
MBEDTLS_DEPRECATED void mbedtls_aes_decrypt( mbedtls_aes_context *ctx,
                                             const unsigned char input[16],
                                             unsigned char output[16] );

#undef MBEDTLS_DEPRECATED
#endif /* !MBEDTLS_DEPRECATED_REMOVED */

/**
 * \brief          Checkup routine.
 *
 * \return         \c 0 on success.
 * \return         \c 1 on failure.
 */
int mbedtls_aes_self_test( int verbose );

#ifdef __cplusplus
}
#endif

#endif /* aes.h */
```

Source - aes.c

The aes.c source code can also be found in the trunk on: aes.c.

```
/*
 *  FIPS-197 compliant AES implementation
 *
 *  Copyright (C) 2006-2015, ARM Limited, All Rights Reserved
 *  SPDX-License-Identifier: Apache-2.0
 *
 *  Licensed under the Apache License, Version 2.0 (the "License"); you may
 *  not use this file except in compliance with the License.
 *  You may obtain a copy of the License at
 *
 *  http://www.apache.org/licenses/LICENSE-2.0
 *
 *  Unless required by applicable law or agreed to in writing, software
 *  distributed under the License is distributed on an "AS IS" BASIS, WITHOUT
```

```
 *  WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *  See the License for the specific language governing permissions and
 *  limitations under the License.
 *
 *  This file is part of mbed TLS (https://tls.mbed.org)
 */
/*
 *  The AES block cipher was designed by Vincent Rijmen and Joan Daemen.
 *
 *  http://csrc.nist.gov/encryption/aes/rijndael/Rijndael.pdf
 *  http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
 */

#if !defined(MBEDTLS_CONFIG_FILE)
#include "mbedtls/config.h"
#else
#include MBEDTLS_CONFIG_FILE
#endif

#if defined(MBEDTLS_AES_C)

#include <string.h>

#include "mbedtls/aes.h"
#include "mbedtls/platform.h"
#include "mbedtls/platform_util.h"
#if defined(MBEDTLS_PADLOCK_C)
#include "mbedtls/padlock.h"
#endif
#if defined(MBEDTLS_AESNI_C)
#include "mbedtls/aesni.h"
#endif

#if defined(MBEDTLS_SELF_TEST)
#if defined(MBEDTLS_PLATFORM_C)
#include "mbedtls/platform.h"
#else
#include <stdio.h>
#define mbedtls_printf printf
#endif /* MBEDTLS_PLATFORM_C */
#endif /* MBEDTLS_SELF_TEST */

#if !defined(MBEDTLS_AES_ALT)

/*
 * 32-bit integer manipulation macros (little endian)
 */
#ifndef GET_UINT32_LE
#define GET_UINT32_LE(n,b,i)                            \
{                                                       \
    (n) = ( (uint32_t) (b)[(i)    ]       )             \
        | ( (uint32_t) (b)[(i) + 1] <<  8 )             \
        | ( (uint32_t) (b)[(i) + 2] << 16 )             \
        | ( (uint32_t) (b)[(i) + 3] << 24 );            \
}
#endif

#ifndef PUT_UINT32_LE
#define PUT_UINT32_LE(n,b,i)                                    \
{                                                               \
    (b)[(i)    ] = (unsigned char) ( ( (n)       ) & 0xFF );    \
    (b)[(i) + 1] = (unsigned char) ( ( (n) >>  8 ) & 0xFF );    \
    (b)[(i) + 2] = (unsigned char) ( ( (n) >> 16 ) & 0xFF );    \
    (b)[(i) + 3] = (unsigned char) ( ( (n) >> 24 ) & 0xFF );    \
}
#endif

#if defined(MBEDTLS_PADLOCK_C) &&                       \
```

```
    ( defined(MBEDTLS_HAVE_X86) || defined(MBEDTLS_PADLOCK_ALIGN16) )
static int aes_padlock_ace = -1;
#endif

#if defined(MBEDTLS_AES_ROM_TABLES)
/*
 * Forward S-box
 */
static const unsigned char FSb[256] =
{
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5,
    0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0,
    0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC,
    0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A,
    0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0,
    0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B,
    0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
    0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5,
    0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17,
    0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88,
    0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C,
    0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9,
    0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6,
    0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E,
    0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94,
    0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68,
    0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};

/*
 * Forward tables
 */
#define FT \
\
    V(A5,63,63,C6), V(84,7C,7C,F8), V(99,77,77,EE), V(8D,7B,7B,F6), \
    V(0D,F2,F2,FF), V(BD,6B,6B,D6), V(B1,6F,6F,DE), V(54,C5,C5,91), \
    V(50,30,30,60), V(03,01,01,02), V(A9,67,67,CE), V(7D,2B,2B,56), \
    V(19,FE,FE,E7), V(62,D7,D7,B5), V(E6,AB,AB,4D), V(9A,76,76,EC), \
    V(45,CA,CA,8F), V(9D,82,82,1F), V(40,C9,C9,89), V(87,7D,7D,FA), \
    V(15,FA,FA,EF), V(EB,59,59,B2), V(C9,47,47,8E), V(0B,F0,F0,FB), \
    V(EC,AD,AD,41), V(67,D4,D4,B3), V(FD,A2,A2,5F), V(EA,AF,AF,45), \
    V(BF,9C,9C,23), V(F7,A4,A4,53), V(96,72,72,E4), V(5B,C0,C0,9B), \
    V(C2,B7,B7,75), V(1C,FD,FD,E1), V(AE,93,93,3D), V(6A,26,26,4C), \
    V(5A,36,36,6C), V(41,3F,3F,7E), V(02,F7,F7,F5), V(4F,CC,CC,83), \
    V(5C,34,34,68), V(F4,A5,A5,51), V(34,E5,E5,D1), V(08,F1,F1,F9), \
    V(93,71,71,E2), V(73,D8,D8,AB), V(53,31,31,62), V(3F,15,15,2A), \
    V(0C,04,04,08), V(52,C7,C7,95), V(65,23,23,46), V(5E,C3,C3,9D), \
    V(28,18,18,30), V(A1,96,96,37), V(0F,05,05,0A), V(B5,9A,9A,2F), \
    V(09,07,07,0E), V(36,12,12,24), V(9B,80,80,1B), V(3D,E2,E2,DF), \
    V(26,EB,EB,CD), V(69,27,27,4E), V(CD,B2,B2,7F), V(9F,75,75,EA), \
    V(1B,09,09,12), V(9E,83,83,1D), V(74,2C,2C,58), V(2E,1A,1A,34), \
    V(2D,1B,1B,36), V(B2,6E,6E,DC), V(EE,5A,5A,B4), V(FB,A0,A0,5B), \
    V(F6,52,52,A4), V(4D,3B,3B,76), V(61,D6,D6,B7), V(CE,B3,B3,7D), \
    V(7B,29,29,52), V(3E,E3,E3,DD), V(71,2F,2F,5E), V(97,84,84,13), \
```

```
    V(F5,53,53,A6), V(68,D1,D1,B9), V(00,00,00,00), V(2C,ED,ED,C1), \
    V(60,20,20,40), V(1F,FC,FC,E3), V(C8,B1,B1,79), V(ED,5B,5B,B6), \
    V(BE,6A,6A,D4), V(46,CB,CB,8D), V(D9,BE,BE,67), V(4B,39,39,72), \
    V(DE,4A,4A,94), V(D4,4C,4C,98), V(E8,58,58,B0), V(4A,CF,CF,85), \
    V(6B,D0,D0,BB), V(2A,EF,EF,C5), V(E5,AA,AA,4F), V(16,FB,FB,ED), \
    V(C5,43,43,86), V(D7,4D,4D,9A), V(55,33,33,66), V(94,85,85,11), \
    V(CF,45,45,8A), V(10,F9,F9,E9), V(06,02,02,04), V(81,7F,7F,FE), \
    V(F0,50,50,A0), V(44,3C,3C,78), V(BA,9F,9F,25), V(E3,A8,A8,4B), \
    V(F3,51,51,A2), V(FE,A3,A3,5D), V(C0,40,40,80), V(8A,8F,8F,05), \
    V(AD,92,92,3F), V(BC,9D,9D,21), V(48,38,38,70), V(04,F5,F5,F1), \
    V(DF,BC,BC,63), V(C1,B6,B6,77), V(75,DA,DA,AF), V(63,21,21,42), \
    V(30,10,10,20), V(1A,FF,FF,E5), V(0E,F3,F3,FD), V(6D,D2,D2,BF), \
    V(4C,CD,CD,81), V(14,0C,0C,18), V(35,13,13,26), V(2F,EC,EC,C3), \
    V(E1,5F,5F,BE), V(A2,97,97,35), V(CC,44,44,88), V(39,17,17,2E), \
    V(57,C4,C4,93), V(F2,A7,A7,55), V(82,7E,7E,FC), V(47,3D,3D,7A), \
    V(AC,64,64,C8), V(E7,5D,5D,BA), V(2B,19,19,32), V(95,73,73,E6), \
    V(A0,60,60,C0), V(98,81,81,19), V(D1,4F,4F,9E), V(7F,DC,DC,A3), \
    V(66,22,22,44), V(7E,2A,2A,54), V(AB,90,90,3B), V(83,88,88,0B), \
    V(CA,46,46,8C), V(29,EE,EE,C7), V(D3,B8,B8,6B), V(3C,14,14,28), \
    V(79,DE,DE,A7), V(E2,5E,5E,BC), V(1D,0B,0B,16), V(76,DB,DB,AD), \
    V(3B,E0,E0,DB), V(56,32,32,64), V(4E,3A,3A,74), V(1E,0A,0A,14), \
    V(DB,49,49,92), V(0A,06,06,0C), V(6C,24,24,48), V(E4,5C,5C,B8), \
    V(5D,C2,C2,9F), V(6E,D3,D3,BD), V(EF,AC,AC,43), V(A6,62,62,C4), \
    V(A8,91,91,39), V(A4,95,95,31), V(37,E4,E4,D3), V(8B,79,79,F2), \
    V(32,E7,E7,D5), V(43,C8,C8,8B), V(59,37,37,6E), V(B7,6D,6D,DA), \
    V(8C,8D,8D,01), V(64,D5,D5,B1), V(D2,4E,4E,9C), V(E0,A9,A9,49), \
    V(B4,6C,6C,D8), V(FA,56,56,AC), V(07,F4,F4,F3), V(25,EA,EA,CF), \
    V(AF,65,65,CA), V(8E,7A,7A,F4), V(E9,AE,AE,47), V(18,08,08,10), \
    V(D5,BA,BA,6F), V(88,78,78,F0), V(6F,25,25,4A), V(72,2E,2E,5C), \
    V(24,1C,1C,38), V(F1,A6,A6,57), V(C7,B4,B4,73), V(51,C6,C6,97), \
    V(23,E8,E8,CB), V(7C,DD,DD,A1), V(9C,74,74,E8), V(21,1F,1F,3E), \
    V(DD,4B,4B,96), V(DC,BD,BD,61), V(86,8B,8B,0D), V(85,8A,8A,0F), \
    V(90,70,70,E0), V(42,3E,3E,7C), V(C4,B5,B5,71), V(AA,66,66,CC), \
    V(D8,48,48,90), V(05,03,03,06), V(01,F6,F6,F7), V(12,0E,0E,1C), \
    V(A3,61,61,C2), V(5F,35,35,6A), V(F9,57,57,AE), V(D0,B9,B9,69), \
    V(91,86,86,17), V(58,C1,C1,99), V(27,1D,1D,3A), V(B9,9E,9E,27), \
    V(38,E1,E1,D9), V(13,F8,F8,EB), V(B3,98,98,2B), V(33,11,11,22), \
    V(BB,69,69,D2), V(70,D9,D9,A9), V(89,8E,8E,07), V(A7,94,94,33), \
    V(B6,9B,9B,2D), V(22,1E,1E,3C), V(92,87,87,15), V(20,E9,E9,C9), \
    V(49,CE,CE,87), V(FF,55,55,AA), V(78,28,28,50), V(7A,DF,DF,A5), \
    V(8F,8C,8C,03), V(F8,A1,A1,59), V(80,89,89,09), V(17,0D,0D,1A), \
    V(DA,BF,BF,65), V(31,E6,E6,D7), V(C6,42,42,84), V(B8,68,68,D0), \
    V(C3,41,41,82), V(B0,99,99,29), V(77,2D,2D,5A), V(11,0F,0F,1E), \
    V(CB,B0,B0,7B), V(FC,54,54,A8), V(D6,BB,BB,6D), V(3A,16,16,2C)

