AN12812

Using Code-Signing Tool with Hardware Security Module

Rev. 0 — June 2020

Application Note

Contents

1 Preface

This document provides the information necessary for the user to effectively use Code-Signing Tool (CST) with a Hardware Security Module (HSM) backend. It is primarily intended for users who are familiar with CST tool to sign codes for the NXP High Assurance Boot (HAB).

The document is valid for CST versions starting from 3.3.1.

1.1 Intended audience and scope

This guide is intended for software, hardware, and system engineers who are planning to use CST to perform code signing for HAB using keys persisted in an HSM.

1.2 References

- Code-Signing Tool User's Guide, Code Signing Tool package downloadable on http://www.nxp.com. Search for CST_TOOL.
- *Secure Boot on i.MX 50, i.MX 53, i.MX 6 and i.MX 7 Series using HAB* (document AN4581).
- i.MX Secure and Encrypted Boot using HABv4, available in *doc/imx/habv4* of the U-Boot project, on the imx_v2018.03_4.14.98_2.0.0_ga GA release branch.
- Open Secure Socket Layer (OpenSSL), 2020, http://www.openssl.org.
- PKCS#11 wrapper library, 2020. https://github.com/OpenSC/libp11.
- The p11-kit web pages, 2020. http://p11-glue.freedesktop.org/p11-kit.html.
- The PKCS #11 URI Scheme, 2020. https://tools.ietf.org/html/rfc7512.

2 Overview

Refering to **Appendix B, Replacing the CST Backend Implementation** of Code-Signing Tool User's Guide, NXP has architected the Code-Signing Tool in two parts a Front-End and a Back-End. The Front-End contains all the NXP proprietary operations, while the Back-End containing all standard cryptographic operations. Users can write a replacement for the reference backend to, for example, interface with an HSM or Smartcard.

The reference backend uses OpenSSL to perform HAB signature generation and encrypted data generation. OpenSSL in his turn, exposes an Engine API, which makes it possible to plug in alternative implementations for some of the cryptographic operations implemented by OpenSSL. We can take advantage of this to reuse the reference backend with an HSM, for example, by offloading cryptographic operations involved during signature generation to HSM.

The engine should re-write an implementation of RSA private encrypt function, and how public certificates and private keys are loaded from the HSM to the appropriate OpenSSL data structure: X509, RSA and EVP_PKEY. Optionally SHA digest functions can be re-written also to be performed at the HSM level.

2.1 PKCS#11 enabled HSM

PKCS#11 is a standardized interface for cryptographic tokens which exposes an API called Cryptoki. Cryptographic token manufacturers provide shared libraries (`.so` or `.dll`) which implements PKCS#11 standard. Those libraries can be used as bridge between the HSM, Smartcard etc and the CST Back-End. Typically, the engine loads the shared PKCS#11 module and invokes the appropriate functions on it.

Figure 1 illustrates the architecture of Code-Signing Tool with a Backend replacement to interface with a PKCS#11 enabled HSM.

