AN1428: SiWx917 Debug Lock

This application note describes the debug lock and unlock feature present on SiWx917 devices. It also provides examples of the procedure to lock debug ports, preventing firmware readout on locked devices.

For more information on provisioning the SiWx917 for debug lock, see the following: UG574: SiWx917 SoC Manufacturing Utility User Guide. For more information on enabling secure boot on SiWx917, see AN1442: SiWx917 SoC Secure Boot with Anti-Rollback Protection.

KEY POINTS

- Debug lock and unlock process for SiWx917
- Debug challenge overview and debug token format
- Prequisites for debug lock
- Examples for debug lock and unlock using a debug token or private key

Table of Contents

1. SiWx917 Security Features

Protecting IoT devices against security threats is central to a quality product. Silicon Labs offers several security options on the SiWx917 to help developers build secure devices, secure application software, and secure paths of communication to manage those devices.

The SiWx917 consists of the following core security functions:

- Secure Boot: Process where the initial boot phase is executed from an immutable memory (such as ROM) and where code is authenticated before being authorized for execution.
- Encrypted XiP: Process that adds confidentiality when instructions are executed in place from off-die or off-chip storage.
- Debug Lock: Used to lock devices to prevent unauthorized memory access for operational security, and can be unlocked when access is required.

User Assistance

The following table summarizes the key security documents:

Table 1.1. Relevant Documentation for SiWx917

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 related issues with SiWx917
AN1428: SiWx917 Debug Lock	Describes 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
UG162: Simplicity Commander Reference Guide	Describes commands available in Simplicity Commander for provisioning eFuses for secure boot and anti-rollback protection
UG574: SiWx917 SoC Manufacturing Utility User Guide	Describes steps for provisioning SiWx917 hardware for production

Key Reference

Signature and debug token validation requires the use of the following cryptographic keys:

Table 1.2. Signature and Debug Token Validation Keys

Key identifier	Description	Key Type	Key Size (bits)	Storage	Lifetime
NWP public key[1,2]	Validates NWP firmware, NWP Debug token	Asymmetric, ECC	256	Flash	Permanent
M4 public key[1,2]	Validates M4 firmware, M4 Debug Token	Asymmetric, ECC	256	Flash	Updatable
Note: 1. Private keys must be kept secure and should be stored as securely as possible. 2. These keys are wrapped for tamper resistance.					

eFuse Reference

Signature validation and debug lock features in the SiWx917 are programmable security features set in eFuses. During development, these options should be set using the Master Boot Record (MBR).
The MBR is stored in flash and contains information like clock frequencies, offsets of structures like eFuse copy, SPI configurations, External Flash details, etc. There are separate MBRs for NWP and M4 at the beginning of their respective flash regions. Any SiWx917 IC that is shipped out of the factory will have a default MBR. Using the OPN of a particular device, the user can update the MBR. For more information on manipulating the MBR, consult UG574: SiWx917 SoC Manufacturing Utility User Guide.

Once development is done, these eFuse options should be set using the eFuses in NWP OTP memory. This securely stores eFuse settings to ensure security features cannot be disabled after production. The examples in this document provide information on programming the MBR for development and programming OTP in production.

The eFuse settings relevant to signature validation and debug locking are summarized in the following table:

Table 1.3. Debug Lock eFuses

eFuse Name	Description	Requirement for Debug Lock
m4_digital_signature_validation	Enables authentication of M4 firmware before executing	Required
ta_digital_signature_validation	Enables authentication of NWP firmware before executing	Required
disable_m4_jtag	Locks the JTAG port of the M4 core	Required
disable_ta_jtag	Locks the JTAG port of the NWP core	Required
disable_m4_access_frm_tass_sec	Enables Secure Zone	Optional

2. Debug Lock Overview

Debug locking is a foundational security feature available on the SiWx917. Without it, unauthorized access to these devices is possible.

Both cores, the NWP and the M4, in the SiWx917 have standard Joint Test Action Group (JTAG) debug interfaces providing access to memory, registers, and firmware debugging capabilities. Unauthorized access to these debug interfaces can leave a system's sensitive data vulnerable to being read or modified. Locking debug ports helps to prevent this unauthorized access from reading or modifying data. Silicon Labs always recommends locking the debug ports of these devices as a final step of production.

