AN1443: SiWx917 Encrypted Execute in Place (XiP)

This application note describes the Encrypted Execute in Place (XiP) functionality on Silicon Labs SiWx917 devices. It provides an overview of how Encrypted XiP is implemented and details for both enabling and configuring this feature.

For more information on prerequisites for provisioning the SiW917 for Encrypted XiP, please refer to UG162 - Simplicity Commander Reference Guide.

KEY POINTS

- Architectural Overview of Encrypted XiP
- Operating modes
- Configuration guides
- Examples enabling Encrypted XiP

Table of Contents

1. Security Features

SiWx917 offers best-in-class features to help secure your low-cost wireless product. Some of these features are interrelated and have overlapping configuration steps. Silicon Labs offers documentation on these features to aid developers integrating SiWx917's security features into their product. The following table summarizes documentation available for these features:

Table 1.1. SiWx917 Security Features Documentation

Document	Summary
AN1431: SiWx917 SoC Firmware Update Application Note	Describes how to perform SoC firmware updates
AN1416: SiWG917 SoC Memory Map Application Note	Describes the SiWG917 SoC Memory Map
AN1439: SiWx917 Hardware Debugging Guidelines	Guidelines for debugging hardware relate issues with SiWx917
AN1428: SiWx917 Debug Lock	How to lock and unlock SiWx917 debug access ports
AN1442: SiWx917 SoC Secure Boot with Anti-Rollback Protection	Describes the secure boot and anti-rollback protection processes on SiWx917
AN1443: Si917 Encrypted Execute in Place	Describes the encrypted execute in place (XiP) capabilities of the SiWx917
UG162: Simplicity Commander Reference Guide	Describes commands available in Simplicity Commander for activating PUF and provisioning the intrinsic device keys for XiP

1.1 Key Reference

Encrypted XiP involves device-internal intrinsic keys and the use of public keys for authenticating firmware before allowing it to run. The following table summarizes the cryptographic keys used for secure boot and their intended purpose:

Table 1.2. List of Public Keys Related to Encrypted XiP

Key Type	Description	Key Type	Key size (bits)	Storage	Lifetime
Master key	Used for decrypting and authenticating keys used by the NWP core	Symmetric, AES	256	Intrinsic(2)	Permanent
Unwrap key	Used for decrypting and authenticating keys used by the M4 core	Symmetric, AES	256	Intrinsic(2)	Permanent
TA OTA key(1)	Used to Encrypt/decrypt TA firmware OTA updates, generate CMAC MIC	Symmetric, AES	256	Flash	Permanent
TA OTA key(1)	Encrypt/decrypt M4 firmware OTA updates, gener-ate CMAC MIC	Symmetric, AES	256	Flash	Permanent
TA FW key1, TA FW key2	Encrypt/decrypt NWP firmware during XiP	Symmetric, AES	128/256	Updatable	Flash
M4 FW key1, M4 FW key2	Encrypt/decrypt NWP firmware during XiP	Symmetric, AES	128/256	Updatable	Flash

Note:

1. These keys are wrapped for tamper resistance.
2. Intrinsic keys are generated at runtime using the PUF and a 52-byte key code, stored in Flash.

1.2 eFuse Reference

Encrypted XiP is configured in the SiWx917 by modifying flags, known as MBR flags. The following table lists the settings required to enable Encrypted XiP:

Table 1.3. Master Boot Record (MBR) flags settings for encrypted XiP

eFuse Name	Description
m4_encrypt_firmware	0: Disables encrypted XiP firmware on the m4 core. 1: Enables encrypted XiP firmware on the m4 core.
m4_fw_encryption_mode	1: Configures encrypted XiP in AES-CTR mode on the M4 core. 2: Configures encrypted XiP inAES-XTS mode on the M4 core.
ta_fw_encryption_mode	Enables and sets encryption mode on the NWP core. 0: Disables encrypted XiP on the NWP core. 1: Enables encrypted XiP in AES-CTR mode on the NWP core. 2: Enables encrypted XiP in AES-XTS mode on the NWP core.
m4_secure_boot_enable	1: Enable M4 secure boot if encrypted XiP is required on M4.
ta_secure_boot_enable	1: Enable NWP secure boot if encrypted XiP is required on NWP