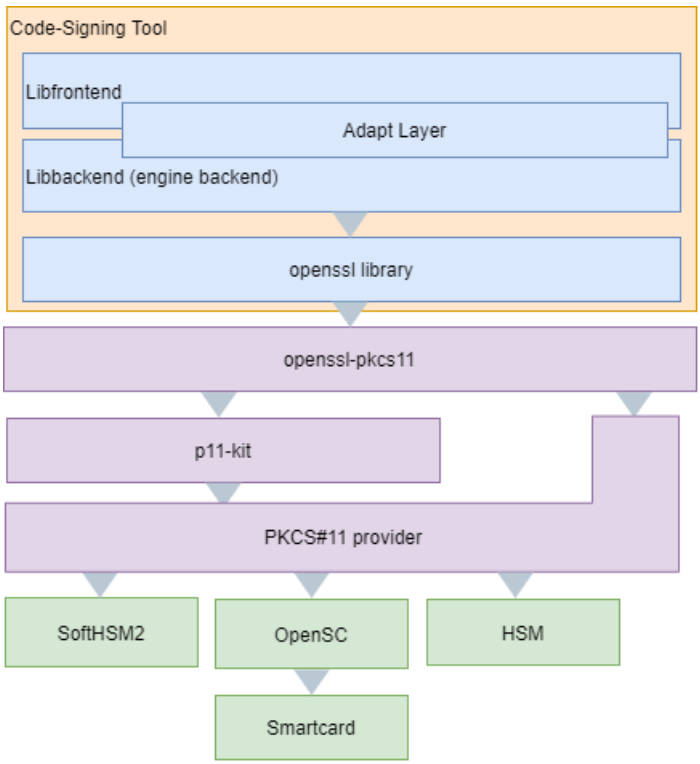

Figure 1. Engine backed CST architecture

To link the CST Frontend to the Backend, a new adaptation layer has been developed to implement two APIs used by the Frontend.

- `gen_sig_data`: The CST Front End uses this API to generate HAB signatures. In this function RSA and ECDSA signing are offloaded to the OpenSSL PKCS#11 engine to be executed in the HSM, Smartcard etc.
- `get_der_encoded_certificate_data`: The CST frontend uses this API to read certificate in DER format. Given a PKCS#11 URI, this function loads a certificate content to a X509 data structure.

Libp11 (openssl-pkcs11) is used as PKCS#11 engine for OpenSSL. Libp11 is also a PKCS#11 library which implements all required functions to manage session and tokens, load public certificates, private keys, sign and hash. Slight modification has been be done at the reference backend to initialize the engine, perform control command and configuration. Libp11 provides a gateway between vendor's PKCS#11 modules (PKCS#11 provider) and the OpenSSL engine API.

The PKCS#11 engine could access directly a PKCS#11 provider. An alternative could be using the p11-kit proxy module which provides access to any configured PKCS #11 module in the system See http://p11-glue.freedesktop.org/p11-kit.html for more information.

2.2 Client/Server based HSM

The engine-backend presented in this work, is mainly written for PKCS#11 enabled HSMs. In case of Client/Server based HSM the built-in engine can be adapted to implement a client to consume a server exposed API for example. The following requirements should be fulfilled:

1. Locate keys/certificates on HSM by an identifier, name or label. This identifier will be kept across cryptographic operation.
2. Load partial attributes of private key. In case of RSA should be able to get the value of the modulus and the exponent. It is mandatory to populate RSA structure with those parameters for private keys. OpenSSL uses that for consistency check between certificate and its corresponding private key. In case your HSM is not able to provide partial private keys parameters, you should patch OpenSSL to ignore `X509_check_private_key` function.
3. Write an implementation for RSA init/finish methods and `rsa_priv_encrypt` method.
4. Optionally implement SHA diget methods if you want to perform digesting at HSM level. Methods are: `digest_init`, `digest_update`, `digest_finish`, `digest_copy`, and `digest_cleanup`.
5. Engine initialization and destruction.
6. In case of stateful service, add a session manager to your implementation.

3 Hands-on

The following instructions will get you a copy of the engine backend implementation and pre-built frontend to build CST up and running on your local machine for development and testing purposes.

The steps described in this section have been tested on an `x86_64` machine running Ubuntu 16.04 and as root user.

3.1 Dependencies

The CST engine backend depends on:

OpenSSL library

```
$ apt-get install openssl libssl-dev
$ openssl version
OpenSSL 1.0.2g  1 Mar 2016
```

Make is required for building the software.

3.2 Getting sources

The engine backend source code for CST can be found in the official CST package.

3.3 Compiling sources

To compile backend and CST from source:

1. Verify that the compiler is working by doing:

   ```
   $ gcc -v
   ```

2. Compile the backend source code using the following commands:

   From CST package:

   ```
   $ tar xzf cst-release.tgz
   $ cd release/code/back_end-engine/src/
   $ make
   ```

   The result is **`libbackend.a`** static library and **`cst`** binary present in the current working folder.

3. Install CST

Optionally, to install the cst binary into your PATH:

```
$ make install
```

3.4 Usage

This section provides instructions and prerequisites to deploy Code-Signing Tool with SoftHSM2. SoftHSM2 will be used as an HSM emulator.

NOTE

All actions are executed with root privileges on a Ubuntu 16.0.4 machine.

3.4.1 Installing and configuring HSM

1. Install SoftHSM2

   ```
   $ apt-get install softhsm2 libsofthsm2-dev
   ```

   For SoftHSM we need to create the following folder, otherwise it fails initializing the token later.