Equally as important to locking a debug port is the process of securely unlocking the debug port. When a device is returned from field and needs to be analyzed to understand the failure, the debug port must be unlocked securely. Unlocking the debug port requires a challenge and authentication process to ensure the entity initiating the unlock is authorized to unlock and access any sensitive data that may be stored on the device.

On the SiWx917, signature validation must also be enabled on each core to enable debug locking. Refer to AN1442: SiWx917 Secure Boot with Anti-rollback for more information on signature validation.

2.1 Sequence for Locking and Unlocking Both Cores

On the SiWx917, the debug lock feature is available for both the M4 application core and the NWP security core. Each core can be locked individually, or both can be locked together. Silicon Labs recommends for both cores to be locked as the final step of production.

When locking both cores, take special consideration to lock and unlock the debug ports. The following sequence should be used in these cases. If this sequence is not followed, an error may be thrown, or unexpected behavior may occur.

1. Lock NWP
2. Lock M4
3. Unlock M4
4. Unlock NWP

2.2 Secure Zone and Debug Lock

Secure Zone is an optional feature that can be enabled with debug lock in order to isolate the M4 and NWP cores. When Secure Zone is disabled, the M4 can access the NWP via a mailbox interface. When Secure Zone is enabled, this mailbox interface is disabled, preventing M4 access to the NWP.

Special considerations should be taken when using the debug lock feature of the SiWx917 with and without Secure Zone. The tables below describe the expected behavior in each scenario.

Table 2.1. Debug Lock Behavior when Secure Zone is Enabled

	NWP Locked	NWP Unlocked
M4 Locked	M4 and NWP JTAG cannot be accessed	NWP JTAG can be accessed, M4 JTAG cannot be accessed, Secure Zone is disabled in hardware by bootloader
M4 Unlocked	M4 JTAG can be accessed, NWP JTAG cannot be accessed	M4 and NWP JTAG can be accessed, Secure Zone is disabled in hardware by bootloader

Table 2.2. Debug Lock Behavior when Secure Zone is Disabled

	NWP Locked	NWP Unlocked
M4 Locked	M4 and NWP JTAG cannot be accessed	NWP JTAG can be accessed, M4 JTAG cannot be accessed
M4 Unlocked	M4 JTAG can be accessed, NWP JTAG cannot be accessed. However, M4 can access NWP memory, registers, etc.	M4 and NWP can be accessed

2.3 Permanent Debug Lock

Each core in the SiWx917 uses a corresponding firmware public key to verify the authenticity of debug tokens. To permanently lock the JTAG ports for each core, preventing the ability to unlock the NWP and M4 cores of the SiWx917, the corresponding firmware public keys must **not** be provisioned to the device, while the disable JTAG eFuses for each core are set in OTP.

Note: Permanently locking JTAG port(s) on the SiWx917 is a one-time operation and cannot be reversed.

3. Debug Lock and Unlock Process

The following sections will describe the debug lock and unlock process, the debug token format, and how unlocking is accomplished using only a private key.

The first step in starting the debug lock process is to initiate a debug challenge. This is done by sending a lock command to the device, either through Simplicity Commander or the ISP bootloader. When a lock command is sent, a nonce is generated by the TRNG on the device. Once generated, the nonce is encrypted with the NWP's PUF-derived intrinsic key, and is stored in flash. The nonce generation and encryption process is the same for each core, except for the separate flash locations where each core's encrypted nonce is stored.

Once stored in flash, a copy of the encrypted nonce is returned to the host to create a debug token. The corresponding debug port will be locked after a power cycle. More details on the debug token format are covered in the following section.

Note: The debug lock register is a write-once register that can only be written to by the NWP after reset. For a change of state to occur when debug locking and unlocking, the device must be power cycled to set or clear the debug lock register.

