Implementing Hardware Cryptographic Support in wolfCrypt

Document Version: 0.1

Table of Contents:

This guide describes how to add support for cryptographic offloading to hardware. It will go over the design of wolfCrypt and the various ways users can add hardware crypto offloading support for new and custom platforms.

For the purpose of this demonstration, this guide will be discussing AES, however the same procedures can also be applied to other algorithms such as SHA256, ECC and RSA depending on what the hardware platform supports.

Option 1: Implementing Cryptographic Callbacks

This option allows runtime replacement of wolfCrypt's software cryptography functionality with a custom or 3rd party implementation. The goal is to make adding hardware cryptographic support to wolfSSL and wolfCrypt easier.

- Pro: Nothing has to be contributed back to wolfSSL to maintain custom changes through release cycles, all the benefits of option 3 (below) but nothing has to be contributed back.

- Con: A bit more complex on initial setup

This feature is enabled using "**--enable-cryptocb**" or "#define **WOLF_CRYPTO_CB**".

Currently supported crypto callbacks include:

- RNG and RNG Seed
- ECC (key gen, sign/verify and shared secret)
- RSA (key gen, sign/verify, encrypt/decrypt)
- AES CBC/GCM
- DES3
- SHA1 and SHA256
- HMAC

To register a cryptographic callback function use the "**wc_CryptoCb_RegisterDevice**" API.

This takes a unique device ID (devId), callback function and optional user context.

```
typedef int (*CryptoDevCallbackFunc)(int devId, wc_CryptoInfo* info,
                                     void* ctx);
WOLFSSL_API int wc_CryptoCb_RegisterDevice(
    int devId,
    CryptoDevCallbackFunc cb,
```

```
    void* ctx);
```

To enable use of the crypto callbacks the “devId” arguments must be supplied on initialization.

For TLS use:

- wolfSSL_CTX_SetDevId(ctx, devId);
- wolfSSL_SetDevId(ssl, devId);

For wolfCrypt API’s use the initialization functions that accept “devId” such as:

- wc_InitRsaKey_ex
- Wc_ecc_init_ex
- wc_AesInit
- wc_InitSha256_ex
- wc_InitSha_ex
- wc_HmacInit

Examples:

- STSAFE-A100 ECC Crypto Callbacks: https://github.com/wolfSSL/wolfssl/blob/master/wolfcrypt/src/port/st/stsafe.c#L330
- TPM 2.0 wolfTPM Crypto Callbacks: https://github.com/wolfSSL/wolfTPM/blob/master/src/tpm2_wrap.c#L2937
- Generic wolfCrypt tests: https://github.com/wolfSSL/wolfssl/blob/master/wolfcrypt/test/test.c#L24304

For additional details see this blog post: https://www.wolfssl.com/wolfcrypt-support-cryptographic-callbacks/

NOTE: This is the preferred solution by wolfSSL and we encourage those adding custom hardware to use this option and put the example code into wolfcrypt/src/port as described in option 3.

Option 2: Adding a New Hardware Section to the Target Source

- Pro: If the work will contributed back, wolfSSL can maintain the changes in all future releases.

- Con: Work has to be re-implemented in each new release unless it is contributed back to wolfSSL. Results in large code additions to aes.c (or the respective source file).

For this option the aes.c source file is used as an example along with AES-CBC mode, since CBC is simpler than GCM. In the wolfssl-x.x.x/wolfcrypt/src/aes.c source file the below pre-processor logic and structure will be found:

```
#if defined(STM32_CRYPTO)
    <code>
    #ifdef WOLFSSL_STM32_CUBEMX
        <code>
    #else /* STD_PERI_LIB */
        <code>
    #endif /* WOLFSSL_STM32_CUBEMX */
#elif defined(HAVE_COLDFIRE_SEC)
    <code>
#elif defined(FREESCALE_LTC)
    <code>
#elif defined(FREESCALE_MMCAU)
    <code>
#elif defined(WOLFSSL_PIC32MZ_CRYPT)
    <code>
#elif defined(WOLFSSL_ESP32WROOM32_CRYPT) && \
      !defined(NO_WOLFSSL_ESP32WROOM32_CRYPT_AES)
    <code>
#elif defined(WOLFSSL_CRYPTOCELL) && defined(WOLFSSL_CRYPTOCELL_AES)
    <code>
#elif defined(WOLFSSL_IMX6_CAAM) && !defined(NO_IMX6_CAAM_AES)
    <code>
#elif defined(WOLFSSL_AFALG)
    <code>
#elif defined(WOLFSSL_DEVCRYPTO_CBC)
    <code>
#else
    /* Software AES - CBC Encrypt */
    <code>
#endif /* AES-CBC block */
#endif /* HAVE_AES_CBC */
```