   ```
   $ mkdir -p /var/lib/softhsm/tokens/
   ```

   SoftHSM2 uses file-based storage by default and this folder holds the tokens.

2. Install PKCS#11 utilities

   To interact with HSM you can use OpenSC utilities.

   ```
   $ apt-get install opensc
   ```

3. Install OpenSSL PKCS#11 engine.

   To generate the HAB PKI tree, OpenSSL with PKCS#11 engine will be used.

   ```
   $ apt-get install libengine-pkcs11-openssl
   ```

   The version used in this example is 0.4.7-3. If you are using an earlier version, it is highly recommended to upgrade.

   ```
   dpkg -s libengine-pkcs11-openssl | grep '^Version'
   Version: 0.4.7-3
   ```

4. Setup the vendor's PKCS#11 library

   You need to locate your HSM vendor's PKCS#11 interface implementation in order to use it later with pkcs11-tools and CST later.

3.4.1.1 Using in systems with p11-kit

The p11-kit-proxy module loads and provides other PKCS#11 libraries such as `softhsm2`. p11-kit proxy only proxies configured PKCS#11 providers. See http://p11-glue.freedesktop.org/p11-kit.html for more information.

1. Install p11-kit

   ```
   # apt-get install p11-kit
   ```

In systems with p11-kit-proxy engine_pkcs11 has access to all the configured PKCS #11 modules and requires no further configuration.

```
$ ls /usr/share/p11-kit/modules/
opensc-pkcs11.module  p11-kit-trust.module  softhsm2.module
```

2. Change the priority of PKCS#11 to be set to SoftHSM.

   For more details please refer to https://manpages.debian.org/testing/p11-kit/pkcs11.conf.5.en.html

```
# echo "priority: 10" >>  /usr/share/p11-kit/modules/softhsm2.module
```

   Assume that there is no entry for you HSM, you can create a new one by defining the path to the HSM vendor's PKCS#11 library and the priority.

   The following is an entry for `softhsm2.` module in case it is not automatically created during installation.

```
cat <<EOF > /usr/share/p11-kit/modules/softhsm2.module
priority: 10
module : /usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
EOF
```

   SoftHSM should appear first in the list.

```
# p11-kit list-modules
softhsm2: /usr/lib/softhsm/libsofthsm2.so
    library-description: Implementation of PKCS11
    library-manufacturer: SoftHSM
    library-version: 2.2
    token: CST-HSM-DEMO
        manufacturer: SoftHSM project
        model: SoftHSM v2
        serial-number: 08e3dc40e2c3ac6a
        hardware-version: 2.2
        firmware-version: 2.2
--truncated
```

3. Set the p11-kit proxy as PKCS11 library.

```
$ find /usr -name p11-kit-proxy.so
/usr/lib/x86_64-linux-gnu/p11-kit-proxy.so
```

   In this case, the PKCS11# library is at */usr/lib/x86_64-linux-gnu/p11-kit-proxy.so*.

```
$ export PKCS11_MODULE=/usr/lib/x86_64-linux-gnu/p11-kit-proxy.so
```

3.4.1.2 Using in systems without p11-kit

1. Locate your HSM vendor's PKCS#11 interface implementation.

   For example, when using SoftHSM2:

```
# find / -name libsofthsm2.so
/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
```

```
export PKCS11_MODULE=/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
```

2. Define Token login parameters:

```
$ export SO_PIN=7635005489180126
$ export USR_PIN=12345678
```

3. Initialize a token:

```
$ pkcs11-tool --module $PKCS11_MODULE --init-token --init-pin --so-pin=$SO_PIN --new-pin=
$USR_PIN --label="CST-HSM-DEMO" --pin=$USR_PIN --login

Using slot 0 with a present token (0x0)
Token successfully initialized
User PIN successfully initialized
```

The token label, **CST-HSM-DEMO**, will be used later with CST to locate certificates and keys needed to sign images.

