AN14753

How to Use ECC Opaque Key Using CAAM for ECDSA on i.MX RT1170

Rev. 1.0 — 18 July 2025 Application note

Document information

Information	Content
Keywords	AN14753, CAAM, ECC, ECDSA, i.MX RT1170
Abstract	This application note describes how to generate a user-invisible ECC key pairs on the NXP i.MX RT1170 microcontroller using the CAAM hardware module to enhance cryptographic protection in customer device.

1 Introduction

In secure IoT devices, managing session keys securely becomes challenging when plaintext keys are exposed to an application software.

This application note presents a secure method for using the Cryptographic Acceleration and Assurance Module (CAAM) on the i.MX RT1170 to generate elliptic curve cryptography (ECC) key pairs. The method ensures that the plain keys remain invisible to the user and inaccessible to software, significantly reducing the risk of key exposure. The document also describes CAAM protocol commands and provides corresponding C implementations for the key generation process.

2 CAAM overview

Software's primary interaction with CAAM is through the submission of descriptor commands. It is up to the developers to provide meaningful descriptor commands for execution.

A job descriptor (JD) is a control structure that causes CAAM to execute a single job, consisting of one or more CAAM commands. A JD is composed of multiple 32-bit words, each representing a specific command, or parameter. Typical components include:

- **Header**: defines the basic information about the descriptor itself, such as type and length
- **Optional data**: for example, protocol data block for operation command
- **Operation command**: indicates the cryptographic function to execute

2.1 HEADER command

Every descriptor begins with a HEADER command. The formats of the JD HEADER command are illustrated in the tables below.

Table 1. Job descriptor header command format

31-27	26-24	23	22	21-16
CTYPE = 10110b	Not used in this doc	One	Reserved	START INDEX / SHR DESCR LENGTH
15-6				5-00
In this document, these bits have been set to 0				DESCLEN

Table 2. Job descriptor header field descriptions

Job descriptor header fields	Description
31-27 CTYPE	Command type 10110b job descriptor HEADER
23 One	The ONE bit is always 1.
21-16 START INDEX/SHR DESCR LENGTH	The SHR DESCR LENGTH field is not used in this document. The start index specifies the position of the word in the descriptor buffer where execution of the JD should continue following execution of the JD HEADER.
5-0 DESCLEN	Descriptor length This field represents the total length in 4-byte words of the descriptor.

Table 2. Job descriptor header field descriptions*...continued*

Job descriptor header fields	Description
	The header word is included in the length.

2.2 PROTOCOL OPERATION command

The PROTOCOL OPERATION command defines the type of cryptographic operation that CAAM performs.

The command format for ECC key pair is described in the following tables.

Table 3. PROTOCOL OPERATION command format

31-27	26-24	23-16
CTYPE = 10000b	OPTYPE = 000b	PROTID
15-0		
PROTINFO		

Table 4. Field PROTID descriptions

PROTID	Description
0x14	OPTYPE 000: DH, DSA, and ECC key pair generation (discrete-log key-pair generation)
0x15	OPTYPE 000: (EC)DSA_Sign
0x16	OPTYPE 000: (EC)DSA_Verify

Table 5. PROTINFO format when used with discrete log protocol

	15-13	12	11-10	9-8	7	6	5	4	3	2	1	0
Format for Sign function	Reserved	SIGN_NO_TEQ	MES_REP	HASH		SIGN_2ND_HALF_ONLY	SIGN_1ST_HALF_ONLY	EXT_PRI	TEST	ENC_PRI	ECC/DL	F2M/Fp
Format for Verify function	Reserved		MES_REP	HASH		Reserved					ECC/DL	F2M/Fp
Format for Keypair Generation	Reserved				KPG_IETF_DH	EKT_Z	ENC_Z	EXT_PRI	KPG_NO_TEQ	ENC_PRI	ECC/DL	F2M/Fp

Table 6. PROTINFO field descriptions

Field	Description
SIGN_NO_TEQ	For signature generation (SIGN) protocol and MPSign, disable timing equalization during SIGN. 0: Run SIGN using normal timing equalization protection. 1: Run SIGN with no timing equalization protection.
MES_REP	For signature generation (SIGN) and verification (VERIFY) protocols, this field indicates the format of the message. 00: F input is a message representative.