Each of the above sections contains an implementation of the API's:

```
wc_AesCbcEncrypt
wc_AesCbcDecrypt
wc_AesSetKey
wc_AesSetIV
```

For a custom hardware implementation port the parameters are passed off to the hardware. For example, in **wc_AesCbcEncrypt** the incoming parameters are:

- wolfCrypt AES structure
- byte array to store the encrypted result
- byte array of plain-text input
- size to denote the length of the plaintext

The Aes structure holds the key and IV already. These are used to prepare the hardware for an encrypt operation based on the key size then perform the encrypt/decrypt operation by calling to the hardware. The encrypted result is stored in the “out” byte array and returned back to wolfCrypt once the Aes structure is updated with the current blocks IV.

Option 3: Adding Hardware Specific Source File(s)

- Pro: Changes can be contributed back without sharing the sources expected to be in wolfcrypt/src/port/<hrdwr>_aes.c. Custom changes can then be maintained in a <hrdwr>_aes.c file and dropped into new releases of wolfSSL if changes need to be kept internal, private, or can’t be open-sourced.

- Con: A bit more complex but results in the least technical debt if at least the changes in aes.c (or other relevant wolfcrypt sources) are contributed back to wolfSSL.

Some devices have a more complex setup process. Due to complexity, customers and wolfSSL engineers sometimes choose to add separate source file(s) to keep aes.c from becoming overly complex. For example, the wolfssl-x.x.x/wolfCrypt/src/port directory contains folders for many devices - each folder might contain support for multiple device families and multiple algorithms.

```
/wolfcrypt/src/port/Espressif
/wolfcrypt/src/port/af_alg
/wolfcrypt/src/port/arm
/wolfcrypt/src/port/atmel
/wolfcrypt/src/port/caam
/wolfcrypt/src/port/cavium
/wolfcrypt/src/port/devcrypto
/wolfcrypt/src/port/intel
/wolfcrypt/src/port/mynewt
```

```
/wolfcrypt/src/port/nrf51.c
/wolfcrypt/src/port/nxp
/wolfcrypt/src/port/pic32
/wolfcrypt/src/port/st
/wolfcrypt/src/port/ti
/wolfcrypt/src/port/xilinx
```

If you have one or more devices OR if you have a large code body to add to get hardware offload working OR if you just don't want your hardware calls to be open source, this is a good stand-alone way to do it. Just add a simple implementation in aes.c (or other wolfcrypt sources) and implement the hardware calls in a separate file.

```
#elif defined(my_hardware)

int wc_AesCbcEncrypt(<params >) {
    /* my_hardware_AesSetKey located in:
     * wolfcrypt/src/port/myhardware/<hrdwr>_aes.c
     */
    return my_hardware_AesSetKey(<params>);
}

#elif …
```

(You'll need to add a header and include it via wolfssl/wolfcrypt/settings.h for all the algorithms you are supporting as well for this option).

For an example of Option 3 see the Espressif WROOM (ESP32WROOM32_CRYPT) define in wolfcrypt/src/aes.c and wolfcrypt/src/port/Espressif/esp32_aes.c.

Option 4: Not Compiling wolfcrypt/src/aes.c and Adding a Custom Separate File

- Pro: Your changes are your own.

- Con: Work needs to be re-implemented in each new release, very difficult to maintain as changes wolfSSL makes in aes.c and that effect aes.c between releases will need resolved in the custom version of aes.c on each new release.

This is an option for users that do not want to contribute their solutions back to wolfSSL for long-term maintenance but need to be able to drop it into each new release that

comes out. For this option users can start by completely removing wolfcrypt/src/aes.c from the build by renaming it to wolfcrypt/src/aes.c.original (or something to your liking) then adding an empty file wolfcrypt/src/aes.c. Add custom versions of the wc_API's calling out to hardware and then pull back in the parts of aes.c.original that do not need ported.

If you have any questions or concerns please contact support@wolfssl.com or reach our support staff through the Zendesk portal at https://wolfssl.zendesk.com