4. To make sure that the pkcs11-tool interacts correctly with the HSM and the HSM is properly configured you can invoke a simple command to display general information about the token

```
$ pkcs11-tool --module $PKCS11_MODULE --show-info
```

5. List the slots:

Verify that the token is initialized.

```
pkcs11-tool --module $PKCS11_MODULE --list-slots
Available slots:
Slot 0 (0x62c3ac6a): SoftHSM slot ID 0x62c3ac6a
  token label        : CST-HSM-DEMO
  token manufacturer : SoftHSM project
  token model        : SoftHSM v2
  token flags        : login required, rng, token initialized, PIN initialized, other flags=0x20
  hardware version   : 2.2
  firmware version   : 2.2
  serial num         : 08e3dc40e2c3ac6a
  pin min/max        : 4/255
Slot 1 (0x1): SoftHSM slot ID 0x1
  token state:   uninitialized
…
```

6. Configure OpenSSL PKCS#11 engine.

Verify that the engine is properly operating by running:

```
# openssl engine -t pkcs11
(pkcs11) pkcs11 engine
     [ available ]
```

If you get an error invoking the command, usually in old OpenSSL versions, this is because *libengine-pkcs11-openssl* writes its engine files to */usr/lib/ssl/engines* rather than the location where current OpenSSL looks for engines.

Manually loading the PKCS#11 engine by defining the correct path to libraries should succeed.

```
# openssl engine dynamic -pre SO_PATH:/usr/lib/engines/engine_pkcs11.so -pre ID:pkcs11 -pre
LIST_ADD:1 -pre LOAD -pre MODULE_PATH:$PKCS11_MODULE
```

To prevent loading the PKCS#11 engine manually each time we want to execute a command within the token, add the following entries to */etc/ssl/openssl.cnf*.

This line must be placed at the top, before any sections are defined:

```
openssl_conf = openssl_def
```

This should be added to the bottom of the file:

```
[openssl_def]
engines = engine_section

[engine_section]
pkcs11 = pkcs11_section

[pkcs11_section]
engine_id=pkcs11
dynamic_path = /usr/lib/engines/engine_pkcs11.so
MODULE_PATH = <PATH TO PKCS#11 MODULE>
init = 0
```

Set the MODULE_PATH parameter to the path of PKCS#11 library depending on your choice to use or not p11-kit.

System without p11-kit:

```
MODULE_PATH = /usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
```

System with p11-kit:

```
MODULE_PATH = /usr/lib/x86_64-linux-gnu/p11-kit-proxy.so
```

Try again:

```
# openssl engine -t pkcs11
(pkcs11) pkcs11 engine
     [ available ]
```

3.4.2 Generating PKI tree

3.4.2.1 HABv4

The PKI tree for this example is simple and it will be based on one SRK key. The SRK is chosen to be a CA key. The key length is 2048 bits.

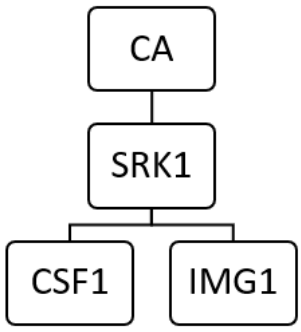

Figure 2. HABv4 PKI tree example

The first SRK as the root of trust. The CSF 1 and IMG 1 keys are used to sign the CSF data and the image respectively. For more details, refer to CST documentation and Secure Boot Application Notes.

1. Change your directory to certs.

```
$ pushd ../../../crts/
```

2. The following is required otherwise OpenSSL complains.

```
$ touch index.txt
$ echo "unique_subject = no" > index.txt.attr
```

3. Create serial file.

   serial - 8-digit OpenSSL uses the contents of this file for the certificate serial numbers.

```
$ echo "12345678" > serial
```

   In the following steps we make use of custom OpenSSL configuration file.

   When using an old OpenSSL version, the engine configuration for */etc/ssl/openssl.cnf* should be replicated for the ***ca/openssl.cnf*** file.

   Optionally you decide to specify the PIN code in the configuration file or type it each time you execute a command within the HSM.