When the debug port needs to be unlocked, the debug token is sent from the host to the NWP to initiate the unlock sequence. Upon receipt of the debug token, the NWP will verify the signature on the debug token, then will place the encrypted nonce in flash. The two encrypted nonces, one previously stored in flash when locking the device, and one received from the debug token, will be decrypted by the NWP's intrinsic key and compared. If the values match, the debug port for the corresponding core will be unlocked. If they do not match, the debug port will remain locked.

Once a debug session is complete, the debug challenge should be rolled. This initiates the generation of a new nonce, which starts the debug lock process again. When the challenge is rolled, a new encrypted nonce will be stored in flash, and the nonce in flash from the previous debug token will no longer be valid. The device will need to be power cycled to lock the debug port again. Once locked, a valid debug token or a private key is needed to unlock the device.

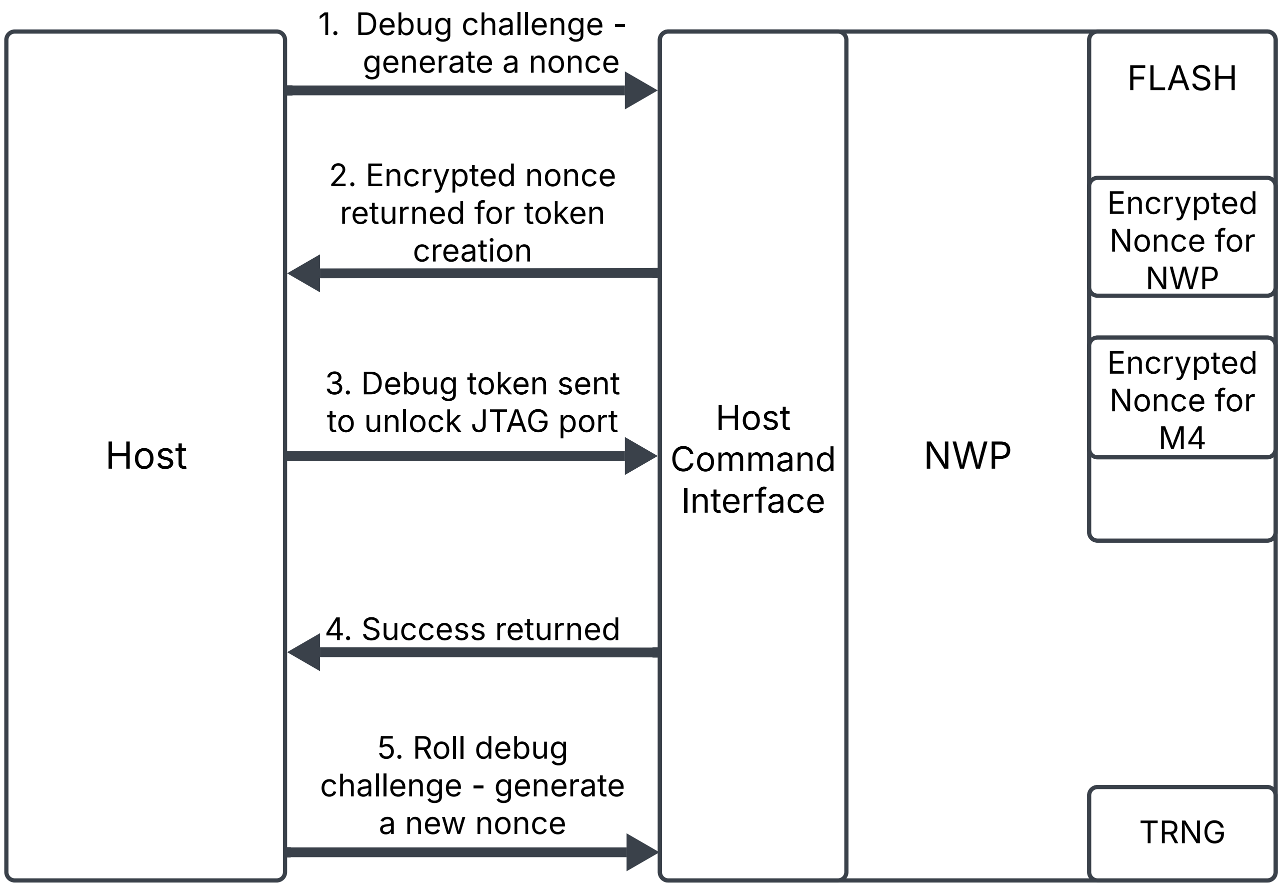
1. Debug challenge - generate a nonce
2. Encrypted nonce returned for token creation
3. Debug token sent to unlock JTAG port
4. Success returned
5. Roll debug challenge - generate a new nonce
Host
Host Command Interface
NWP
FLASH
Encrypted Nonce for NWP
Encrypted Nonce for M4
TRNG