#define V(a,b,c,d) 0x##a##b##c##d
static const uint32_t FT0[256] = { FT };
#undef V

#if !defined(MBEDTLS_AES_FEWER_TABLES)

#define V(a,b,c,d) 0x##b##c##d##a
static const uint32_t FT1[256] = { FT };
#undef V

#define V(a,b,c,d) 0x##c##d##a##b
static const uint32_t FT2[256] = { FT };
#undef V

#define V(a,b,c,d) 0x##d##a##b##c
static const uint32_t FT3[256] = { FT };
#undef V

#endif /* !MBEDTLS_AES_FEWER_TABLES */

#undef FT

/*
 * Reverse S-box
```

```
 */
static const unsigned char RSb[256] =
{
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38,
    0xBF, 0x40, 0xA3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87,
    0x34, 0x8E, 0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D,
    0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2,
    0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16,
    0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA,
    0x5E, 0x15, 0x46, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A,
    0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02,
    0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA,
    0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85,
    0xE2, 0xF9, 0x37, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89,
    0x6F, 0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20,
    0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31,
    0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D,
    0x2D, 0xE5, 0x7A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0,
    0xC8, 0xEB, 0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26,
    0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
};

/*
 * Reverse tables
 */
#define RT \
\
    V(50,A7,F4,51), V(53,65,41,7E), V(C3,A4,17,1A), V(96,5E,27,3A), \
    V(CB,6B,AB,3B), V(F1,45,9D,1F), V(AB,58,FA,AC), V(93,03,E3,4B), \
    V(55,FA,30,20), V(F6,6D,76,AD), V(91,76,CC,88), V(25,4C,02,F5), \
    V(FC,D7,E5,4F), V(D7,CB,2A,C5), V(80,44,35,26), V(8F,A3,62,B5), \
    V(49,5A,B1,DE), V(67,1B,BA,25), V(98,0E,EA,45), V(E1,C0,FE,5D), \
    V(02,75,2F,C3), V(12,F0,4C,81), V(A3,97,46,8D), V(C6,F9,D3,6B), \
    V(E7,5F,8F,03), V(95,9C,92,15), V(EB,7A,6D,BF), V(DA,59,52,95), \
    V(2D,83,BE,D4), V(D3,21,74,58), V(29,69,E0,49), V(44,C8,C9,8E), \
    V(6A,89,C2,75), V(78,79,8E,F4), V(6B,3E,58,99), V(DD,71,B9,27), \
    V(B6,4F,E1,BE), V(17,AD,88,F0), V(66,AC,20,C9), V(B4,3A,CE,7D), \
    V(18,4A,DF,63), V(82,31,1A,E5), V(60,33,51,97), V(45,7F,53,62), \
    V(E0,77,64,B1), V(84,AE,6B,BB), V(1C,A0,81,FE), V(94,2B,08,F9), \
    V(58,68,48,70), V(19,FD,45,8F), V(87,6C,DE,94), V(B7,F8,7B,52), \
    V(23,D3,73,AB), V(E2,02,4B,72), V(57,8F,1F,E3), V(2A,AB,55,66), \
    V(07,28,EB,B2), V(03,C2,B5,2F), V(9A,7B,C5,86), V(A5,08,37,D3), \
    V(F2,87,28,30), V(B2,A5,BF,23), V(BA,6A,03,02), V(5C,82,16,ED), \
    V(2B,1C,CF,8A), V(92,B4,79,A7), V(F0,F2,07,F3), V(A1,E2,69,4E), \
    V(CD,F4,DA,65), V(D5,BE,05,06), V(1F,62,34,D1), V(8A,FE,A6,C4), \
    V(9D,53,2E,34), V(A0,55,F3,A2), V(32,E1,8A,05), V(75,EB,F6,A4), \
    V(39,EC,83,0B), V(AA,EF,60,40), V(06,9F,71,5E), V(51,10,6E,BD), \
    V(F9,8A,21,3E), V(3D,06,DD,96), V(AE,05,3E,DD), V(46,BD,E6,4D), \
    V(B5,8D,54,91), V(05,5D,C4,71), V(6F,D4,06,04), V(FF,15,50,60), \
    V(24,FB,98,19), V(97,E9,BD,D6), V(CC,43,40,89), V(77,9E,D9,67), \
    V(BD,42,E8,B0), V(88,8B,89,07), V(38,5B,19,E7), V(DB,EE,C8,79), \
    V(47,0A,7C,A1), V(E9,0F,42,7C), V(C9,1E,84,F8), V(00,00,00,00), \
    V(83,86,80,09), V(48,ED,2B,32), V(AC,70,11,1E), V(4E,72,5A,6C), \
    V(FB,FF,0E,FD), V(56,38,85,0F), V(1E,D5,AE,3D), V(27,39,2D,36), \
```

```
    V(64,D9,0F,0A), V(21,A6,5C,68), V(D1,54,5B,9B), V(3A,2E,36,24), \
    V(B1,67,0A,0C), V(0F,E7,57,93), V(D2,96,EE,B4), V(9E,91,9B,1B), \
    V(4F,C5,C0,80), V(A2,20,DC,61), V(69,4B,77,5A), V(16,1A,12,1C), \
    V(0A,BA,93,E2), V(E5,2A,A0,C0), V(43,E0,22,3C), V(1D,17,1B,12), \
    V(0B,0D,09,0E), V(AD,C7,8B,F2), V(B9,A8,B6,2D), V(C8,A9,1E,14), \
    V(85,19,F1,57), V(4C,07,75,AF), V(BB,DD,99,EE), V(FD,60,7F,A3), \
    V(9F,26,01,F7), V(BC,F5,72,5C), V(C5,3B,66,44), V(34,7E,FB,5B), \
    V(76,29,43,8B), V(DC,C6,23,CB), V(68,FC,ED,B6), V(63,F1,E4,B8), \
    V(CA,DC,31,D7), V(10,85,63,42), V(40,22,97,13), V(20,11,C6,84), \
    V(7D,24,4A,85), V(F8,3D,BB,D2), V(11,32,F9,AE), V(6D,A1,29,C7), \
    V(4B,2F,9E,1D), V(F3,30,B2,DC), V(EC,52,86,0D), V(D0,E3,C1,77), \
    V(6C,16,B3,2B), V(99,B9,70,A9), V(FA,48,94,11), V(22,64,E9,47), \
    V(C4,8C,FC,A8), V(1A,3F,F0,A0), V(D8,2C,7D,56), V(EF,90,33,22), \
    V(C7,4E,49,87), V(C1,D1,38,D9), V(FE,A2,CA,8C), V(36,0B,D4,98), \
    V(CF,81,F5,A6), V(28,DE,7A,A5), V(26,8E,B7,DA), V(A4,BF,AD,3F), \
    V(E4,9D,3A,2C), V(0D,92,78,50), V(9B,CC,5F,6A), V(62,46,7E,54), \
    V(C2,13,8D,F6), V(E8,B8,D8,90), V(5E,F7,39,2E), V(F5,AF,C3,82), \
    V(BE,80,5D,9F), V(7C,93,D0,69), V(A9,2D,D5,6F), V(B3,12,25,CF), \
    V(3B,99,AC,C8), V(A7,7D,18,10), V(6E,63,9C,E8), V(7B,BB,3B,DB), \
    V(09,78,26,CD), V(F4,18,59,6E), V(01,B7,9A,EC), V(A8,9A,4F,83), \
    V(65,6E,95,E6), V(7E,E6,FF,AA), V(08,CF,BC,21), V(E6,E8,15,EF), \
    V(D9,9B,E7,BA), V(CE,36,6F,4A), V(D4,09,9F,EA), V(D6,7C,B0,29), \
    V(AF,B2,A4,31), V(31,23,3F,2A), V(30,94,A5,C6), V(C0,66,A2,35), \
    V(37,BC,4E,74), V(A6,CA,82,FC), V(B0,D0,90,E0), V(15,D8,A7,33), \
    V(4A,98,04,F1), V(F7,DA,EC,41), V(0E,50,CD,7F), V(2F,F6,91,17), \
    V(8D,D6,4D,76), V(4D,B0,EF,43), V(54,4D,AA,CC), V(DF,04,96,E4), \
    V(E3,B5,D1,9E), V(1B,88,6A,4C), V(B8,1F,2C,C1), V(7F,51,65,46), \
    V(04,EA,5E,9D), V(5D,35,8C,01), V(73,74,87,FA), V(2E,41,0B,FB), \
    V(5A,1D,67,B3), V(52,D2,DB,92), V(33,56,10,E9), V(13,47,D6,6D), \
    V(8C,61,D7,9A), V(7A,0C,A1,37), V(8E,14,F8,59), V(89,3C,13,EB), \
    V(EE,27,A9,CE), V(35,C9,61,B7), V(ED,E5,1C,E1), V(3C,B1,47,7A), \
    V(59,DF,D2,9C), V(3F,73,F2,55), V(79,CE,14,18), V(BF,37,C7,73), \
    V(EA,CD,F7,53), V(5B,AA,FD,5F), V(14,6F,3D,DF), V(86,DB,44,78), \
    V(81,F3,AF,CA), V(3E,C4,68,B9), V(2C,34,24,38), V(5F,40,A3,C2), \
    V(72,C3,1D,16), V(0C,25,E2,BC), V(8B,49,3C,28), V(41,95,0D,FF), \
    V(71,01,A8,39), V(DE,B3,0C,08), V(9C,E4,B4,D8), V(90,C1,56,64), \
    V(61,84,CB,7B), V(70,B6,32,D5), V(74,5C,6C,48), V(42,57,B8,D0)

#define V(a,b,c,d) 0x##a##b##c##d
static const uint32_t RT0[256] = { RT };
#undef V

#if !defined(MBEDTLS_AES_FEWER_TABLES)

#define V(a,b,c,d) 0x##b##c##d##a
static const uint32_t RT1[256] = { RT };
#undef V