Table 6. PROTINFO field descriptions*...continued*

Field	Description
	01: Calculate the message representative from the message (using a SEQ IN PTR command), and the hash function specified by the HASH field. The message representative is calculated using the equivalent of EMSA1 (IEEE-1363). 10: F input is a hashed message, with length specified in the PDB. The protocol formats the message as required. 11: Reserved.
HASH	The hash function used to calculate a message representative from a message; valid when MES_REP=01. 000 MD5 001 SHA-1 010 SHA-224 011 SHA-256 100 SHA-384 101 SHA-512 110 SHA-512/224 111 SHA-512/256
KPG_IETF_DH	For KPG, this bit enables running IETF_style DH 0: No IETF-style DH. 1: Run KPG with IETF-style Diffie-Hellman.
SIGN_2ND_HALF_ONLY	For signature generation (SIGN) protocol only; otherwise reserved. Run 2nd half (signature 'd' generation) only. 0: Run full SIGN or 1st half, depending on the SIGN_1ST_HALF_ONLY setting. 1: Run 2nd half of SIGN only, generating 'd' result. Requires SIGN_1ST_HALF_ONLY = 0.
SIGN_1ST_HALF_ONLY	For signature generation (SIGN) protocol only; otherwise reserved. Run 1st half (signature 'c' generation) only. 0: Run full SIGN or 2nd half, depending on the SIGN_2ND_HALF_ONLY setting. 1: Run 1st half of SIGN only, generating 'c' result. Requires SIGN_2ND_HALF_ONLY = 0.
EKT_Z	If ENC_Z=1, key encryption type (used only with DH; otherwise reserved.) 0: Secret output is encrypted with AES-ECB mode. 1: Secret output is encrypted with AES-CCM mode.
ENC_Z	Encrypt the DH shared secret (used only with DH; otherwise reserved.) 0: The DH output is public and is unencrypted. 1: The DH output is secret and encrypted.
EXT_PRI	If ENC_PRI=1, encrypted key type for private key. 0: The private key is encrypted with AES-ECB mode. 1: The private key is encrypted with AES-CCM mode.
KPG_NO_TEQ	KPG_NO_TEQ For KPG, MPPrivK, and MPPubK. 0: Key pair generation runs with timing equalization protection. 1: Key pair generation runs with timing equalization disabled.
TEST	TEST 0: Signature generation protects the per message secret. 1: Signature generation outputs the per message secret to aid in testing and verification. This is not allowed in trusted or secure states.
ENC_PRI	Encrypted private key 0: The private key is not encrypted. ENC_PRI must be 0 if SIGN_2ND_HALF_ONLY=1.

Table 6. PROTINFO field descriptions*...continued*

Field	Description
	1: The private key must be decrypted before use (see KEY command for further information.). For key generation, this causes the private key to be encrypted.
ECC/DL	Public key operation type 0 DL: Discrete Log 1 ECC: Elliptic curve cryptography
F2M/Fp	Finite field type 0 Fp: Prime field 1 F2M: Binary field

2.3 ECC domain curves

When executing an ECC function, the ECC domain curve must be specified. If the predefined domain (PD) bit in the function's protocol data block (PDB) is 0, the curve parameters are supplied via the PDB. But when PD is 1, the elliptic curve domain selection (ECDSEL) field in the PDB is used to select one of the built-in ECC domains.

Most of the curve parameters are supplied by the hardware. For a complete list of supported built-in ECC domains, refer to Table 6-100 in the *Security Reference Manual for the i.MX RT1170 Processor* (document IMXRT1170SRM).

This document uses the built-in ECC curve **secp256r1 (0x2)**, defined over a prime field (Fp).

2.4 Key protection

If an encrypted private key is requested during the key-pair generation process, the private key is encrypted using the job descriptor key encryption key (JDKEK). The encryption can be performed using either AES-ECB or AES-CCM mode, and the resulting encrypted key is referred to as a black key.

A black key encrypted using CCM mode is always at least 12 bytes longer than the original key. This is because the encapsulation process uses a 6-byte nonce and appends a 6-byte integrity check value (ICV). If the plain key is not already a multiple of 8 bytes in length, it is padded accordingly to ensure 8-byte alignment. The final black key includes the padded plain key, plus the 12 additional bytes from the nonce and ICV.

When CAAM is instructed to use a black key in a cryptographic operation, it automatically decrypts the black key internally and loads the result into a protected key register. The decrypted key is never exposed to software and is used only within the CAAM hardware for the specified operation.

The JDKEK values are not persistent across power cycles. Upon each power on, new 256-bit JDKEKs are generated internally from the true random number generator (TRNG). As a result, black keys created in one power session cannot be decrypted in subsequent sessions, making them unsuitable for persistent key storage.