1.3 Minimum Wireless Pro Kit Firmware Version

SiWx917 is supported on Wireless Pro Kit (WPK) Mainboards after firmware version 1v5p0b240 and later. When using SiWx917 on Silicon Labs' WPK, ensure your adapter firmware is up to date by consulting this "How to Update" guide on community.silabs.com.

2. Introduction

Execute in Place (XIP) is a technology used in computing to run software directly from long-term storage, such as a ROM or Flash memory, without requiring that the software be copied into RAM before execution. By eliminating RAM usage, an embedded system with XiP support can reduce memory requirements and die cost can improve power consumption in IoT devices. While XiP offers enhanced security, it also has drawbacks in terms of execution speed that must be considered during system design. The encryption of flash memory content requires additional CPU cycles for decryption, leading to a decrease in execution speed by 25 to 43 %.

In SiWx917 devices, Execute in Place (XiP) refers to loading instructions from Flash memory directly via the Quad-Serial Peripheral Interface (Quad SPI). Additionally, SiWx917 has integrated hardware AES accelerators that can decrypt XiP instructions in real-time without the executing program being aware of the decryption process. Quad SPI is used to improve speed of execution, while an on-the-fly hardware AES engine improves firmware confidentiality. SiWx917 supports encrypted and unencrypted XiP on both the NWP and M4 cores in all operating modes except Radio Co-Processor (RCP) mode. Additionally, XiP can be configured independently per-core.

Some AI/ML algorithms or applications which drive displays using Frame buffers require huge RAM. Internal RAM may not be sufficient in such cases, so an external PSRAM of 2, 4, 8, or 16 MB can be added. If these contents are in plain form bad actors can easily tamper with the contents of external RAM. Additionally, programs can also be configured to be executed directly from PSRAM. So, to secure the contents of PSRAM encrypted XIP functionality can be extended to PSRAM.

As illustrated in the diagram below, M4 accesses both FLASH and PSRAM via QSPI. The NWP accesses the contents through a secure AHB bridge. The QSPI controller retrieves contents from external memory, decrypts them, and places them in cache memory.

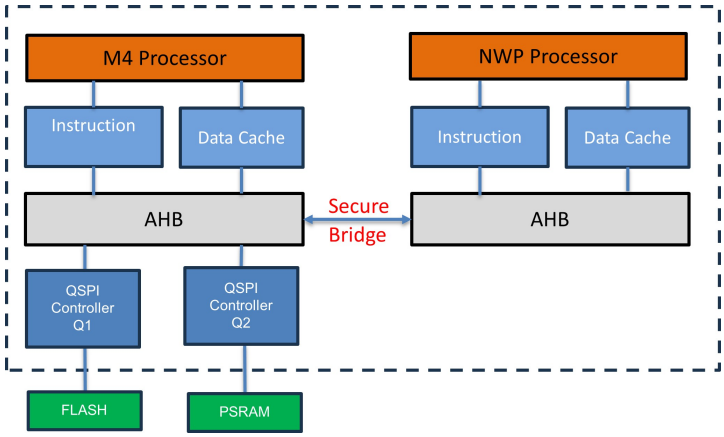

Figure 2.1. M4/NWP FLASH/PSRAM Access

For PSRAM, only AES-CTR mode with 128-bit and 256-bit key sizes is supported. Whereas for FLASH both AES-XTS and AES-CTR modes with 128-bit and 256-bit key sizes are supported.

Note: Entire contents of FLASH/PSRAM contents are not encrypted. Areas that need to be encrypted are configurable by application program.