#define V(a,b,c,d) 0x##c##d##a##b
static const uint32_t RT2[256] = { RT };
#undef V

#define V(a,b,c,d) 0x##d##a##b##c
static const uint32_t RT3[256] = { RT };
#undef V

#endif /* !MBEDTLS_AES_FEWER_TABLES */

#undef RT

/*
 * Round constants
 */
static const uint32_t RCON[10] =
{
    0x00000001, 0x00000002, 0x00000004, 0x00000008,
    0x00000010, 0x00000020, 0x00000040, 0x00000080,
    0x0000001B, 0x00000036
};
```

```
#else /* MBEDTLS_AES_ROM_TABLES */

/*
 * Forward S-box & tables
 */
static unsigned char FSb[256];
static uint32_t FT0[256];
#if !defined(MBEDTLS_AES_FEWER_TABLES)
static uint32_t FT1[256];
static uint32_t FT2[256];
static uint32_t FT3[256];
#endif /* !MBEDTLS_AES_FEWER_TABLES */

/*
 * Reverse S-box & tables
 */
static unsigned char RSb[256];
static uint32_t RT0[256];
#if !defined(MBEDTLS_AES_FEWER_TABLES)
static uint32_t RT1[256];
static uint32_t RT2[256];
static uint32_t RT3[256];
#endif /* !MBEDTLS_AES_FEWER_TABLES */

/*
 * Round constants
 */
static uint32_t RCON[10];

/*
 * Tables generation code
 */
#define ROTL8(x) ( ( x << 8 ) & 0xFFFFFFFF ) | ( x >> 24 )
#define XTIME(x) ( ( x << 1 ) ^ ( ( x & 0x80 ) ? 0x1B : 0x00 ) )
#define MUL(x,y) ( ( x && y ) ? pow[(log[x]+log[y]) % 255] : 0 )

static int aes_init_done = 0;

static void aes_gen_tables( void )
{
    int i, x, y, z;
    int pow[256];
    int log[256];

    /*
     * compute pow and log tables over GF(2^8)
     */
    for( i = 0, x = 1; i < 256; i++ )
    {
        pow[i] = x;
        log[x] = i;
        x = ( x ^ XTIME( x ) ) & 0xFF;
    }

    /*
     * calculate the round constants
     */
    for( i = 0, x = 1; i < 10; i++ )
    {
        RCON[i] = (uint32_t) x;
        x = XTIME( x ) & 0xFF;
    }

    /*
     * generate the forward and reverse S-boxes
     */
    FSb[0x00] = 0x63;
    RSb[0x63] = 0x00;
```

```
    for( i = 1; i < 256; i++ )
    {
        x = pow[255 - log[i]];

        y  = x; y = ( ( y << 1 ) | ( y >> 7 ) ) & 0xFF;
        x ^= y; y = ( ( y << 1 ) | ( y >> 7 ) ) & 0xFF;
        x ^= y; y = ( ( y << 1 ) | ( y >> 7 ) ) & 0xFF;
        x ^= y; y = ( ( y << 1 ) | ( y >> 7 ) ) & 0xFF;
        x ^= y ^ 0x63;

        FSb[i] = (unsigned char) x;
        RSb[x] = (unsigned char) i;
    }

    /*
     * generate the forward and reverse tables
     */
    for( i = 0; i < 256; i++ )
    {
        x = FSb[i];
        y = XTIME( x ) & 0xFF;
        z =  ( y ^ x ) & 0xFF;

        FT0[i] = ( (uint32_t) y       ) ^
                 ( (uint32_t) x <<  8 ) ^
                 ( (uint32_t) x << 16 ) ^
                 ( (uint32_t) z << 24 );

#if !defined(MBEDTLS_AES_FEWER_TABLES)
        FT1[i] = ROTL8( FT0[i] );
        FT2[i] = ROTL8( FT1[i] );
        FT3[i] = ROTL8( FT2[i] );
#endif /* !MBEDTLS_AES_FEWER_TABLES */

        x = RSb[i];

        RT0[i] = ( (uint32_t) MUL( 0x0E, x )       ) ^
                 ( (uint32_t) MUL( 0x09, x ) <<  8 ) ^
                 ( (uint32_t) MUL( 0x0D, x ) << 16 ) ^
                 ( (uint32_t) MUL( 0x0B, x ) << 24 );

#if !defined(MBEDTLS_AES_FEWER_TABLES)
        RT1[i] = ROTL8( RT0[i] );
        RT2[i] = ROTL8( RT1[i] );
        RT3[i] = ROTL8( RT2[i] );
#endif /* !MBEDTLS_AES_FEWER_TABLES */
    }
}

#undef ROTL8

#endif /* MBEDTLS_AES_ROM_TABLES */

#if defined(MBEDTLS_AES_FEWER_TABLES)

#define ROTL8(x)  ( (uint32_t)( ( x ) <<  8 ) + (uint32_t)( ( x ) >> 24 ) )
#define ROTL16(x) ( (uint32_t)( ( x ) << 16 ) + (uint32_t)( ( x ) >> 16 ) )
#define ROTL24(x) ( (uint32_t)( ( x ) << 24 ) + (uint32_t)( ( x ) >>  8 ) )

#define AES_RT0(idx) RT0[idx]
#define AES_RT1(idx) ROTL8(  RT0[idx] )
#define AES_RT2(idx) ROTL16( RT0[idx] )
#define AES_RT3(idx) ROTL24( RT0[idx] )

#define AES_FT0(idx) FT0[idx]
#define AES_FT1(idx) ROTL8(  FT0[idx] )
#define AES_FT2(idx) ROTL16( FT0[idx] )
#define AES_FT3(idx) ROTL24( FT0[idx] )
```

```
#else /* MBEDTLS_AES_FEWER_TABLES */

#define AES_RT0(idx) RT0[idx]
#define AES_RT1(idx) RT1[idx]
#define AES_RT2(idx) RT2[idx]
#define AES_RT3(idx) RT3[idx]

#define AES_FT0(idx) FT0[idx]
#define AES_FT1(idx) FT1[idx]
#define AES_FT2(idx) FT2[idx]
#define AES_FT3(idx) FT3[idx]

#endif /* MBEDTLS_AES_FEWER_TABLES */

void mbedtls_aes_init( mbedtls_aes_context *ctx )
{
    memset( ctx, 0, sizeof( mbedtls_aes_context ) );
}

void mbedtls_aes_free( mbedtls_aes_context *ctx )
{
    if( ctx == NULL )
        return;

    mbedtls_platform_zeroize( ctx, sizeof( mbedtls_aes_context ) );
}

#if defined(MBEDTLS_CIPHER_MODE_XTS)
void mbedtls_aes_xts_init( mbedtls_aes_xts_context *ctx )
{
    mbedtls_aes_init( &ctx->crypt );
    mbedtls_aes_init( &ctx->tweak );
}

void mbedtls_aes_xts_free( mbedtls_aes_xts_context *ctx )
{
    mbedtls_aes_free( &ctx->crypt );
    mbedtls_aes_free( &ctx->tweak );
}
#endif /* MBEDTLS_CIPHER_MODE_XTS */

/*
 * AES key schedule (encryption)
 */
#if !defined(MBEDTLS_AES_SETKEY_ENC_ALT)
int mbedtls_aes_setkey_enc( mbedtls_aes_context *ctx, const unsigned char *key,
                    unsigned int keybits )
{
    unsigned int i;
    uint32_t *RK;

#if !defined(MBEDTLS_AES_ROM_TABLES)
    if( aes_init_done == 0 )
    {
        aes_gen_tables();
        aes_init_done = 1;

    }
#endif

    switch( keybits )
    {
        case 128: ctx->nr = 10; break;
        case 192: ctx->nr = 12; break;
        case 256: ctx->nr = 14; break;
        default : return( MBEDTLS_ERR_AES_INVALID_KEY_LENGTH );
    }
```

```
#if defined(MBEDTLS_PADLOCK_C) && defined(MBEDTLS_PADLOCK_ALIGN16)
    if( aes_padlock_ace == -1 )
        aes_padlock_ace = mbedtls_padlock_has_support( MBEDTLS_PADLOCK_ACE );

    if( aes_padlock_ace )
        ctx->rk = RK = MBEDTLS_PADLOCK_ALIGN16( ctx->buf );
    else
#endif
    ctx->rk = RK = ctx->buf;

#if defined(MBEDTLS_AESNI_C) && defined(MBEDTLS_HAVE_X86_64)
    if( mbedtls_aesni_has_support( MBEDTLS_AESNI_AES ) )
        return( mbedtls_aesni_setkey_enc( (unsigned char *) ctx->rk, key, keybits ) );
#endif

    for( i = 0; i < ( keybits >> 5 ); i++ )
    {
        GET_UINT32_LE( RK[i], key, i << 2 );
    }

    switch( ctx->nr )
    {
        case 10:

            for( i = 0; i < 10; i++, RK += 4 )
            {
                RK[4]  = RK[0] ^ RCON[i] ^
                ( (uint32_t) FSb[ ( RK[3] >>  8 ) & 0xFF ]       ) ^
                ( (uint32_t) FSb[ ( RK[3] >> 16 ) & 0xFF ] <<  8 ) ^
                ( (uint32_t) FSb[ ( RK[3] >> 24 ) & 0xFF ] << 16 ) ^
                ( (uint32_t) FSb[ ( RK[3]       ) & 0xFF ] << 24 );

                RK[5]  = RK[1] ^ RK[4];
                RK[6]  = RK[2] ^ RK[5];
                RK[7]  = RK[3] ^ RK[6];
            }
            break;

        case 12:

            for( i = 0; i < 8; i++, RK += 6 )
            {
                RK[6]  = RK[0] ^ RCON[i] ^
                ( (uint32_t) FSb[ ( RK[5] >>  8 ) & 0xFF ]       ) ^
                ( (uint32_t) FSb[ ( RK[5] >> 16 ) & 0xFF ] <<  8 ) ^
                ( (uint32_t) FSb[ ( RK[5] >> 24 ) & 0xFF ] << 16 ) ^
                ( (uint32_t) FSb[ ( RK[5]       ) & 0xFF ] << 24 );

                RK[7]  = RK[1] ^ RK[6];
                RK[8]  = RK[2] ^ RK[7];
                RK[9]  = RK[3] ^ RK[8];
                RK[10] = RK[4] ^ RK[9];
                RK[11] = RK[5] ^ RK[10];
            }
            break;

        case 14:

            for( i = 0; i < 7; i++, RK += 8 )
            {
                RK[8]  = RK[0] ^ RCON[i] ^
                ( (uint32_t) FSb[ ( RK[7] >>  8 ) & 0xFF ]       ) ^
                ( (uint32_t) FSb[ ( RK[7] >> 16 ) & 0xFF ] <<  8 ) ^
                ( (uint32_t) FSb[ ( RK[7] >> 24 ) & 0xFF ] << 16 ) ^
                ( (uint32_t) FSb[ ( RK[7]       ) & 0xFF ] << 24 );