For securely storing keys across chip resets or power cycles, CAAM provides a dedicated blob mechanism. This mechanism encrypts key material in a format that can be decrypted in future sessions under controlled conditions.

2.5 ECC key-pair generation

The protocol data block (PDB) with the PD field set to 1 is defined as follows:

Table 7. Public-key generation protocol data block

PDB word 1	SGF(bits 31..26) (Not used in this document)	PD (=1) (bit 25)	Reserved (bits 24..14)	ECC curve field (bits 13..7)	Reserved (bits 6..0)

Table 7. Public-key generation protocol data block*...continued*

PDB word 2	Pointer points to private key (s)
PDB word 3	Pointer points to public key

2.6 Generating ECDSA signatures

CAAM can take either a message or a message representative as input, controlled by the MSG_REP bit in the OPERATION command.

There are three different commands that can be used when generating an ECDSA signature. The full sign command performs the entire ECDSA signature operation. The first-half sign command performs the first half of the signature operation, which does not require as input the message that is to be signed. The second-half sign command takes the output of the first-half sign command and the message representative and completes the signature operation.

This document uses the full sign command to execute the entire ECDSA signature process.

The PDB with the PD field set to 1 is defined as follows:

Table 8. ECDSA Signature Generation protocol data block (first-half sign)

PDB word 1	SGF(bits 31..23) (Not used in this doc)	PD (=1) (bit 22)	Reserved (bits 21..14)	ECC curve field (bits 13..7)	Reserved (bits 6..0)
PDB word 2	Pointer points to the first part of digital signature				
PDB word 3	Pointer points to the second part of digital signature				
PDB word 4	Pointer points to random number included only if TEST=1				

Table 9. ECDSA Signature Generation protocol data block (full sign or second-half sign)

PDB word 1	SGF(bits 31..23) (Not used in this doc)	PD (=1) (bit 22)	Reserved (bits 21..14)	ECC curve field (bits 13..7)	Reserved (bits 6..0)
PDB word 2	Pointer points to private key (s)				
PDB word 3	Pointer points to message representative (typically the hash of the message) (MES_REP=0) or actual message (MES_REP=1)				
PDB word 4	Pointer points to the first part of digital signature				
PDB word 5	Pointer points to the second part of digital signature. The buffer for this parameter must be a multiple of 16 bytes, as it is used to store an encrypted intermediate result, which may include padding.				
PDB word 6	Pointer points to random number included only if TEST=1				
PDB word 7	Message length (One 32-bit word) included if MES_REP!=0, else this word is omitted from the PDB				

2.7 Verifying ECDSA signatures

CAAM implements digital signature algorithm (DSA) verification functionality, including support for ECDSA in both prime fields and binary fields. The module can process either raw messages or message representatives as input, with selection controlled by the MSG_REP bit in the OPERATION command.

There are two versions of the ECDSA signature verification command. One version takes the public key of the signer as an input. The other version takes the private key of the signer as an input.

Table 10. ECDSA signature verification protocol data block

PDB word 1	SGF (bits 31..23) (Not used in this doc)	PD (=1) (bit 22)	Reserved (bits 21..14)	ECC curve field (bits 13..7)	Reserved (bits 6..0)
PDB word 2	Verification with public key: Pointer to public key. Verification with private key: Pointer to private key.				
PDB word 3	Pointer to message representative (typically the hash of the message) (MES_REP=0) or actual message (MES_REP=1).				
PDB word 4	Pointer points to the first part of a digital signature.				
PDB word 5	Pointer points to the second part of a digital signature.				
PDB word 6	Pointer points to temporary storage for intermediate results if verification with public key, otherwise this word is omitted from the PDB.				
PDB word 7	Message length (One 32-bit word) included if MES_REP != 0.				

3 Implementation

This section provides reference C code implementations for the CAAM ECC protocol commands described in Section 2. Some macros used in the code are available in the RT1170 SDK. Users can integrate the ECC-related functions into the CAAM driver source file, *fsl_caam.c*. The CAAM driver demo included in the SDK serves as an excellent starting point for understanding and integrating these example codes. It is located at: *<SDK_PATH>\boards\evkbmimxrt1170\driver_examples\caam\*.