2.1 QSPI Overview

Encrypted XiP is achieved on the SiWx917 through its integrated Quad SPI (QSPI) and AES engine components. The AES engine is designed to be compliant with NIST FIPS 197 and supports two ciphering modes from NIST SP 800-38: non-chaining CTR mode and AES-XTS.

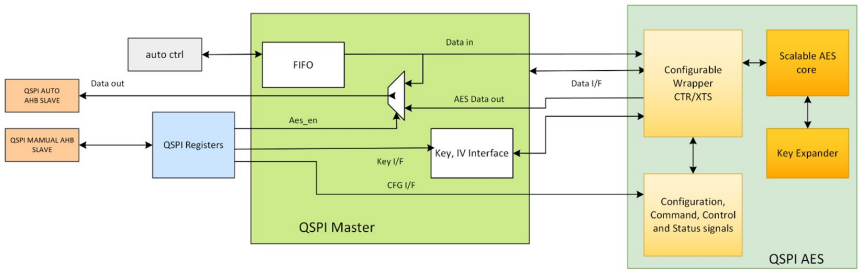

Figure 2.2. QSPI and AES Engine Block Diagram

2.2 PUF Initialization

Both cores of the SiWx917 can be independently configured for XiP with or without encryption. To enable encrypted XiP the SiWx917 PUF token must be initialized, and a corresponding encrypted XiP intrinsic key needs to be provisioned for the given core being placed into encrypted XiP mode.

Without provisioning the PUF, intrinsic XiP encryption keys will not be generated and XiP will not function. For more information configuring the PUF and provisioning device intrinsic keys, see Section 3.2.1 - Activation Code Generation for PUF Block of UG574 – SiWx917 SoC Manufacturing Utility Users Guide – which contains sample mbr json files for configuring encrypted XiP in both operating modes.

3. Operating Modes

3.1 AES-CTR Mode

The Advanced Encryption Standard Counter Mode or AES-CTR is a symmetric-key cipher that uses the AES Electronic Code Book (AES-ECB) algorithm to encrypt a number used once and a counter that is incremented to produce a sequence of output blocks that are combined via exclusive-or (XOR) with the plaintext to produce the ciphertext, and vice versa. Because each block can be constructed independently of each other, AES-CTR mode can be parallelized for applications like XiP where high throughput is required. Additionally, because each block can be independently encrypted or decrypted, AES-CTR mode is well-suited to random access – which is useful for XiP where many branch instructions may require accessing different blocks that are not necessarily close together. Finally, the resultant cipher-text from any plaintext will always be the same width and does not require padding, making it suitable for data of arbitrary length, such as an encrypted firmware executable.

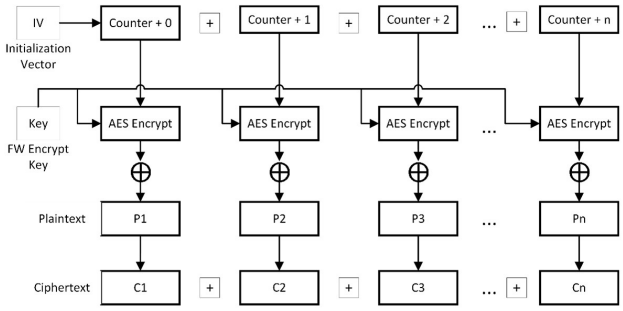

Figure 3.1. AES-CTR Encryption

In AES-CTR encryption mode, the precomputed counter blocks are XORed with a block of plaintext and the results of XORed blocks are concatenated into the final ciphertext.