                RK[9]  = RK[1] ^ RK[8];
                RK[10] = RK[2] ^ RK[9];
                RK[11] = RK[3] ^ RK[10];
```

```
                RK[12] = RK[4] ^
                ( (uint32_t) FSb[ ( RK[11]       ) & 0xFF ]       ) ^
                ( (uint32_t) FSb[ ( RK[11] >>  8 ) & 0xFF ] <<  8 ) ^
                ( (uint32_t) FSb[ ( RK[11] >> 16 ) & 0xFF ] << 16 ) ^
                ( (uint32_t) FSb[ ( RK[11] >> 24 ) & 0xFF ] << 24 );

                RK[13] = RK[5] ^ RK[12];
                RK[14] = RK[6] ^ RK[13];
                RK[15] = RK[7] ^ RK[14];
            }
            break;
    }

    return( 0 );
}
#endif /* !MBEDTLS_AES_SETKEY_ENC_ALT */

/*
 * AES key schedule (decryption)
 */
#if !defined(MBEDTLS_AES_SETKEY_DEC_ALT)
int mbedtls_aes_setkey_dec( mbedtls_aes_context *ctx, const unsigned char *key,
                    unsigned int keybits )
{
    int i, j, ret;
    mbedtls_aes_context cty;
    uint32_t *RK;
    uint32_t *SK;

    mbedtls_aes_init( &cty );

#if defined(MBEDTLS_PADLOCK_C) && defined(MBEDTLS_PADLOCK_ALIGN16)
    if( aes_padlock_ace == -1 )
        aes_padlock_ace = mbedtls_padlock_has_support( MBEDTLS_PADLOCK_ACE );

    if( aes_padlock_ace )
        ctx->rk = RK = MBEDTLS_PADLOCK_ALIGN16( ctx->buf );
    else
#endif
    ctx->rk = RK = ctx->buf;

    /* Also checks keybits */
    if( ( ret = mbedtls_aes_setkey_enc( &cty, key, keybits ) ) != 0 )
        goto exit;

    ctx->nr = cty.nr;

#if defined(MBEDTLS_AESNI_C) && defined(MBEDTLS_HAVE_X86_64)
    if( mbedtls_aesni_has_support( MBEDTLS_AESNI_AES ) )
    {
        mbedtls_aesni_inverse_key( (unsigned char *) ctx->rk,
                           (const unsigned char *) cty.rk, ctx->nr );
        goto exit;
    }
#endif

    SK = cty.rk + cty.nr * 4;

    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;

    for( i = ctx->nr - 1, SK -= 8; i > 0; i--, SK -= 8 )
    {
        for( j = 0; j < 4; j++, SK++ )
        {
            *RK++ = AES_RT0( FSb[ ( *SK       ) & 0xFF ] ) ^
```

```
                    AES_RT1( FSb[ ( *SK >>  8 ) & 0xFF ] ) ^
                    AES_RT2( FSb[ ( *SK >> 16 ) & 0xFF ] ) ^
                    AES_RT3( FSb[ ( *SK >> 24 ) & 0xFF ] );
        }
    }

    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;
    *RK++ = *SK++;

exit:
    mbedtls_aes_free( &cty );

    return( ret );
}

#if defined(MBEDTLS_CIPHER_MODE_XTS)
static int mbedtls_aes_xts_decode_keys( const unsigned char *key,
                                        unsigned int keybits,
                                        const unsigned char **key1,
                                        unsigned int *key1bits,
                                        const unsigned char **key2,
                                        unsigned int *key2bits )
{
    const unsigned int half_keybits = keybits / 2;
    const unsigned int half_keybytes = half_keybits / 8;

    switch( keybits )
    {
        case 256: break;
        case 512: break;
        default : return( MBEDTLS_ERR_AES_INVALID_KEY_LENGTH );
    }

    *key1bits = half_keybits;
    *key2bits = half_keybits;
    *key1 = &key[0];
    *key2 = &key[half_keybytes];

    return 0;
}

int mbedtls_aes_xts_setkey_enc( mbedtls_aes_xts_context *ctx,
                                const unsigned char *key,
                                unsigned int keybits)
{
    int ret;
    const unsigned char *key1, *key2;
    unsigned int key1bits, key2bits;

    ret = mbedtls_aes_xts_decode_keys( key, keybits, &key1, &key1bits,
                                       &key2, &key2bits );
    if( ret != 0 )
        return( ret );

    /* Set the tweak key. Always set tweak key for the encryption mode. */
    ret = mbedtls_aes_setkey_enc( &ctx->tweak, key2, key2bits );
    if( ret != 0 )
        return( ret );

    /* Set crypt key for encryption. */
    return mbedtls_aes_setkey_enc( &ctx->crypt, key1, key1bits );
}

int mbedtls_aes_xts_setkey_dec( mbedtls_aes_xts_context *ctx,
                                const unsigned char *key,
                                unsigned int keybits)
{
```

```
    int ret;
    const unsigned char *key1, *key2;
    unsigned int key1bits, key2bits;

    ret = mbedtls_aes_xts_decode_keys( key, keybits, &key1, &key1bits,
                                       &key2, &key2bits );
    if( ret != 0 )
        return( ret );

    /* Set the tweak key. Always set tweak key for encryption. */
    ret = mbedtls_aes_setkey_enc( &ctx->tweak, key2, key2bits );
    if( ret != 0 )
        return( ret );

    /* Set crypt key for decryption. */
    return mbedtls_aes_setkey_dec( &ctx->crypt, key1, key1bits );
}
#endif /* MBEDTLS_CIPHER_MODE_XTS */

#endif /* !MBEDTLS_AES_SETKEY_DEC_ALT */

#define AES_FROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)         \
{                                                   \
    X0 = *RK++ ^ AES_FT0( ( Y0       ) & 0xFF ) ^   \
                 AES_FT1( ( Y1 >>  8 ) & 0xFF ) ^   \
                 AES_FT2( ( Y2 >> 16 ) & 0xFF ) ^   \
                 AES_FT3( ( Y3 >> 24 ) & 0xFF );    \
                                                    \
    X1 = *RK++ ^ AES_FT0( ( Y1       ) & 0xFF ) ^   \
                 AES_FT1( ( Y2 >>  8 ) & 0xFF ) ^   \
                 AES_FT2( ( Y3 >> 16 ) & 0xFF ) ^   \
                 AES_FT3( ( Y0 >> 24 ) & 0xFF );    \
                                                    \
    X2 = *RK++ ^ AES_FT0( ( Y2       ) & 0xFF ) ^   \
                 AES_FT1( ( Y3 >>  8 ) & 0xFF ) ^   \
                 AES_FT2( ( Y0 >> 16 ) & 0xFF ) ^   \
                 AES_FT3( ( Y1 >> 24 ) & 0xFF );    \
                                                    \
    X3 = *RK++ ^ AES_FT0( ( Y3       ) & 0xFF ) ^   \
                 AES_FT1( ( Y0 >>  8 ) & 0xFF ) ^   \
                 AES_FT2( ( Y1 >> 16 ) & 0xFF ) ^   \
                 AES_FT3( ( Y2 >> 24 ) & 0xFF );    \
}

#define AES_RROUND(X0,X1,X2,X3,Y0,Y1,Y2,Y3)         \
{                                                   \
    X0 = *RK++ ^ AES_RT0( ( Y0       ) & 0xFF ) ^   \
                 AES_RT1( ( Y3 >>  8 ) & 0xFF ) ^   \
                 AES_RT2( ( Y2 >> 16 ) & 0xFF ) ^   \
                 AES_RT3( ( Y1 >> 24 ) & 0xFF );    \
                                                    \
    X1 = *RK++ ^ AES_RT0( ( Y1       ) & 0xFF ) ^   \
                 AES_RT1( ( Y0 >>  8 ) & 0xFF ) ^   \
                 AES_RT2( ( Y3 >> 16 ) & 0xFF ) ^   \
                 AES_RT3( ( Y2 >> 24 ) & 0xFF );    \
                                                    \
    X2 = *RK++ ^ AES_RT0( ( Y2       ) & 0xFF ) ^   \
                 AES_RT1( ( Y1 >>  8 ) & 0xFF ) ^   \
                 AES_RT2( ( Y0 >> 16 ) & 0xFF ) ^   \
                 AES_RT3( ( Y3 >> 24 ) & 0xFF );    \
                                                    \
    X3 = *RK++ ^ AES_RT0( ( Y3       ) & 0xFF ) ^   \
                 AES_RT1( ( Y2 >>  8 ) & 0xFF ) ^   \
                 AES_RT2( ( Y1 >> 16 ) & 0xFF ) ^   \
                 AES_RT3( ( Y0 >> 24 ) & 0xFF );    \
}

/*
 * AES-ECB block encryption
```

```
 */
#if !defined(MBEDTLS_AES_ENCRYPT_ALT)
int mbedtls_internal_aes_encrypt( mbedtls_aes_context *ctx,
                                  const unsigned char input[16],
                                  unsigned char output[16] )
{
    int i;
    uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;

    RK = ctx->rk;

    GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
    GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
    GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
    GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;

    for( i = ( ctx->nr >> 1 ) - 1; i > 0; i-- )
    {
        AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
        AES_FROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
    }

    AES_FROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );

    X0 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y0       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );

    X1 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y1       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );

    X2 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y2       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );

    X3 = *RK++ ^ \
            ( (uint32_t) FSb[ ( Y3       ) & 0xFF ]       ) ^
            ( (uint32_t) FSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) FSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) FSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );

    PUT_UINT32_LE( X0, output,  0 );
    PUT_UINT32_LE( X1, output,  4 );
    PUT_UINT32_LE( X2, output,  8 );
    PUT_UINT32_LE( X3, output, 12 );

    return( 0 );
}
#endif /* !MBEDTLS_AES_ENCRYPT_ALT */

#if !defined(MBEDTLS_DEPRECATED_REMOVED)
void mbedtls_aes_encrypt( mbedtls_aes_context *ctx,
                          const unsigned char input[16],
                          unsigned char output[16] )
{
    mbedtls_internal_aes_encrypt( ctx, input, output );
}
#endif /* !MBEDTLS_DEPRECATED_REMOVED */

/*
 * AES-ECB block decryption
 */
```

```
#if !defined(MBEDTLS_AES_DECRYPT_ALT)
int mbedtls_internal_aes_decrypt( mbedtls_aes_context *ctx,
                                  const unsigned char input[16],
                                  unsigned char output[16] )
{
    int i;
    uint32_t *RK, X0, X1, X2, X3, Y0, Y1, Y2, Y3;

    RK = ctx->rk;

    GET_UINT32_LE( X0, input,  0 ); X0 ^= *RK++;
    GET_UINT32_LE( X1, input,  4 ); X1 ^= *RK++;
    GET_UINT32_LE( X2, input,  8 ); X2 ^= *RK++;
    GET_UINT32_LE( X3, input, 12 ); X3 ^= *RK++;

    for( i = ( ctx->nr >> 1 ) - 1; i > 0; i-- )
    {
        AES_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );
        AES_RROUND( X0, X1, X2, X3, Y0, Y1, Y2, Y3 );
    }

    AES_RROUND( Y0, Y1, Y2, Y3, X0, X1, X2, X3 );

    X0 = *RK++ ^ \
            ( (uint32_t) RSb[ ( Y0       ) & 0xFF ]       ) ^
            ( (uint32_t) RSb[ ( Y3 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) RSb[ ( Y2 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) RSb[ ( Y1 >> 24 ) & 0xFF ] << 24 );