3.1 ECC key-pair generation

The following function uses the CAAM's built-in ECC prime field curves. For encrypted private key generation, the output buffer has sufficient capacity to hold both the encrypted key. The generated ECC key pairs can be used for later cryptographic operations.

```
/* ECC key-pair generation */
static const uint32_t templateDLKeypairGenHW[] = {
    /* 00 */ 0xB0800000u, /* HEADER */
    /* 01 */ 0x02000000u, /* Built-in ECC domains */
    /* 02 */ 0x00000000u, /* pointer to private key buffer */
    /* 03 */ 0x00000000u, /* pointer to public key buffer */
    /* 04 */ 0x80140000u, /* OPERATION: ECC Key Pair Generation Operation, Prime
 Field */
};
status_t CAAM_ECC_Keypairs_Generation(CAAM_Type *base,
                                      caam_handle_t *handle,
                                      uint8_t curve_sel,
                                      bool enc_private,
                                      uint8_t *priv_key,
                                      uint8_t *pub_key)
{
    status_t status = kStatus_InvalidArgument;
    uint32_t descriptor[ARRAY_SIZE(templateDLKeypairGenHW)];

    /* create job descriptor */
    BUILD_ASSURE(sizeof(templateDLKeypairGenHW) <= sizeof(descriptor),
 descriptor_size);
    uint32_t descriptorSize = ARRAY_SIZE(templateDLKeypairGenHW);
    (void)caam_memcpy(descriptor, templateDLKeypairGenHW,
 sizeof(templateDLKeypairGenHW));
    /* Add descriptor HEADER len and index */
    descriptor[0] |= descriptorSize | (((descriptorSize - 1) & 0x3F) << 16);
    descriptor[1] |= (curve_sel << 7);
    descriptor[2] = (uint32_t)ADD_OFFSET((uint32_t)priv_key);
    descriptor[3] = (uint32_t)ADD_OFFSET((uint32_t)pub_key);
    descriptor[4] |= ECDSA_PROTINFO_KPG_ECC | ECDSA_PROTINFO_KPG_NO_TEQ;

    /* Operation: Encrypted ECC private key with CCM mode */
    if (enc_private)
    {
        descriptor[4] |= ECDSA_PROTINFO_KPG_ENC_PRI |
 ECDSA_PROTINFO_KPG_ENC_CCM;
    }

    /* schedule the job and block wait for result */
    do
    {
        status = caam_in_job_ring_add(base, handle->jobRing, &descriptor[0]);
    } while (status != kStatus_Success);

    status = CAAM_Wait(base, handle, &descriptor[0], kCAAM_Blocking);
    return status;
}
```

3.2 Generating elliptic curve digital signature algorithm (ECDSA) signatures

The below function shows how to sign the actual message with the elliptic curve cryptography (ECC) private key.

```
#define CAAM_ECDSA_PD                   (1 << 22)
#define ECDSA_PROTINFO_ECC                (1 << 1)
#define ECDSA_PROTINFO_SIGN_ENC_PRI       (1 << 2)
#define ECDSA_PROTINFO_SIGN_ENC_CCM       (1 << 4)
#define ECDSA_PROTINFO_HASH_SHA256        (3 << 7)
#define ECDSA_PROTINFO_MES_REP            (1 << 10)
#define ECDSA_PROTINFO_SIGN_NO_TEQ        (1 << 12)
static const uint32_t templateEcdsaSign[] = {
    /* 00 */ 0xB0800000u, /* HEADER */
    /* 01 */ 0x00000000u, /* Built-in ECC domains */
    /* 02 */ 0x00000000u, /* pointer to private key buffer */
    /* 03 */ 0x00000000u, /* place: a pointer to message address to be loaded */
    /* 04 */ 0x00000000u, /* sign_c: Signature in the format (c, d) */
    /* 05 */ 0x00000000u, /* sign_d: Signature in the format (c, d) */
    /* 06 */ 0x00000000u, /* message length */
    /* 07 */ 0x80150000u, /* OPERATION: ECDSA sign Operation */
};
status_t CAAM_ECDSA_Sign(CAAM_Type *base,
                         caam_handle_t *handle,
                         uint8_t curve_sel,
                         bool enc_private,
                         uint8_t *msg,
                         uint32_t msg_len,
                         uint8_t *priv_key,
                         uint8_t *sign_c,
                         uint8_t *sign_d)
{
    status_t status = kStatus_InvalidArgument;
    uint32_t descriptor[ARRAY_SIZE(templateEcdsaSign)];

    /* create job descriptor */
    BUILD_ASSURE(sizeof(templateEcdsaSign) <= sizeof(descriptor),
 templateEcdsaSign_descriptor);
    uint32_t descriptorSize = ARRAY_SIZE(templateEcdsaSign);

    (void)caam_memcpy(descriptor, templateEcdsaSign, sizeof(templateEcdsaSign));
    /* Add descriptor HEADER len and index */
    descriptor[0] |= descriptorSize | (((descriptorSize - 1) & 0x3F) << 16);
    descriptor[1] |= (curve_sel << 7) | CAAM_ECDSA_PD;
    descriptor[2] = (uint32_t)ADD_OFFSET((uint32_t)priv_key);
    descriptor[3] = (uint32_t)ADD_OFFSET((uint32_t)msg);
    descriptor[4] = (uint32_t)ADD_OFFSET((uint32_t)sign_c);
    descriptor[5] = (uint32_t)ADD_OFFSET((uint32_t)sign_d);
    descriptor[6] = msg_len;
    /* Operation:sha256, Encrypted ECC private key with CCM mode,  */
    descriptor[7] |= ECDSA_PROTINFO_ECC | ECDSA_PROTINFO_MES_REP |
                ECDSA_PROTINFO_HASH_SHA256 | ECDSA_PROTINFO_SIGN_NO_TEQ;

    /* Operation: Encrypted ECC private key with CCM mode */
    if (enc_private)
    {
        descriptor[7] |= ECDSA_PROTINFO_SIGN_ENC_PRI |
 ECDSA_PROTINFO_SIGN_ENC_CCM;
    }

    /* schedule the job and block wait for result */
    do
    {
        status = caam_in_job_ring_add(base, handle->jobRing, &descriptor[0]);
    } while (status != kStatus_Success);

    status = CAAM_Wait(base, handle, &descriptor[0], kCAAM_Blocking);
    return status;
}
```