Figure 3.1. Debug Lock and Unlock Process Overview

Unlocking with a Private Key

As shown in the following examples, a user has the option to either unlock the device using the debug token generated when locking the device, or by unlocking the debug port using only the core-specific private key. The process for unlocking the device using only the private key is the same process behind the scenes as what is documented in the previous section. When initiating an unlock using the private key, the device will be prompted to generate and store a new encrypted nonce, return a copy of the encrypted nonce, create a token to be signed by the specified private key, then send the debug token created back to the NWP. This process happens in the background, so it is not seen by the user. After a debug session is complete, the debug challenge should be rolled and a power cycle should occur so that the debug port is locked again.

3.1 Debug Token

A debug token is created during debug locking, and can be saved by the user for unlocking at a later time. The debug token is specific to each core and comprises four data fields: an encrypted nonce, a core command, user data, and a signature.

The encrypted nonce is a 16 byte field in the debug token which guarantees uniqueness of the debug token. When unlocking, the encrypted nonce from the debug token will be passed to the device, decrypted with the device's PUF-derived intrinsic key, and compared to the decrypted value of the nonce generated during debug locking. This guarantees that debug tokens are only usable on the device that generated the encrypted nonce.

The core command is a 1 byte field used to indicate which core to unlock when the debug token is sent. The core command can only be two values, either 0x74, indicating a "t" in ASCII, which corresponds to the NWP core, or 0x6D, indicating an "m" in ASCII, which corresponds to the M4 core.

The user data field is a 7 byte field of padding that can be optionally used to store user data. For example, a user may want to implement a versioning scheme to track versions of debug tokens.

The signature field is a 72 byte ECDSA P-256 signature, and the last field of data in the debug token. The signature is used for authentication, to ensure the user sending the token is authorized to unlock the debug port. The signature is created using the core-specific private key, and is verified on the device by the NWP using the corresponding public key. If the signature verification fails when a debug token is sent, the debug port will remain locked.

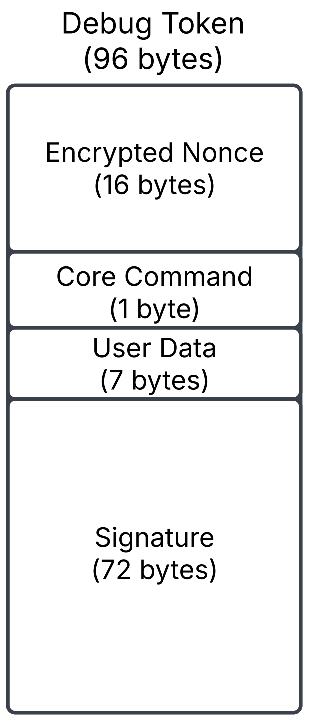

Figure 3.2. Debug Token Format

4. Failure Analysis

Debug Locked Devices

Silicon Labs will accept debug locked SiWx917 devices for failure analysis. To unlock the device to perform failure analysis, a valid debug token is needed. Follow the procedures listed in 5. Examples to generate a valid debug lock token for each locked core and each device returned.

Permanent Locked Devices

Silicon Labs is unable to accept permanently locked SiWx917 devices for failure analysis.

5. Examples

5.1 Requirements

Before proceeding to the examples for debug lock, ensure the following requirements are met. If each of these requirements are not used, debug locking and unlocking may not occur as expected.