    X1 = *RK++ ^ \
            ( (uint32_t) RSb[ ( Y1       ) & 0xFF ]       ) ^
            ( (uint32_t) RSb[ ( Y0 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) RSb[ ( Y3 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) RSb[ ( Y2 >> 24 ) & 0xFF ] << 24 );

    X2 = *RK++ ^ \
            ( (uint32_t) RSb[ ( Y2       ) & 0xFF ]       ) ^
            ( (uint32_t) RSb[ ( Y1 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) RSb[ ( Y0 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) RSb[ ( Y3 >> 24 ) & 0xFF ] << 24 );

    X3 = *RK++ ^ \
            ( (uint32_t) RSb[ ( Y3       ) & 0xFF ]       ) ^
            ( (uint32_t) RSb[ ( Y2 >>  8 ) & 0xFF ] <<  8 ) ^
            ( (uint32_t) RSb[ ( Y1 >> 16 ) & 0xFF ] << 16 ) ^
            ( (uint32_t) RSb[ ( Y0 >> 24 ) & 0xFF ] << 24 );

    PUT_UINT32_LE( X0, output,  0 );
    PUT_UINT32_LE( X1, output,  4 );
    PUT_UINT32_LE( X2, output,  8 );
    PUT_UINT32_LE( X3, output, 12 );

    return( 0 );
}
#endif /* !MBEDTLS_AES_DECRYPT_ALT */

#if !defined(MBEDTLS_DEPRECATED_REMOVED)
void mbedtls_aes_decrypt( mbedtls_aes_context *ctx,
                          const unsigned char input[16],
                          unsigned char output[16] )
{
    mbedtls_internal_aes_decrypt( ctx, input, output );
}
#endif /* !MBEDTLS_DEPRECATED_REMOVED */

/*
 * AES-ECB block encryption/decryption
 */
int mbedtls_aes_crypt_ecb( mbedtls_aes_context *ctx,
```

```
                    int mode,
                    const unsigned char input[16],
                    unsigned char output[16] )
{
#if defined(MBEDTLS_AESNI_C) && defined(MBEDTLS_HAVE_X86_64)
    if( mbedtls_aesni_has_support( MBEDTLS_AESNI_AES ) )
        return( mbedtls_aesni_crypt_ecb( ctx, mode, input, output ) );
#endif

#if defined(MBEDTLS_PADLOCK_C) && defined(MBEDTLS_HAVE_X86)
    if( aes_padlock_ace )
    {
        if( mbedtls_padlock_xcryptecb( ctx, mode, input, output ) == 0 )
            return( 0 );

        // If padlock data misaligned, we just fall back to
        // unaccelerated mode
        //
    }
#endif

    if( mode == MBEDTLS_AES_ENCRYPT )
        return( mbedtls_internal_aes_encrypt( ctx, input, output ) );
    else
        return( mbedtls_internal_aes_decrypt( ctx, input, output ) );
}

#if defined(MBEDTLS_CIPHER_MODE_CBC)
/*
 * AES-CBC buffer encryption/decryption
 */
int mbedtls_aes_crypt_cbc( mbedtls_aes_context *ctx,
                    int mode,
                    size_t length,
                    unsigned char iv[16],
                    const unsigned char *input,
                    unsigned char *output )
{
    int i;
    unsigned char temp[16];

    if( length % 16 )
        return( MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH );

#if defined(MBEDTLS_PADLOCK_C) && defined(MBEDTLS_HAVE_X86)
    if( aes_padlock_ace )
    {
        if( mbedtls_padlock_xcryptcbc( ctx, mode, length, iv, input, output ) == 0 )
            return( 0 );

        // If padlock data misaligned, we just fall back to
        // unaccelerated mode
        //
    }
#endif

    if( mode == MBEDTLS_AES_DECRYPT )
    {
        while( length > 0 )
        {
            memcpy( temp, input, 16 );
            mbedtls_aes_crypt_ecb( ctx, mode, input, output );

            for( i = 0; i < 16; i++ )
                output[i] = (unsigned char)( output[i] ^ iv[i] );

            memcpy( iv, temp, 16 );

            input  += 16;
```

```
            output += 16;
            length -= 16;
        }
    }
    else
    {
        while( length > 0 )
        {
            for( i = 0; i < 16; i++ )
                output[i] = (unsigned char)( input[i] ^ iv[i] );

            mbedtls_aes_crypt_ecb( ctx, mode, output, output );
            memcpy( iv, output, 16 );

            input  += 16;
            output += 16;
            length -= 16;
        }
    }

    return( 0 );
}
#endif /* MBEDTLS_CIPHER_MODE_CBC */

#if defined(MBEDTLS_CIPHER_MODE_XTS)

/* Endianess with 64 bits values */
#ifndef GET_UINT64_LE
#define GET_UINT64_LE(n,b,i)                            \
{                                                       \
    (n) = ( (uint64_t) (b)[(i) + 7] << 56 )             \
        | ( (uint64_t) (b)[(i) + 6] << 48 )             \
        | ( (uint64_t) (b)[(i) + 5] << 40 )             \
        | ( (uint64_t) (b)[(i) + 4] << 32 )             \
        | ( (uint64_t) (b)[(i) + 3] << 24 )             \
        | ( (uint64_t) (b)[(i) + 2] << 16 )             \
        | ( (uint64_t) (b)[(i) + 1] <<  8 )             \
        | ( (uint64_t) (b)[(i)    ]       );            \
}
#endif

#ifndef PUT_UINT64_LE
#define PUT_UINT64_LE(n,b,i)                            \
{                                                       \
    (b)[(i) + 7] = (unsigned char) ( (n) >> 56 );       \
    (b)[(i) + 6] = (unsigned char) ( (n) >> 48 );       \
    (b)[(i) + 5] = (unsigned char) ( (n) >> 40 );       \
    (b)[(i) + 4] = (unsigned char) ( (n) >> 32 );       \
    (b)[(i) + 3] = (unsigned char) ( (n) >> 24 );       \
    (b)[(i) + 2] = (unsigned char) ( (n) >> 16 );       \
    (b)[(i) + 1] = (unsigned char) ( (n) >>  8 );       \
    (b)[(i)    ] = (unsigned char) ( (n)       );       \
}
#endif

typedef unsigned char mbedtls_be128[16];

/*
 * GF(2^128) multiplication function
 *
 * This function multiplies a field element by x in the polynomial field
 * representation. It uses 64-bit word operations to gain speed but compensates
 * for machine endianess and hence works correctly on both big and little
 * endian machines.
 */
static void mbedtls_gf128mul_x_ble( unsigned char r[16],
                                    const unsigned char x[16] )
{
    uint64_t a, b, ra, rb;
```

```
    GET_UINT64_LE( a, x, 0 );
    GET_UINT64_LE( b, x, 8 );

    ra = ( a << 1 )  ^ 0x0087 >> ( 8 - ( ( b >> 63 ) << 3 ) );
    rb = ( a >> 63 ) | ( b << 1 );

    PUT_UINT64_LE( ra, r, 0 );
    PUT_UINT64_LE( rb, r, 8 );
}

/*
 * AES-XTS buffer encryption/decryption
 */
int mbedtls_aes_crypt_xts( mbedtls_aes_xts_context *ctx,
                           int mode,
                           size_t length,
                           const unsigned char data_unit[16],
                           const unsigned char *input,
                           unsigned char *output )
{
    int ret;
    size_t blocks = length / 16;
    size_t leftover = length % 16;
    unsigned char tweak[16];
    unsigned char prev_tweak[16];
    unsigned char tmp[16];

    /* Data units must be at least 16 bytes long. */
    if( length < 16 )
        return MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH;

    /* NIST SP 800-38E disallows data units larger than 2**20 blocks. */
    if( length > ( 1 << 20 ) * 16 )
        return MBEDTLS_ERR_AES_INVALID_INPUT_LENGTH;

    /* Compute the tweak. */
    ret = mbedtls_aes_crypt_ecb( &ctx->tweak, MBEDTLS_AES_ENCRYPT,
                                 data_unit, tweak );
    if( ret != 0 )
        return( ret );

    while( blocks-- )
    {
        size_t i;

        if( leftover && ( mode == MBEDTLS_AES_DECRYPT ) && blocks == 0 )
        {
            /* We are on the last block in a decrypt operation that has
             * leftover bytes, so we need to use the next tweak for this block,
             * and this tweak for the lefover bytes. Save the current tweak for
             * the leftovers and then update the current tweak for use on this,
             * the last full block. */
            memcpy( prev_tweak, tweak, sizeof( tweak ) );
            mbedtls_gf128mul_x_ble( tweak, tweak );
        }

        for( i = 0; i < 16; i++ )
            tmp[i] = input[i] ^ tweak[i];

        ret = mbedtls_aes_crypt_ecb( &ctx->crypt, mode, tmp, tmp );
        if( ret != 0 )
            return( ret );

        for( i = 0; i < 16; i++ )
            output[i] = tmp[i] ^ tweak[i];

        /* Update the tweak for the next block. */
        mbedtls_gf128mul_x_ble( tweak, tweak );
```

```
        output += 16;
        input += 16;
    }

    if( leftover )
    {
        /* If we are on the leftover bytes in a decrypt operation, we need to
         * use the previous tweak for these bytes (as saved in prev_tweak). */
        unsigned char *t = mode == MBEDTLS_AES_DECRYPT ? prev_tweak : tweak;

        /* We are now on the final part of the data unit, which doesn't divide
         * evenly by 16. It's time for ciphertext stealing. */
        size_t i;
        unsigned char *prev_output = output - 16;

        /* Copy ciphertext bytes from the previous block to our output for each
         * byte of cyphertext we won't steal. At the same time, copy the
         * remainder of the input for this final round (since the loop bounds
         * are the same). */
        for( i = 0; i < leftover; i++ )
        {
            output[i] = prev_output[i];
            tmp[i] = input[i] ^ t[i];
        }

        /* Copy ciphertext bytes from the previous block for input in this
         * round. */
        for( ; i < 16; i++ )
            tmp[i] = prev_output[i] ^ t[i];

        ret = mbedtls_aes_crypt_ecb( &ctx->crypt, mode, tmp, tmp );
        if( ret != 0 )
            return ret;