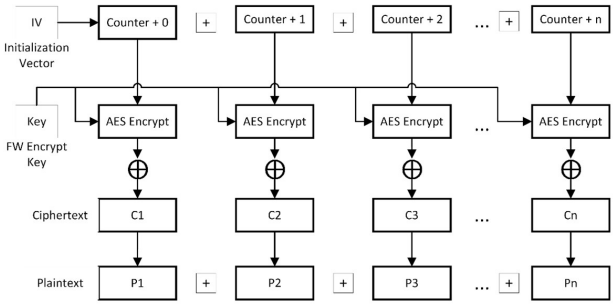

Figure 3.2. AES-CTR Decryption

In decryption mode, the inverse is performed: each counter block is XORed with the ciphertext before being decrypted into corresponding plaintext blocks which are concatenated into the original plaintext.

The sequence of counters must never repeat for any message under a specific key. If a counter repeats, then an attacker with a chosen ciphertext can use that ciphertext to determine contents of other unknown ciphertexts without knowledge of the underlying encryption key.

3.2 AES-CTR Configuration

A section of flash memory, known as Master Boot Record (MBR), contains important configuration information of SiWx917, such as flash configuration, security features, peripheral configuration etc. Upon power-on, the device is configured according to the MBR (Master Boot Record) settings. To enable XIP in various modes minimal MBR configuration JSON content is shown further.

AES-CTR XiP on NWP Only

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data": {
        “ta_encrypt_firmware”: 1
    },
    "key_desc_table_addr": 768
}
```

AES-CTR XiP on M4 Only

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data": {
        "m4_encrypt_firmware": 1,
        "m4_fw_encryption_mode": 1,
    },
    "key_desc_table_addr": 768
}
```

AES-CTR XiP on Both Cores

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data": {
        “ta_encrypt_firmware”: 1,
        "m4_encrypt_firmware": 1,
        "m4_fw_encryption_mode": 1,
    },
    "key_desc_table_addr": 768
}
```

Note: NWP only has one field to both enable and configure encryption, while the M4 has one field to enable encryption and a separate field to set the mode.

3.3 AES-XTS Mode

AES XEX-based tweaked-codebook with ciphertext stealing aka AES-XTS – is a block cipher mode of operation for AES that modifies AES-XEX with a feature called ciphertext stealing to allow plaintexts that are at least one or more blocks of 128 bytes plus a final block of less than 128 bytes to be encrypted and decrypted without use of padding. Ciphertext stealing works by taking a portion of the second-to-last block's output ciphertext, which is then blanked, and using it to fill the last block's plaintext before encryption.