3.3 Verifying ECDSA signatures

The below function shows how to verify the signature with the ECC public key.

```
static const uint32_t templateEcdsaVerify[] = {
    /* 00 */ 0xB0800000u, /* HEADER */
    /* 01 */ 0x00000000u, /* Built-in ECC domains */
    /* 02 */ 0x00000000u, /* pointer to private key buffer */
    /* 03 */ 0x00000000u, /* place: a pointer to message address to be loaded */
    /* 04 */ 0x00000000u, /* sign_c: Signature in the format (c, d) */
    /* 05 */ 0x00000000u, /* sign_d: Signature in the format (c, d) */
    /* 06 */ 0x00000000u, /* temp buffer */
    /* 07 */ 0x00000000u, /* message length */
    /* 08 */ 0x80160000u, /* OPERATION: ECDSA Verify Operation */
};
status_t CAAM_ECDSA_Verify(CAAM_Type *base,
                           caam_handle_t *handle,
                           uint8_t curve_sel,
                           uint8_t *msg,
                           uint32_t msg_len,
                           uint8_t *priv_key,
                           uint8_t *sign_c,
                            uint8_t *sign_d,
                            uint8_t *temp_buffer)
{
    status_t status = kStatus_InvalidArgument;
    uint32_t descriptor[ARRAY_SIZE(templateEcdsaVerify)];

    /* create job descriptor */
    BUILD_ASSURE(sizeof(templateEcdsaVerify) <= sizeof(descriptor),
 templateEcdsaVerify_descriptor);
    uint32_t descriptorSize = ARRAY_SIZE(templateEcdsaVerify);

    (void)caam_memcpy(descriptor, templateEcdsaVerify,
 sizeof(templateEcdsaVerify));
    /* Add descriptor HEADER len and index */
    descriptor[0] |= descriptorSize | (((descriptorSize - 1) & 0x3F) << 16);
    descriptor[1] |= (curve_sel << 7) | CAAM_ECDSA_PD;
    descriptor[2] = (uint32_t)ADD_OFFSET((uint32_t)priv_key);
    descriptor[3] = (uint32_t)ADD_OFFSET((uint32_t)msg);
    descriptor[4] = (uint32_t)ADD_OFFSET((uint32_t)sign_c);
    descriptor[5] = (uint32_t)ADD_OFFSET((uint32_t)sign_d);
    descriptor[6] = (uint32_t)ADD_OFFSET((uint32_t)temp_buffer);
    descriptor[7] = msg_len;
    /* Operation:sha256, Encrypted ECC private key with CCM mode,  */
    descriptor[8] |= ECDSA_PROTINFO_ECC | ECDSA_PROTINFO_MES_REP |
 ECDSA_PROTINFO_HASH_SHA256;

    /* schedule the job and block wait for result */
    do
    {
        status = caam_in_job_ring_add(base, handle->jobRing, &descriptor[0]);
    } while (status != kStatus_Success);

    status = CAAM_Wait(base, handle, &descriptor[0], kCAAM_Blocking);
    return status;
}
```