        /* Write the result back to the previous block, overriding the previous
         * output we copied. */
        for( i = 0; i < 16; i++ )
            prev_output[i] = tmp[i] ^ t[i];
    }

    return( 0 );
}
#endif /* MBEDTLS_CIPHER_MODE_XTS */

#if defined(MBEDTLS_CIPHER_MODE_CFB)
/*
 * AES-CFB128 buffer encryption/decryption
 */
int mbedtls_aes_crypt_cfb128( mbedtls_aes_context *ctx,
                      int mode,
                      size_t length,
                      size_t *iv_off,
                      unsigned char iv[16],
                      const unsigned char *input,
                      unsigned char *output )
{
    int c;
    size_t n = *iv_off;

    if( mode == MBEDTLS_AES_DECRYPT )
    {
        while( length-- )
        {
            if( n == 0 )
                mbedtls_aes_crypt_ecb( ctx, MBEDTLS_AES_ENCRYPT, iv, iv );

            c = *input++;
            *output++ = (unsigned char)( c ^ iv[n] );
```

```
            iv[n] = (unsigned char) c;

            n = ( n + 1 ) & 0x0F;
        }
    }
    else
    {
        while( length-- )
        {
            if( n == 0 )
                mbedtls_aes_crypt_ecb( ctx, MBEDTLS_AES_ENCRYPT, iv, iv );

            iv[n] = *output++ = (unsigned char)( iv[n] ^ *input++ );

            n = ( n + 1 ) & 0x0F;
        }
    }

    *iv_off = n;

    return( 0 );
}

/*
 * AES-CFB8 buffer encryption/decryption
 */
int mbedtls_aes_crypt_cfb8( mbedtls_aes_context *ctx,
                       int mode,
                       size_t length,
                       unsigned char iv[16],
                       const unsigned char *input,
                       unsigned char *output )
{
    unsigned char c;
    unsigned char ov[17];

    while( length-- )
    {
        memcpy( ov, iv, 16 );
        mbedtls_aes_crypt_ecb( ctx, MBEDTLS_AES_ENCRYPT, iv, iv );

        if( mode == MBEDTLS_AES_DECRYPT )
            ov[16] = *input;

        c = *output++ = (unsigned char)( iv[0] ^ *input++ );

        if( mode == MBEDTLS_AES_ENCRYPT )
            ov[16] = c;

        memcpy( iv, ov + 1, 16 );
    }

    return( 0 );
}
#endif /* MBEDTLS_CIPHER_MODE_CFB */

#if defined(MBEDTLS_CIPHER_MODE_OFB)
/*
 * AES-OFB (Output Feedback Mode) buffer encryption/decryption
 */
int mbedtls_aes_crypt_ofb( mbedtls_aes_context *ctx,
                           size_t length,
                           size_t *iv_off,
                           unsigned char iv[16],
                           const unsigned char *input,
                           unsigned char *output )
{
    int ret = 0;
    size_t n = *iv_off;
```

```
    while( length-- )
    {
        if( n == 0 )
        {
            ret = mbedtls_aes_crypt_ecb( ctx, MBEDTLS_AES_ENCRYPT, iv, iv );
            if( ret != 0 )
                goto exit;
        }
        *output++ =  *input++ ^ iv[n];

        n = ( n + 1 ) & 0x0F;
    }

    *iv_off = n;

exit:
    return( ret );
}
#endif /* MBEDTLS_CIPHER_MODE_OFB */

#if defined(MBEDTLS_CIPHER_MODE_CTR)
/*
 * AES-CTR buffer encryption/decryption
 */
int mbedtls_aes_crypt_ctr( mbedtls_aes_context *ctx,
                       size_t length,
                       size_t *nc_off,
                       unsigned char nonce_counter[16],
                       unsigned char stream_block[16],
                       const unsigned char *input,
                       unsigned char *output )
{
    int c, i;
    size_t n = *nc_off;

    if ( n > 0x0F )
        return( MBEDTLS_ERR_AES_BAD_INPUT_DATA );

    while( length-- )
    {
        if( n == 0 ) {
            mbedtls_aes_crypt_ecb( ctx, MBEDTLS_AES_ENCRYPT, nonce_counter, stream_block );

            for( i = 16; i > 0; i-- )
                if( ++nonce_counter[i - 1] != 0 )
                    break;
        }
        c = *input++;
        *output++ = (unsigned char)( c ^ stream_block[n] );

        n = ( n + 1 ) & 0x0F;
    }

    *nc_off = n;

    return( 0 );
}
#endif /* MBEDTLS_CIPHER_MODE_CTR */

#endif /* !MBEDTLS_AES_ALT */

#if defined(MBEDTLS_SELF_TEST)
/*
 * AES test vectors from:
 *
 * http://csrc.nist.gov/archive/aes/rijndael/rijndael-vals.zip
 */
static const unsigned char aes_test_ecb_dec[3][16] =
```

```
{
    { 0x44, 0x41, 0x6A, 0xC2, 0xD1, 0xF5, 0x3C, 0x58,
      0x33, 0x03, 0x91, 0x7E, 0x6B, 0xE9, 0xEB, 0xE0 },
    { 0x48, 0xE3, 0x1E, 0x9E, 0x25, 0x67, 0x18, 0xF2,
      0x92, 0x29, 0x31, 0x9C, 0x19, 0xF1, 0x5B, 0xA4 },
    { 0x05, 0x8C, 0xCF, 0xFD, 0xBB, 0xCB, 0x38, 0x2D,
      0x1F, 0x6F, 0x56, 0x58, 0x5D, 0x8A, 0x4A, 0xDE }
};

static const unsigned char aes_test_ecb_enc[3][16] =
{
    { 0xC3, 0x4C, 0x05, 0x2C, 0xC0, 0xDA, 0x8D, 0x73,
      0x45, 0x1A, 0xFE, 0x5F, 0x03, 0xBE, 0x29, 0x7F },
    { 0xF3, 0xF6, 0x75, 0x2A, 0xE8, 0xD7, 0x83, 0x11,
      0x38, 0xF0, 0x41, 0x56, 0x06, 0x31, 0xB1, 0x14 },
    { 0x8B, 0x79, 0xEE, 0xCC, 0x93, 0xA0, 0xEE, 0x5D,
      0xFF, 0x30, 0xB4, 0xEA, 0x21, 0x63, 0x6D, 0xA4 }
};

#if defined(MBEDTLS_CIPHER_MODE_CBC)
static const unsigned char aes_test_cbc_dec[3][16] =
{
    { 0xFA, 0xCA, 0x37, 0xE0, 0xB0, 0xC8, 0x53, 0x73,
      0xDF, 0x70, 0x6E, 0x73, 0xF7, 0xC9, 0xAF, 0x86 },
    { 0x5D, 0xF6, 0x78, 0xDD, 0x17, 0xBA, 0x4E, 0x75,
      0xB6, 0x17, 0x68, 0xC6, 0xAD, 0xEF, 0x7C, 0x7B },
    { 0x48, 0x04, 0xE1, 0x81, 0x8F, 0xE6, 0x29, 0x75,
      0x19, 0xA3, 0xE8, 0x8C, 0x57, 0x31, 0x04, 0x13 }
};

static const unsigned char aes_test_cbc_enc[3][16] =
{
    { 0x8A, 0x05, 0xFC, 0x5E, 0x09, 0x5A, 0xF4, 0x84,
      0x8A, 0x08, 0xD3, 0x28, 0xD3, 0x68, 0x8E, 0x3D },
    { 0x7B, 0xD9, 0x66, 0xD5, 0x3A, 0xD8, 0xC1, 0xBB,
      0x85, 0xD2, 0xAD, 0xFA, 0xE8, 0x7B, 0xB1, 0x04 },
    { 0xFE, 0x3C, 0x53, 0x65, 0x3E, 0x2F, 0x45, 0xB5,
      0x6F, 0xCD, 0x88, 0xB2, 0xCC, 0x89, 0x8F, 0xF0 }
};
#endif /* MBEDTLS_CIPHER_MODE_CBC */

#if defined(MBEDTLS_CIPHER_MODE_CFB)
/*
 * AES-CFB128 test vectors from:
 *
 * http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf
 */
static const unsigned char aes_test_cfb128_key[3][32] =
{
    { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
      0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C },
    { 0x8E, 0x73, 0xB0, 0xF7, 0xDA, 0x0E, 0x64, 0x52,
      0xC8, 0x10, 0xF3, 0x2B, 0x80, 0x90, 0x79, 0xE5,
      0x62, 0xF8, 0xEA, 0xD2, 0x52, 0x2C, 0x6B, 0x7B },
    { 0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4 }
};

static const unsigned char aes_test_cfb128_iv[16] =
{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
};

static const unsigned char aes_test_cfb128_pt[64] =
{
    0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
```

```
    0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A,
    0xAE, 0x2D, 0x8A, 0x57, 0x1E, 0x03, 0xAC, 0x9C,
    0x9E, 0xB7, 0x6F, 0xAC, 0x45, 0xAF, 0x8E, 0x51,
    0x30, 0xC8, 0x1C, 0x46, 0xA3, 0x5C, 0xE4, 0x11,
    0xE5, 0xFB, 0xC1, 0x19, 0x1A, 0x0A, 0x52, 0xEF,
    0xF6, 0x9F, 0x24, 0x45, 0xDF, 0x4F, 0x9B, 0x17,
    0xAD, 0x2B, 0x41, 0x7B, 0xE6, 0x6C, 0x37, 0x10
};

static const unsigned char aes_test_cfb128_ct[3][64] =
{
    { 0x3B, 0x3F, 0xD9, 0x2E, 0xB7, 0x2D, 0xAD, 0x20,
      0x33, 0x34, 0x49, 0xF8, 0xE8, 0x3C, 0xFB, 0x4A,
      0xC8, 0xA6, 0x45, 0x37, 0xA0, 0xB3, 0xA9, 0x3F,
      0xCD, 0xE3, 0xCD, 0xAD, 0x9F, 0x1C, 0xE5, 0x8B,
      0x26, 0x75, 0x1F, 0x67, 0xA3, 0xCB, 0xB1, 0x40,
      0xB1, 0x80, 0x8C, 0xF1, 0x87, 0xA4, 0xF4, 0xDF,
      0xC0, 0x4B, 0x05, 0x35, 0x7C, 0x5D, 0x1C, 0x0E,
      0xEA, 0xC4, 0xC6, 0x6F, 0x9F, 0xF7, 0xF2, 0xE6 },
    { 0xCD, 0xC8, 0x0D, 0x6F, 0xDD, 0xF1, 0x8C, 0xAB,
      0x34, 0xC2, 0x59, 0x09, 0xC9, 0x9A, 0x41, 0x74,
      0x67, 0xCE, 0x7F, 0x7F, 0x81, 0x17, 0x36, 0x21,
      0x96, 0x1A, 0x2B, 0x70, 0x17, 0x1D, 0x3D, 0x7A,
      0x2E, 0x1E, 0x8A, 0x1D, 0xD5, 0x9B, 0x88, 0xB1,
      0xC8, 0xE6, 0x0F, 0xED, 0x1E, 0xFA, 0xC4, 0xC9,
      0xC0, 0x5F, 0x9F, 0x9C, 0xA9, 0x83, 0x4F, 0xA0,
      0x42, 0xAE, 0x8F, 0xBA, 0x58, 0x4B, 0x09, 0xFF },
    { 0xDC, 0x7E, 0x84, 0xBF, 0xDA, 0x79, 0x16, 0x4B,
      0x7E, 0xCD, 0x84, 0x86, 0x98, 0x5D, 0x38, 0x60,
      0x39, 0xFF, 0xED, 0x14, 0x3B, 0x28, 0xB1, 0xC8,
      0x32, 0x11, 0x3C, 0x63, 0x31, 0xE5, 0x40, 0x7B,
      0xDF, 0x10, 0x13, 0x24, 0x15, 0xE5, 0x4B, 0x92,
      0xA1, 0x3E, 0xD0, 0xA8, 0x26, 0x7A, 0xE2, 0xF9,
      0x75, 0xA3, 0x85, 0x74, 0x1A, 0xB9, 0xCE, 0xF8,
      0x20, 0x31, 0x62, 0x3D, 0x55, 0xB1, 0xE4, 0x71 }
};
#endif /* MBEDTLS_CIPHER_MODE_CFB */