During decryption, the last block is decrypted first, recovering the partial plaintext and 'stolen' ciphertext, which can then be added back into the second-to-last block before it is decrypted. AES-XTS uses two keys, one for encryption of block data and one for encryption of the 'tweak' which is used to modify blocks for added security by acting similarly to a NONCE: ensuring that if the same data is encrypted in two separate blocks, their resulting ciphertexts will still be different due to the tweak. In contrast to NONCE, a tweak is generated using the flash page number and a given block's offset within its page, so that tweak does not require extra flash storage.

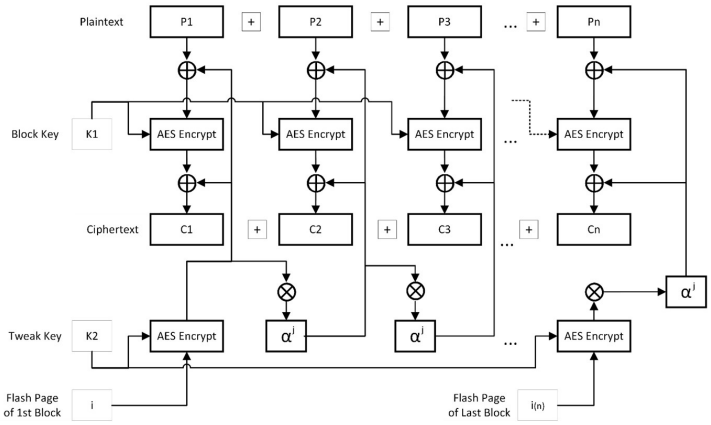

Figure 3.3. AES-XTS Encryption

In AES-XTS encryption mode, the tweak is computed and XORed into the plaintext and the result is encrypted with the Block Key before being XORed with the tweak again.

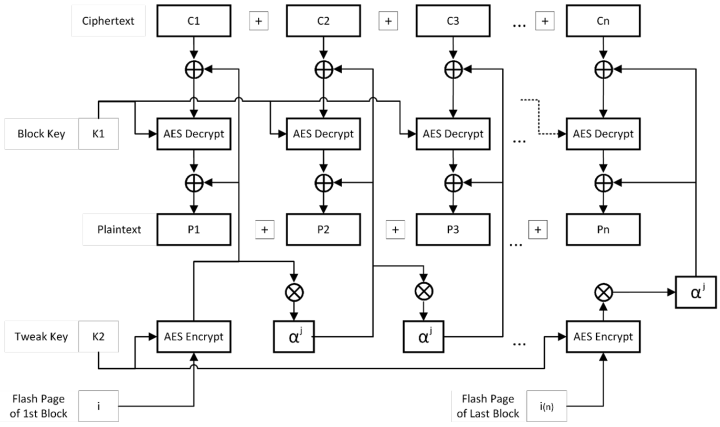

Figure 3.4. AES-XTS Decryption

In AES-XTS decryption mode, the tweak is computed and XORed into the ciphertext and the result is decrypted with the Block Key before being XORed with the tweak again.

It is important to note that while AES-XTS can preserve confidentiality – it does not ensure data integrity. It is not possible to detect that a ciphertext has been modified from its original.

3.4 AES-XTS Configuration

The Master Boot Record of the 917 contains important configuration for the SiWx917. The efuse_data section in this file allows you to configure the functionality of the device. For example, minimal JSON files for enabling XiP in AES-XTS mode using **`commander manufacturing init`** appear further. The modified sections of the default JSON appear in **bold type.**

3.4.1 AES-XTS XiP on TA Only

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data": {
       “ta_encrypt_firmware”: 2,
        “m4_secure_boot_enable”:0,
        "ta_secure_boot_enable": 1
    },
    "key_desc_table_addr": 768
}
```