4. Generate CA key and certificate.

```
export CA_KEY="CA1_sha256_2048_ca"
```

   Create CA RSA key-pair.

```
$ pkcs11-tool --module $PKCS11_MODULE -l --pin $USR_PIN --keypairgen --key-type rsa:2048 --label
$CA_KEY --id 1000
Using slot 0 with a present token (0x10)
Key pair generated:
Private Key Object; RSA
  label:      CA1_sha256_2048_ca
  Usage:      decrypt, sign, unwrap
Public Key Object; RSA 2048 bits
  label:      CA1_sha256_2048_ca
  Usage:      encrypt, verify, wrap
```

   Generate a self-signed root certificate:

```
openssl req -engine pkcs11 -new -batch -subj "/CN=${CA_KEY}/" -key "label_${CA_KEY}" -keyform
engine -out ${CA_KEY}.pem -text -x509 -days 3640 -config ../ca/openssl.cnf
```

   Enter User PIN when prompts (if not specified in configuration file).

   The flag, `label_`, identifies the key label you're using. If you want to refer to a different token or key id, you can change these.

5. Generate SRK key and certificate.

```
export SRK1_KEY="SRK1_sha256_2048_ca"
```

```
pkcs11-tool --module $PKCS11_MODULE -l --pin $USR_PIN --keypairgen --key-type rsa:2048 --label
$SRK1_KEY --id 1001
```

Generate SRK certificate signing request.

```
openssl req -engine pkcs11 -new -batch -subj "/CN=${SRK1_KEY}/" -keyform engine -key "label_$
{SRK1_KEY}" -out temp_srk_req.pem
```

Generate SRK certificate (this is a CA cert).

```
openssl ca -engine pkcs11 -batch \
                  -md sha256 -outdir ./ \
                  -in ./temp_srk_req.pem \
                  -cert "${CA_KEY}.pem" \
                  -keyform engine -keyfile "label_${CA_KEY}" \
                  -extfile ../ca/v3_ca.cnf \
                  -out "${SRK1_KEY}.pem"  \
                  -notext \
                  -days 3640  \
                  -config ../ca/openssl.cnf
```

Convert it to the DER format.

```
openssl x509 -outform der -in ${SRK1_KEY}.pem -out ${SRK1_KEY}.der
```

Store it back into your HSM.

```
pkcs11-tool --module $PKCS11_MODULE -l  --write-object ${SRK1_KEY}.der  --type cert --label
$SRK1_KEY --id 1001 --pin $USR_PIN
```

Clean.

```
rm temp_srk_req.pem
```

6. Generate CSF key and certificate.

```
export CSF1_KEY="CSF1_1_sha256_2048_usr"
```

```
pkcs11-tool --module $PKCS11_MODULE -l --pin $USR_PIN --keypairgen --key-type rsa:2048  --label
$CSF1_KEY --id 1002
```

Generate CSF certificate signing request.

```
openssl req -engine pkcs11 -new -batch \
                -subj "/CN=${CSF1_KEY}/" \
-keyform engine -key "label_${CSF1_KEY}" \
                -out ./temp_csf_req.pem
```

Generate CSF certificate (this is a user cert).

```
openssl ca -engine pkcs11 -batch -md sha256 -outdir ./ \
               -in ./temp_csf_req.pem \
               -cert "${SRK1_KEY}.pem" \
             -keyform engine -keyfile "label_${SRK1_KEY}"    \
               -extfile ../ca/v3_usr.cnf \
               -out "${CSF1_KEY}.pem" \
               -notext \
               -days 3640  \
               -config ../ca/openssl.cnf
```

Convert it to the DER format.

```
openssl x509 -outform der -in ${CSF1_KEY}.pem -out ${CSF1_KEY}.der
```

And store it back into your HSM.

```
pkcs11-tool --module $PKCS11_MODULE -l  --write-object ${CSF1_KEY}.der  --type cert --label
$CSF1_KEY --id 1002 --pin $USR_PIN
```

Clean.

```
rm temp_csf_req.pem
```

7. Generate IMG key and certificate.

```
export IMG1_KEY="IMG1_1_sha256_2048_usr"
```

```
pkcs11-tool --module $PKCS11_MODULE -l --pin $USR_PIN --keypairgen --key-type rsa:2048 --label
$IMG1_KEY --id 1003
```

Generate IMG certificate signing request.

```
openssl req -engine pkcs11 -new -batch \
                -subj "/CN=${IMG1_KEY}/" \
-keyform engine -key "label_${IMG1_KEY}"  \
                -out ./temp_img_req.pem
```