5.1.1 Simplicity Commander

Silicon Labs recommends using Simplicity Commander version 1v17p0 (1.17.0) or later for SiWx917 operations. To check the version of commander installed, enter the following command.

```
commander -v
```

```
Simplicity Commander 1v17p1b1824

JLink DLL version: 7.96t
Qt 5.15.2 Copyright (C) 2017 The Qt Company Ltd.
EMDLL Version: 0v19p11b768
mbed TLS version: 2.16.6

DONE
```

5.1.2 Connectivity Firmware

Silicon Labs strongly recommends upgrading connectivity firmware for the NWP to the latest version available to include the latest features, bug fixes, and security patches. Refer to the Simplicity Studio User's Guide for instructions on updating the connectivity firmware through Simplcity Studio. Refer to the WiSeConnect SDK Release Notes for details on each connectivity firmware release.

5.1.3 ISP Mode

Wireless Pro Kit Adapter Firmware

Provisioning security settings onto a factory new SiWx917 requires access to the ISP bootloader through the UART. On the BRD4002A Wireless Pro Kit (WPK), access is available through the virtual com port (VCOM). The WPK adapter firmware version required for communication with the ISP bootloader is 1v5p0b240 or greater. Refer to the Simplicity Studio User's Guide for details on updating adapter firmware.

Entering ISP Mode

To enable access to the ISP bootloader through the VCOM port, the following steps are required:

1. Connect the WPK board by USB to your computer.
2. Open Simplicity Studio.
3. Right click the WPK in the "Debug Adapters" panel and select the "Launch Console" menu item.
4. Select the "Admin" tab.
5. Enter the following in the text entry box and press enter "serial vcom config handshake aux".
6. Close Simplicity Studio.

To Enter ISP mode, press and hold the ISP button on the SiWx917 radioboard (for example, BRD4338A) while pressing and releasing the reset button on the WPK (BRD4002A), and before releasing the ISP button.

To verify that the device is in ISP mode, open a terminal window, and use the UART settings listed in Table 5.1 Bootloader UART Settings on page 11 for communication with the ISP bootloader. Once in ISP mode, enter any key in the terminal window. You then should see a prompt stating "Enter 'U'" in the terminal window.

Table 5.1. Bootloader UART Settings

Setting	Value
Baud rate	115200
Parity	None
Data bits	8
Stop bits	1

5.1.4 Signature Validation

Before proceeding to the examples section, complete the steps listed in the Examples section of AN1442 for provisioning a device and signing an application for signature validation. It is recommended to flash the signed application image before proceeding with enabling debug locking.

5.2 Locking Debug Access

This section demonstrates how to lock the JTAG ports of the SiWx917. When enabling this feature in development, eFuses should be set in MBR. In production, these eFuse settings should be programmed in eFuse data in OTP. In this section, the MBR settings will be used to enable the debug locks.

In this example, a copy of the debug token is saved locally when locking the JTAG ports. Saving a copy of the debug token is optional but recommended because the debug token can be used later to unlock a JTAG port without needing to access a private key.

Note: Before continuing with the steps outlined in this section, ensure all requirements listed in 5.1 Requirements are met, including enabling signature validation.

1. Read out the MBR file.

```
commander mfg917 read tambr --out tambr.json
```

```
Reading data from region: tambr
Reading 496 bytes from 0x04000000
Writing JSON...
Manufacturing data saved to file 'tambr.json'
DONE
```

2. Edit MBR contents to set the disable JTAG eFuses.

```
 {
    "valids": 0,
    "puf_activation_code_addr": 8192,
    "valids": 0,
 "efuse_data": {
        "m4_digital_signature_validation": 1,
        "ta_digital_signature_validation": 1,
        "disable_ta_jtag":1,
        "disable_m4_jtag":1
    },
    "key_desc_table_addr": 768
}
```

3. First, enter ISP mode using the steps listed in 5.1.3 ISP Mode. Once in ISP mode, write the updated MBR file to flash.

```
commander mfg917 write tambr --data tambr.json
```

```
Reading JSON...
Updating CRCs...
Writing data to region: tambr
Loading RAM code (321776 bytes)...
Starting RAM code...
Data loaded successfully
Region 'tambr' was successfully written to device.
DONE
```