3.4.2 AES-XTS XiP on M4 Only

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data": {
        "m4_encrypt_firmware": 1,
        "m4_fw_encryption_mode": 2,
        “m4_secure_boot_enable”:1,
        "ta_secure_boot_enable": 0
    },
    "key_desc_table_addr": 768
}
```

3.4.3 AES-XTS XiP on Both Cores

```
{
    "puf_activation_code_addr": 8192,
    "efuse_data":
    {
        “ta_encrypt_firmware”: 2,
        "m4_encrypt_firmware": 1,
        "m4_fw_encryption_mode": 2,
        “m4_secure_boot_enable”:1,
        "ta_secure_boot_enable": 1
    },
    "key_desc_table_addr": 768
}
```

Note: NWP only has one field to both enable and configure encryption, while the M4 has one field to enable encryption and a separate field to set the mode.

4. Updating XiP Images

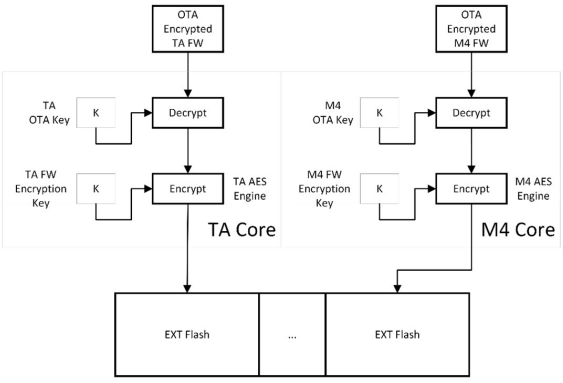

Figure 4.1. OTA Encrypted Firmware Updates Saved for XiP

During firmware update for either core, the SiWx917 Bootloader checks if the encrypted XiP feature is enabled in the MBR for that core. If enabled, the bootloader reads the firmware for that core from the download location and encrypts it with that core's intrinsic XiP key and then saves it at the static target location from where the firmware must execute. If the firmware image in the download location (this encryption is done using the per-core OTA keys, which are different from the per-core XiP encryption keys), the Bootloader decrypts firmware images for a specific core using that core's OTA key and then again encrypts with that core's corresponding XiP key before saving it in the executable region. Firmware images are stored in flash based on device flash configuration. To determine where firmware is stored, consult the following table for your device configuration:

Configuration	Firmware Image Start Address	Size
NWP 4 [MB] / 8 [MB]	0x4011000	1.871 [MB]
NWP Dual Flash	0x4011000	3.87 [MB]
NWP Optimized Common Flash	0x4011000	1.309 [MB]
M4 Common Flash 4 [MB]	0x41C1000	444 [kB]
M4 Common Flash 8 [MB]	0x4201000	2044 [kB]
M4 Common Flash	0x4201000	3004 [kB]
M4 Dual Flash 8 [MB]	0x8011000	3964 [kB]
M4 Optimized Common Flash	0x4171000	1084 [kB]

5. Examples

5.1 Enable encrypted XiP in devices with external Flash

1. Initialize PUF intrinsic keys.

```
commander manufacturing init --mbr default
```

Power Off and on the device to ensure the PUF takes effect.

2. Generate key configuration file with following command

```
commander util genkeyconfig --outfile keys.json --device Si917
```

Keys.json file will contain keys for signing the firmware image and keys for decrypting the image during OTA and keys for secure boot.

You can generate your own keys. It is highly recommended to secure these keys. Once written, the keys remain permanently in the device. If private keys are leaked or lost, the device can no longer be upgraded and becomes vulnerable to compromise.

Note: The keys inside keys.json are used only for encrypting, signing and MIC calculations during OTA process.Intrinsic keys which are generated during PUF initialization are used to encrypt/decrypt FLASH contents.These keys will not be available to the user by any means.

3. Set e-fuses as shown in 3.2 AES-CTR Configuration and 3.4 AES-XTS Configuration, write keys, and efuses into device with this command.

```
commander manufacturing provision --keys keys.json --data mbrEfuses.json
```

Sample mbrEfuses.json file as shown below

```
{
    "puf_activation_code_addr": 8192,
     "efuse_data":
   {
        "m4_encrypt_firmware": 1,
        "m4_fw_encryption_mode": 1,
        “ta_encrypt_firmware”:1,
        "m4_secure_boot_enable": 1,
        “ta_secure_boot_enable”: 1
    },
    "key_desc_table_addr": 768
}
```

4. Download the latest WiseConnect SDK from Silicon labs website. WiseConnect SDK is firmware for NWP processors. Please refer to this page for more details about WiseConnect.
5. Since secure boot is enabled, si917 device expects minimum MIC computed image. But Si917 bootloader is intelligent and flexible enough to accept a signed, encrypted and MIC computed image to be flashed into the device.

Use following commands to encrypt, sign and compute MIC for application and NWP images.

```
 commander rps convert applicationSigned.rps --app applicaiton.rps --mic keys.json --encrypt keys.json --
sign keys.json
commander rps convert NWPFirmwarSigned.rps --taapp NWPFirmware.rps --mic keys.json --encrypt keys.json --
sign keys.json
```

6. Flash both singed and encrypted images into the device using following commands