Generate IMG certificate (this is a user cert).

```
openssl ca -engine pkcs11 -batch -md sha256 -outdir ./ \
               -in ./temp_img_req.pem \
               -cert "${SRK1_KEY}.pem" \
            -keyform engine -keyfile "label_${SRK1_KEY}"     \
               -extfile ../ca/v3_usr.cnf \
               -out "${IMG1_KEY}.pem" \
               -notext \
               -days 3640 \
               -config ../ca/openssl.cnf
```

Convert it to the DER format.

```
openssl x509 -outform der -in ${IMG1_KEY}.pem -out ${IMG1_KEY}.der
```

Store it back into your HSM.

```
pkcs11-tool --module $PKCS11_MODULE -l --write-object ${IMG1_KEY}.der  --type cert --label
$IMG1_KEY --id 1003 --pin $USR_PIN
```

Clean.

```
rm temp_img_req.pem
```

8. Verify that all cryptographic materials were written correctly to token by running.

List all objects.

```
pkcs11-tool --module $PKCS11_MODULE -l --pin $USR_PIN --list-objects
```

3.4.3 Generating SRK table

Create fuse table and binary (SRK table) to be flashed.

```
$ ../linux64/bin/srktool -h 4 -t SRK_1_table.bin -e SRK_1_fuse.bin -d sha256 -c ./
SRK1_sha256_2048_ca.pem -f 1
```

- `SRK_1_table.bin` - SRK table contents with HAB data
- `SRK_1_fuse.bin` - Contains SHA256 result to be burned to fuse

3.4.4 HAB command sequence file

Create *imx-boot.csf* file.

```
[Header]
   Version = 4.3
   Hash Algorithm = sha256
   Engine = CAAM
   Engine Configuration = 0
   Certificate Format = X509
   Signature Format = CMS

[Install SRK]
   File = "./SRK_1_table.bin"
   Source index = 0

[Install CSFK]
   File = "pkcs11:token=CST-HSM-DEMO;object=CSF1_1_sha256_2048_usr;type=cert;pin-value=12345678"

[Authenticate CSF]

[Unlock]
   Engine = CAAM
   Features = MID

[Unlock]
    Engine = CAAM
    Features = MFG

[Install Key]
   Verification index = 0
   Target index = 2
   File = "pkcs11:token=CST-HSM-DEMO;object=IMG1_1_sha256_2048_usr;type=cert;pin-value=12345678"

[Authenticate Data]
   Verification index = 2
   Blocks = 0x7e0fc0 0x0 0x2bc00 "flash.bin"
```

3.4.5 Sign

Generate the CSF binary signature.

```
$ cst -i imx-boot.csf -o imx-boot.csf.bin
CSF Processed successfully and signed data available in imx-boot.csf.bin
```

3.5 Debugging

To trace the PKCS#11 flow between CST and the HSM, pkcs11-spy tool could be used.

For security reason, you should only use the tool for debugging, and preferable only with test keys as the tool logs all calls including PIN, signatures, etc.

Set the log output to a file:

```
export PKCS11SPY_OUTPUT=cst.log
```

Define the path to the real PKCS#11 library depending on if the system is with or without p11-kit.

Assume we are using SoftHSM2 on system without p11-kit.

```
export PKCS11SPY=/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so
```

Locate pkcs11-spy and use it instead of the real PKCS#11 provider:

```
find / -name pkcs11-spy.so
/usr/lib/x86_64-linux-gnu/pkcs11-spy.so
```

Edit *openssl.cnf* to use PKCS11 SPY as:

```
MODULE_PATH=/usr/lib/x86_64-linux-gnu/pkcs11-spy.so
```

Invoke `cst` again. A *cst.log* file should be available with all the invoked calls with the associated arguments dump.

```
*************** OpenSC PKCS#11 spy *****************
Loaded: "/usr/lib/x86_64-linux-gnu/softhsm/libsofthsm2.so"

0: C_GetFunctionList
2020-04-23 15:38:25.647
Returned:  0 CKR_OK

1: C_Initialize
2020-04-23 15:38:25.647
[in] pInitArgs = (nil)
Returned:  0 CKR_OK

….
469: C_Sign
2020-04-23 15:38:25.701
[in] hSession = 0x11
…
Returned:  0 CKR_OK
```

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

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

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

Date of release: June 2020

Document identifier: AN12812

arm