3.4 Example application

This section provides a test function that demonstrates how to use the functions described in the previous sections to generate and use ECC key pairs on the RT1170. Users can invoke this function within the CAAM driver demo using the following call: `ECCKeyTest(CAAM, &caamHandle)`. The test is based on the secp256r1 curve. For other curves, buffer sizes may need to be adjusted accordingly. For guidance on CAAM initialization and job ring interface setup, users can refer to the CAAM driver demo included in the SDK, located at: *<SDK_PATH>\boards\evkbmimxrt1170\driver_examples\caam*.

```
static void ECCKeyTest(CAAM_Type *base, caam_handle_t *handle)
{
    status_t status;
    uint8_t curve_sel = 0x02; //secp256r1
    bool enc_private = true;

    uint8_t priv_key[32 + 16];
    uint8_t pub_key[64];

    memset(priv_key, 0, sizeof(priv_key));
    memset(pub_key, 0, sizeof(pub_key));

    status = CAAM_ECC_Keypairs_Generation(base, handle, curve_sel,
                                                enc_private, priv_key, pub_key);
    if (status == kStatus_Success)
    {
        PRINTF("- ECCKeypairGen success!\r\n\r\n");
    }
    else
    {
        PRINTF("- ECCKeypairGen failed!\r\n\r\n");
        return;
    }

    uint8_t msg[] = "0123456789";
    uint32_t msg_len = sizeof(msg) - 1;
    uint8_t sign_c[32];
    uint8_t sign_d[32];

    memset(sign_c, 0, sizeof(sign_c));
    memset(sign_d, 0, sizeof(sign_d));

    status = CAAM_ECDSA_Sign(base, handle, curve_sel, enc_private, msg, msg_len,
                             priv_key, sign_c, sign_d);
    if (status == kStatus_Success)
    {
        PRINTF("- ECC Sign message success!\r\n\r\n");
    }
    else
    {
        PRINTF("- ECC Sign message failed!\r\n\r\n");
        return;
    }

    uint8_t temp_buffer[32 * 2];

    memset(temp_buffer, 0, sizeof(temp_buffer));
    status = CAAM_ECDSA_Verify(base, handle, curve_sel, msg, msg_len,
                               pub_key, sign_c, sign_d, temp_buffer);
    if (status == kStatus_Success)
    {
        PRINTF("- ECC Verify message success!\r\n\r\n");
    }
    else
    {
        PRINTF("- ECC Verify message failed!\r\n\r\n");
    }
}
```

4 Abbreviations and acronyms

lists the abbreviation and acronyms used in this document.

Table 11. Abbreviations and acronyms

Abbreviation/ acronyms	Description
CAAM	Cryptographic Acceleration and Assurance Module
ECC	Elliptic curve cryptography
ECDSA	Elliptic curve digital signature algorithm

Table 11. Abbreviations and acronyms*...continued*

Abbreviation/ acronyms	Description
JDKEK	Job descriptor key encryption key
JD	Job descriptor
PDB	Protocol data block
TRNG	True random number generator
ICV	Integrity check value
AES	Advanced Encryption Standard
ECB	Electronic Codebook
CCM	Counter with CBC-MAC
CBC	Cipher Block Chaining
MAC	Message Authentication Code

5 Related documentation

Table 12 lists the additional documents and resources that can be referred to if more information is required. Some of the documents listed below may be available only under a non-disclosure agreement (NDA). To request access to these documents, contact your local field applications engineer (FAE) or sales representative.

Table 12. Related documentation

Document	Link/how to access
Security Reference Manual for the i.MX RT1170 Processor (document IMXRT1170SRM)	Contact NXP local field applications engineer (FAE) or sales representative.
i.MX RT1170 Keyblob Encapsulation and Decapsulation (AN13711)	Contact NXP local field applications engineer (FAE) or sales representative.

6 Note about the source code in the document

The example code shown in this document has the following copyright and BSD-3-Clause license:

7 Revision history

Section 7 summarizes revisions to this document.

Table 13. Revision history

Document ID	Release date	Description
AN14753 v.1.0	18 July 2025	Initial public release

Legal information

Contents

Date of release: 18 July 2025
Document identifier: AN14753