#if defined(MBEDTLS_CIPHER_MODE_OFB)
/*
 * AES-OFB test vectors from:
 *
 * https://csrc.nist.gov/publications/detail/sp/800-38a/final
 */
static const unsigned char aes_test_ofb_key[3][32] =
{
    { 0x2B, 0x7E, 0x15, 0x16, 0x28, 0xAE, 0xD2, 0xA6,
      0xAB, 0xF7, 0x15, 0x88, 0x09, 0xCF, 0x4F, 0x3C },
    { 0x8E, 0x73, 0xB0, 0xF7, 0xDA, 0x0E, 0x64, 0x52,
      0xC8, 0x10, 0xF3, 0x2B, 0x80, 0x90, 0x79, 0xE5,
      0x62, 0xF8, 0xEA, 0xD2, 0x52, 0x2C, 0x6B, 0x7B },
    { 0x60, 0x3D, 0xEB, 0x10, 0x15, 0xCA, 0x71, 0xBE,
      0x2B, 0x73, 0xAE, 0xF0, 0x85, 0x7D, 0x77, 0x81,
      0x1F, 0x35, 0x2C, 0x07, 0x3B, 0x61, 0x08, 0xD7,
      0x2D, 0x98, 0x10, 0xA3, 0x09, 0x14, 0xDF, 0xF4 }
};

static const unsigned char aes_test_ofb_iv[16] =
{
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F
};

static const unsigned char aes_test_ofb_pt[64] =
{
    0x6B, 0xC1, 0xBE, 0xE2, 0x2E, 0x40, 0x9F, 0x96,
    0xE9, 0x3D, 0x7E, 0x11, 0x73, 0x93, 0x17, 0x2A,
    0xAE, 0x2D, 0x8A, 0x57, 0x1E, 0x03, 0xAC, 0x9C,
    0x9E, 0xB7, 0x6F, 0xAC, 0x45, 0xAF, 0x8E, 0x51,
```

```
    0x30, 0xC8, 0x1C, 0x46, 0xA3, 0x5C, 0xE4, 0x11,
    0xE5, 0xFB, 0xC1, 0x19, 0x1A, 0x0A, 0x52, 0xEF,
    0xF6, 0x9F, 0x24, 0x45, 0xDF, 0x4F, 0x9B, 0x17,
    0xAD, 0x2B, 0x41, 0x7B, 0xE6, 0x6C, 0x37, 0x10
};

static const unsigned char aes_test_ofb_ct[3][64] =
{
    { 0x3B, 0x3F, 0xD9, 0x2E, 0xB7, 0x2D, 0xAD, 0x20,
      0x33, 0x34, 0x49, 0xF8, 0xE8, 0x3C, 0xFB, 0x4A,
      0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
      0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25,
      0x97, 0x40, 0x05, 0x1e, 0x9c, 0x5f, 0xec, 0xf6,
      0x43, 0x44, 0xf7, 0xa8, 0x22, 0x60, 0xed, 0xcc,
      0x30, 0x4c, 0x65, 0x28, 0xf6, 0x59, 0xc7, 0x78,
      0x66, 0xa5, 0x10, 0xd9, 0xc1, 0xd6, 0xae, 0x5e },
    { 0xCD, 0xC8, 0x0D, 0x6F, 0xDD, 0xF1, 0x8C, 0xAB,
      0x34, 0xC2, 0x59, 0x09, 0xC9, 0x9A, 0x41, 0x74,
      0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
      0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01,
      0x8d, 0x9a, 0x9a, 0xea, 0xc0, 0xf6, 0x59, 0x6f,
      0x55, 0x9c, 0x6d, 0x4d, 0xaf, 0x59, 0xa5, 0xf2,
      0x6d, 0x9f, 0x20, 0x08, 0x57, 0xca, 0x6c, 0x3e,
      0x9c, 0xac, 0x52, 0x4b, 0xd9, 0xac, 0xc9, 0x2a },
    { 0xDC, 0x7E, 0x84, 0xBF, 0xDA, 0x79, 0x16, 0x4B,
      0x7E, 0xCD, 0x84, 0x86, 0x98, 0x5D, 0x38, 0x60,
      0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
      0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d,
      0x71, 0xab, 0x47, 0xa0, 0x86, 0xe8, 0x6e, 0xed,
      0xf3, 0x9d, 0x1c, 0x5b, 0xba, 0x97, 0xc4, 0x08,
      0x01, 0x26, 0x14, 0x1d, 0x67, 0xf3, 0x7b, 0xe8,
      0x53, 0x8f, 0x5a, 0x8b, 0xe7, 0x40, 0xe4, 0x84 }
};
#endif /* MBEDTLS_CIPHER_MODE_OFB */

#if defined(MBEDTLS_CIPHER_MODE_CTR)
/*
 * AES-CTR test vectors from:
 *
 * http://www.faqs.org/rfcs/rfc3686.html
 */

static const unsigned char aes_test_ctr_key[3][16] =
{
    { 0xAE, 0x68, 0x52, 0xF8, 0x12, 0x10, 0x67, 0xCC,
      0x4B, 0xF7, 0xA5, 0x76, 0x55, 0x77, 0xF3, 0x9E },
    { 0x7E, 0x24, 0x06, 0x78, 0x17, 0xFA, 0xE0, 0xD7,
      0x43, 0xD6, 0xCE, 0x1F, 0x32, 0x53, 0x91, 0x63 },
    { 0x76, 0x91, 0xBE, 0x03, 0x5E, 0x50, 0x20, 0xA8,
      0xAC, 0x6E, 0x61, 0x85, 0x29, 0xF9, 0xA0, 0xDC }
};

static const unsigned char aes_test_ctr_nonce_counter[3][16] =
{
    { 0x00, 0x00, 0x00, 0x30, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01 },
    { 0x00, 0x6C, 0xB6, 0xDB, 0xC0, 0x54, 0x3B, 0x59,
      0xDA, 0x48, 0xD9, 0x0B, 0x00, 0x00, 0x00, 0x01 },
    { 0x00, 0xE0, 0x01, 0x7B, 0x27, 0x77, 0x7F, 0x3F,
      0x4A, 0x17, 0x86, 0xF0, 0x00, 0x00, 0x00, 0x01 }
};

static const unsigned char aes_test_ctr_pt[3][48] =
{
    { 0x53, 0x69, 0x6E, 0x67, 0x6C, 0x65, 0x20, 0x62,
      0x6C, 0x6F, 0x63, 0x6B, 0x20, 0x6D, 0x73, 0x67 },

    { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
      0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
```

```
      0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
      0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F },

    { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
      0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
      0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
      0x18, 0x19, 0x1A, 0x1B, 0x1C, 0x1D, 0x1E, 0x1F,
      0x20, 0x21, 0x22, 0x23 }
};

static const unsigned char aes_test_ctr_ct[3][48] =
{
    { 0xE4, 0x09, 0x5D, 0x4F, 0xB7, 0xA7, 0xB3, 0x79,
      0x2D, 0x61, 0x75, 0xA3, 0x26, 0x13, 0x11, 0xB8 },
    { 0x51, 0x04, 0xA1, 0x06, 0x16, 0x8A, 0x72, 0xD9,
      0x79, 0x0D, 0x41, 0xEE, 0x8E, 0xDA, 0xD3, 0x88,
      0xEB, 0x2E, 0x1E, 0xFC, 0x46, 0xDA, 0x57, 0xC8,
      0xFC, 0xE6, 0x30, 0xDF, 0x91, 0x41, 0xBE, 0x28 },
    { 0xC1, 0xCF, 0x48, 0xA8, 0x9F, 0x2F, 0xFD, 0xD9,
      0xCF, 0x46, 0x52, 0xE9, 0xEF, 0xDB, 0x72, 0xD7,
      0x45, 0x40, 0xA4, 0x2B, 0xDE, 0x6D, 0x78, 0x36,
      0xD5, 0x9A, 0x5C, 0xEA, 0xAE, 0xF3, 0x10, 0x53,
      0x25, 0xB2, 0x07, 0x2F }
};

static const int aes_test_ctr_len[3] =
    { 16, 32, 36 };
#endif /* MBEDTLS_CIPHER_MODE_CTR */

#if defined(MBEDTLS_CIPHER_MODE_XTS)
/*
 * AES-XTS test vectors from:
 *
 * IEEE P1619/D16 Annex B
 * https://web.archive.org/web/20150629024421/http://grouper.ieee.org/groups/1619/email/pdf00086.
pdf
 * (Archived from original at http://grouper.ieee.org/groups/1619/email/pdf00086.pdf)
 */
static const unsigned char aes_test_xts_key[][32] =
{
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
      0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11,
      0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22,
      0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22 },
    { 0xff, 0xfe, 0xfd, 0xfc, 0xfb, 0xfa, 0xf9, 0xf8,
      0xf7, 0xf6, 0xf5, 0xf4, 0xf3, 0xf2, 0xf1, 0xf0,
      0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22,
      0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22, 0x22 },
};

static const unsigned char aes_test_xts_pt32[][32] =
{
    { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
    { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44 },
    { 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44,
      0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44, 0x44 },
};
```

```
static const unsigned char aes_test_xts_ct32[][32] =
{
    { 0x91, 0x7c, 0xf6, 0x9e, 0xbd, 0x68, 0xb2, 0xec,
      0x9b, 0x9f, 0xe9, 0xa3, 0xea, 0xdd, 0xa6, 0x92,
      0xcd, 0x43, 0xd2, 0xf5, 0x95, 0x98, 0xed, 0x85,
      0x8c, 0x02, 0xc2, 0x65, 0x2f, 0xbf, 0x92, 0x2e },
    { 0xc4, 0x54, 0x18, 0x5e, 0x6a, 0x16, 0x93, 0x6e,
      0x39, 0x33, 0x40, 0x38, 0xac, 0xef, 0x83, 0x8b,
      0xfb, 0x18, 0x6f, 0xff, 0x74, 0x80, 0xad, 0xc4,
      0x28, 0x93, 0x82, 0xec, 0xd6, 0xd3, 0x94, 0xf0 },
    { 0xaf, 0x85, 0x33, 0x6b, 0x59, 0x7a, 0xfc, 0x1a,
      0x90, 0x0b, 0x2e, 0xb2, 0x1e, 0xc9, 0x49, 0xd2,
      0x92, 0xdf, 0x4c, 0x04, 0x7e, 0x0b, 0x21, 0x53,
      0x21, 0x86, 0xa5, 0x97, 0x1a, 0x22, 0x7a, 0x89 },
};

static const unsigned char aes_test_xts_data_unit[][16] =
{
   { 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
   { 0x33, 0x33, 0x33, 0x33, 0x33, 0x00, 0x00, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
   { 0x33, 0x33, 0x33, 0x33, 0x33, 0x00, 0x00, 0x00,
     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 },
};