```
commander rps load applicationSigned.rps -d si917
commander rps load NWPFirmwarSigned.rps -d si917
```

After flashing the images, the user should see their application is running.

5.2 Enable encrypted XiP in devices with external PSRAM

As mentioned in section 2, to execute some AI/ML algorithms and to drive displays internal RAM of a chip is not sufficient. So, an external PSRAM is added. In some cases, code is made to run from PSRAM. Since both code and data reside in external PSRAM, if PSRAM contents are not encrypted bad actor can easily tamper the contents. So SiWx917 has a feature that encrypts code and data while writing/updating into PSRAM and decrypts the contents back before using or executing the contents.

Note: Only configured memory regions are encrypted.

Following are the steps for enabling encrypted XiP in PSRAM.

1. Initialize PUF intrinsic keys. (This is one time process. If it is already done , no need to do again)

```
commander manufacturing init --mbr default
```

   Power Off and on the device to ensure the PUF takes effect.

2. Generate key configuration file with following command. If keys are already available, this step is not required

```
commander util genkeyconfig --outfile keys.json --device Si917
```

3. Set MBR configuraiton as shown further. You can configure 4 sections of PSRAM to enable security. `psram_section_start_add`, `psram_section_start_add` fuses contains starting and end address of section which needs to be protected.

   Note: Address range must be given only in decimal format.

   Use this command to update keys and efuses

```
commander manufacturing provision --keys keys.json --data mbr_security_PSRAM.json
```

   Sample mbr_security_PSRAM file is shown further

```
{
    "puf_activation_code_addr": 8192,
    "valids":
    {
        "psram_security_segments_valid":1
    },
    "psram_section_start_add":[
        0,
        1048576,
        2097152,
        3145728
    ]
    "psram_section_end_add":[
        1048575,
        2097151,
        3145727,
        4194303
    ],
    "key_desc_table_addr": 768
}
```

4. Build a sample app, such as “psram_blinky” found https://github.com/SiliconLabs/wiseconnect/tree/master/exam-ples/si91x_soc/peripheral/psram_blinky, as well as in the WiSeConnect SDK available in the Simplicity Studio IDE.
5. PFlash the signed, encrypted image and press the reset button on the WPK board. The application will run as expected.

5.3 How to ensure that PSRAM contents are encrypted

1. Take example program like “psram_driver_example” found at https://github.com/SiliconLabs/wiseconnect/tree/master/exam-ples/si91x_soc/peripheral/psram_driver_example
2. Modify MBR flags as shown above in section 5.2.
3. Add following Snippet code in main.c after sl_si91x_psram_init() is called
4. Users can observe that data read in auto mode is same as data wrote, whereas data read in manual mode is 0. Since data needs to be protected, data is not read in manual mode from secure sections. To know more about auto and manual mode refer to https://docs.silabs.com/d/wiseconnect-api-reference-guide-si91x-peripherals/3.2.0/psram

```
/// Auto Write to PSRAM in secure area
psram_read_address = PSRAM_BASE_ADDRESS + 0x1040000;
uint8_t* psramBufWrtPtr = (uint8_t*)psram_read_address;
for (uint32_t index = 0; index < BIT_8_READ_WRITE_LENGTH; index++) {
    psramBufWrtPtr[index] = testBuf[index];
}
DEBUGOUT("Reading back data in auto mode: \r\n");
for (size_t i = 0; i < 10; i++) {
DEBUGOUT("0x%08X 0x%02X 0x%02X",(unsigned int)&psramBufWrtPtr[i],testBuf[i],psramBufWrtPtr[i]);
DEBUGOUT("\r\n");
}
DEBUGOUT("Reading back data in manual mode: \r\n");
sl_si91x_psram_manual_read_in_blocking_mode((uint32)psramBufWrtPtr, verifyBuf, sizeof(uint8_t), 10);
for (size_t i = 0; i < 10; i++) {
DEBUGOUT("0x%08X 0x%02X 0x%02X",(unsigned int)&psramBufWrtPtr[i],testBuf[i], verifyBuf[i]);
DEBUGOUT("\r\n");
}
```

6. Revision History

Revision 1.0

Feburary, 2025

Initial release.

Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support & Community
www.silabs.com/community

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

www.silabs.com