#endif /* MBEDTLS_CIPHER_MODE_XTS */

/*
 * Checkup routine
 */
int mbedtls_aes_self_test( int verbose )
{
    int ret = 0, i, j, u, mode;
    unsigned int keybits;
    unsigned char key[32];
    unsigned char buf[64];
    const unsigned char *aes_tests;
#if defined(MBEDTLS_CIPHER_MODE_CBC) || defined(MBEDTLS_CIPHER_MODE_CFB)
    unsigned char iv[16];
#endif
#if defined(MBEDTLS_CIPHER_MODE_CBC)
    unsigned char prv[16];
#endif
#if defined(MBEDTLS_CIPHER_MODE_CTR) || defined(MBEDTLS_CIPHER_MODE_CFB) || \
    defined(MBEDTLS_CIPHER_MODE_OFB)
    size_t offset;
#endif
#if defined(MBEDTLS_CIPHER_MODE_CTR) || defined(MBEDTLS_CIPHER_MODE_XTS)
    int len;
#endif
#if defined(MBEDTLS_CIPHER_MODE_CTR)
    unsigned char nonce_counter[16];
    unsigned char stream_block[16];
#endif
    mbedtls_aes_context ctx;

    memset( key, 0, 32 );
    mbedtls_aes_init( &ctx );

    /*
     * ECB mode
     */
    for( i = 0; i < 6; i++ )
    {
        u = i >> 1;
        keybits = 128 + u * 64;
        mode = i & 1;
```

```
        if( verbose != 0 )
            mbedtls_printf( "  AES-ECB-%3d (%s): ", keybits,
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memset( buf, 0, 16 );

        if( mode == MBEDTLS_AES_DECRYPT )
        {
            ret = mbedtls_aes_setkey_dec( &ctx, key, keybits );
            aes_tests = aes_test_ecb_dec[u];
        }
        else
        {
            ret = mbedtls_aes_setkey_enc( &ctx, key, keybits );
            aes_tests = aes_test_ecb_enc[u];
        }

        /*
         * AES-192 is an optional feature that may be unavailable when
         * there is an alternative underlying implementation i.e. when
         * MBEDTLS_AES_ALT is defined.
         */
        if( ret == MBEDTLS_ERR_PLATFORM_FEATURE_UNSUPPORTED && keybits == 192 )
        {
            mbedtls_printf( "skipped\n" );
            continue;
        }
        else if( ret != 0 )
        {
            goto exit;
        }

        for( j = 0; j < 10000; j++ )
        {
            ret = mbedtls_aes_crypt_ecb( &ctx, mode, buf, buf );
            if( ret != 0 )
                goto exit;
        }

        if( memcmp( buf, aes_tests, 16 ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );

#if defined(MBEDTLS_CIPHER_MODE_CBC)
    /*
     * CBC mode
     */
    for( i = 0; i < 6; i++ )
    {
        u = i >> 1;
        keybits = 128 + u * 64;
        mode = i & 1;

        if( verbose != 0 )
            mbedtls_printf( "  AES-CBC-%3d (%s): ", keybits,
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memset( iv , 0, 16 );
        memset( prv, 0, 16 );
```

```
        memset( buf, 0, 16 );

        if( mode == MBEDTLS_AES_DECRYPT )
        {
            ret = mbedtls_aes_setkey_dec( &ctx, key, keybits );
            aes_tests = aes_test_cbc_dec[u];
        }
        else
        {
            ret = mbedtls_aes_setkey_enc( &ctx, key, keybits );
            aes_tests = aes_test_cbc_enc[u];
        }

        /*
         * AES-192 is an optional feature that may be unavailable when
         * there is an alternative underlying implementation i.e. when
         * MBEDTLS_AES_ALT is defined.
         */
        if( ret == MBEDTLS_ERR_PLATFORM_FEATURE_UNSUPPORTED && keybits == 192 )
        {
            mbedtls_printf( "skipped\n" );
            continue;
        }
        else if( ret != 0 )
        {
            goto exit;
        }

        for( j = 0; j < 10000; j++ )
        {
            if( mode == MBEDTLS_AES_ENCRYPT )
            {
                unsigned char tmp[16];

                memcpy( tmp, prv, 16 );
                memcpy( prv, buf, 16 );
                memcpy( buf, tmp, 16 );
            }

            ret = mbedtls_aes_crypt_cbc( &ctx, mode, 16, iv, buf, buf );
            if( ret != 0 )
                goto exit;

        }

        if( memcmp( buf, aes_tests, 16 ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );
#endif /* MBEDTLS_CIPHER_MODE_CBC */

#if defined(MBEDTLS_CIPHER_MODE_CFB)
    /*
     * CFB128 mode
     */
    for( i = 0; i < 6; i++ )
    {
        u = i >> 1;
        keybits = 128 + u * 64;
        mode = i & 1;
```

```
        if( verbose != 0 )
            mbedtls_printf( "  AES-CFB128-%3d (%s): ", keybits,
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memcpy( iv,  aes_test_cfb128_iv, 16 );
        memcpy( key, aes_test_cfb128_key[u], keybits / 8 );

        offset = 0;
        ret = mbedtls_aes_setkey_enc( &ctx, key, keybits );
        /*
         * AES-192 is an optional feature that may be unavailable when
         * there is an alternative underlying implementation i.e. when
         * MBEDTLS_AES_ALT is defined.
         */
        if( ret == MBEDTLS_ERR_PLATFORM_FEATURE_UNSUPPORTED && keybits == 192 )
        {
            mbedtls_printf( "skipped\n" );
            continue;
        }
        else if( ret != 0 )
        {
            goto exit;
        }

        if( mode == MBEDTLS_AES_DECRYPT )
        {
            memcpy( buf, aes_test_cfb128_ct[u], 64 );
            aes_tests = aes_test_cfb128_pt;
        }
        else
        {
            memcpy( buf, aes_test_cfb128_pt, 64 );
            aes_tests = aes_test_cfb128_ct[u];
        }

        ret = mbedtls_aes_crypt_cfb128( &ctx, mode, 64, &offset, iv, buf, buf );
        if( ret != 0 )
            goto exit;

        if( memcmp( buf, aes_tests, 64 ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );
#endif /* MBEDTLS_CIPHER_MODE_CFB */

#if defined(MBEDTLS_CIPHER_MODE_OFB)
    /*
     * OFB mode
     */
    for( i = 0; i < 6; i++ )
    {
        u = i >> 1;
        keybits = 128 + u * 64;
        mode = i & 1;

        if( verbose != 0 )
            mbedtls_printf( "  AES-OFB-%3d (%s): ", keybits,
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memcpy( iv,  aes_test_ofb_iv, 16 );
        memcpy( key, aes_test_ofb_key[u], keybits / 8 );
```

```
        offset = 0;
        ret = mbedtls_aes_setkey_enc( &ctx, key, keybits );
        /*
         * AES-192 is an optional feature that may be unavailable when
         * there is an alternative underlying implementation i.e. when
         * MBEDTLS_AES_ALT is defined.
         */
        if( ret == MBEDTLS_ERR_PLATFORM_FEATURE_UNSUPPORTED && keybits == 192 )
        {
            mbedtls_printf( "skipped\n" );
            continue;
        }
        else if( ret != 0 )
        {
            goto exit;
        }

        if( mode == MBEDTLS_AES_DECRYPT )
        {
            memcpy( buf, aes_test_ofb_ct[u], 64 );
            aes_tests = aes_test_ofb_pt;
        }
        else
        {
            memcpy( buf, aes_test_ofb_pt, 64 );
            aes_tests = aes_test_ofb_ct[u];
        }

        ret = mbedtls_aes_crypt_ofb( &ctx, 64, &offset, iv, buf, buf );
        if( ret != 0 )
            goto exit;

        if( memcmp( buf, aes_tests, 64 ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );
#endif /* MBEDTLS_CIPHER_MODE_OFB */

#if defined(MBEDTLS_CIPHER_MODE_CTR)
    /*
     * CTR mode
     */
    for( i = 0; i < 6; i++ )
    {
        u = i >> 1;
        mode = i & 1;

        if( verbose != 0 )
            mbedtls_printf( "  AES-CTR-128 (%s): ",
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memcpy( nonce_counter, aes_test_ctr_nonce_counter[u], 16 );
        memcpy( key, aes_test_ctr_key[u], 16 );

        offset = 0;
        if( ( ret = mbedtls_aes_setkey_enc( &ctx, key, 128 ) ) != 0 )
            goto exit;

        len = aes_test_ctr_len[u];
```

```
        if( mode == MBEDTLS_AES_DECRYPT )
        {
            memcpy( buf, aes_test_ctr_ct[u], len );
            aes_tests = aes_test_ctr_pt[u];
        }
        else
        {
            memcpy( buf, aes_test_ctr_pt[u], len );
            aes_tests = aes_test_ctr_ct[u];
        }

        ret = mbedtls_aes_crypt_ctr( &ctx, len, &offset, nonce_counter,
                                     stream_block, buf, buf );
        if( ret != 0 )
            goto exit;

        if( memcmp( buf, aes_tests, len ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );
#endif /* MBEDTLS_CIPHER_MODE_CTR */

#if defined(MBEDTLS_CIPHER_MODE_XTS)
    {
    static const int num_tests =
        sizeof(aes_test_xts_key) / sizeof(*aes_test_xts_key);
    mbedtls_aes_xts_context ctx_xts;

    /*
     * XTS mode
     */
    mbedtls_aes_xts_init( &ctx_xts );

    for( i = 0; i < num_tests << 1; i++ )
    {
        const unsigned char *data_unit;
        u = i >> 1;
        mode = i & 1;

        if( verbose != 0 )
            mbedtls_printf( "  AES-XTS-128 (%s): ",
                            ( mode == MBEDTLS_AES_DECRYPT ) ? "dec" : "enc" );

        memset( key, 0, sizeof( key ) );
        memcpy( key, aes_test_xts_key[u], 32 );
        data_unit = aes_test_xts_data_unit[u];

        len = sizeof( *aes_test_xts_ct32 );

        if( mode == MBEDTLS_AES_DECRYPT )
        {
            ret = mbedtls_aes_xts_setkey_dec( &ctx_xts, key, 256 );
            if( ret != 0)
                goto exit;
            memcpy( buf, aes_test_xts_ct32[u], len );
            aes_tests = aes_test_xts_pt32[u];
        }
        else
        {
            ret = mbedtls_aes_xts_setkey_enc( &ctx_xts, key, 256 );
            if( ret != 0)
```

```
                goto exit;
            memcpy( buf, aes_test_xts_pt32[u], len );
            aes_tests = aes_test_xts_ct32[u];
        }


        ret = mbedtls_aes_crypt_xts( &ctx_xts, mode, len, data_unit,
                                     buf, buf );
        if( ret != 0 )
            goto exit;

        if( memcmp( buf, aes_tests, len ) != 0 )
        {
            ret = 1;
            goto exit;
        }

        if( verbose != 0 )
            mbedtls_printf( "passed\n" );
    }

    if( verbose != 0 )
        mbedtls_printf( "\n" );

    mbedtls_aes_xts_free( &ctx_xts );
    }
#endif /* MBEDTLS_CIPHER_MODE_XTS */

    ret = 0;

exit:
    if( ret != 0 && verbose != 0 )
        mbedtls_printf( "failed\n" );

    mbedtls_aes_free( &ctx );

    return( ret );
}

#endif /* MBEDTLS_SELF_TEST */

#endif /* MBEDTLS_AES_C */
```