4. Generate a nonce to lock access to the NWP core and save the token. The NWP core must be locked before the M4 core lock is applied.

```
commander serial lock ta --token debug_nwp.token --key ta_private_key.pem --serialport COM3 --device si917
```

```
Initializing debug lock...

Nonce generated by the device: 0C8BDB0DF4936171F6977E3237D3FED2

Debug access will be locked after the device is reset.

Image SHA256: 434b15abf5e30f1a2eea7782799723d9e9c7cd78462a82cae7c010b8bb5a0144

R = 0E9BF63D25AFFD678390D402EC665D004FC98B3457417F02B37EC14790020F8F

S = 1865648147A4A906BDAA095A45051B02FC10E3E3B2B2DF642ED5EF39A9953417

Debuglock token raw data:

0c8bdb0df4936171f6977e3237d3fed2740000000000000000304402200e9bf63d25affd678390d402ec665d004fc98b3457417f02b37ec
14790020f8f02201865648147a4a906bdaa095a45051b02fc10e3e3b2b2df642ed5ef39a99534170000

Debuglock token written to 'debug_nwp.token'.

DONE
```

Note: This command will also accept a keys.json file when specifying the key used to sign the debug token.

5. Perform a hard reset to lock the NWP JTAG port.
6. Generate a nonce to lock access to the M4 core and save the token.

```
commander serial lock m4 --token debug_m4.token --key m4_private_key.pem --serialport COM3 --device si917
```

```
Initializing debug lock...

Nonce generated by the device: 603B663BFD006CC79EB61F805FC8D3F6

Debug access will be locked after the device is reset.

Image SHA256: 56bdc05ecf5732b0b1374a99d8f78835a3edc38e0a8755da4ffd3bb5e031174f

R = E0026EA7FD4064E8E15B651A3251FD8071B0DEDE1EF802D7A84A0FAD491B3B3E

S = 8273DD099F1AB3B4FC05048438778ABAF908B9ED7DBB4A960CC618F3F8BFFEB6

Debuglock token raw data:

603b663bfd006cc79eb61f805fc8d3f66d00000000000000003046022100e0026ea7fd4064e8e15b651a3251fd8071b0dede1ef802d
7a84a0fad491b3b3e0221008273dd099f1ab3b4fc05048438778abaf908b9ed7dbb4a960cc618f3f8bffeb6

Debuglock token written to 'debug_m4.token'.

DONE
```

Note: This command will also accept a keys.json file when specifying the key used to sign the debug token.

7. Perform a hard reset to lock the M4 JTAG port.

5.3 Unlocking Debug Access with a Token

1. Unlock the M4 JTAG port using the debug token generated in Step 6 of 5.2 Locking Debug Access.

```
commander serial unlock m4 --token debug_m4.token --serialport COM36 --device si917
```

```
Using the provided token 'debug_m4.token' to unlock debug access...

Verifying debuglock token 'debug_m4.token'...

Initializing debug unlock...

Debug access will be unlocked after the device is reset.

DONE
```

2. Perform a hard reset to unlock the M4 JTAG port.
3. Unlock the NWP JTAG port using the debug token generated in Step 4 of 5.2 Locking Debug Access.

```
commander serial unlock ta --token debug_nwp.token --serialport COM3 --device si917
```

```
Using the provided token 'debug_nwp.token' to unlock debug access...

Verifying debuglock token 'debug_nwp.token'...

Initializing debug unlock...

Debug access will be unlocked after the device is reset.

DONE
```

4. Perform a hard reset to unlock the NWP JTAG port.
5. Once the debugging session is complete, update the challenge nonce by locking the ports again as described in 5.2 Locking Debug Access.

5.4 Unlocking Debug Access with a Private Key

Another option for unlocking the JTAG port is to use the private key per core to unlock the port, instead of using a pre-saved token. In the background, when a core is unlocked using a private key, Simplicity Commander will ask the device to generate and return a new nonce, and will package the nonce, signature, core command, and other data as a token. Simplicity Commander then sends this new token to the device to unlock the core.

1. Unlock the M4 JTAG port.

```
commander serial unlock m4 --key m4_private_key.pem --serialport COM36 --device si917
```

```
Using serial port 'COM36' for file transfers.
Initializing debug unlock...
Nonce generated by the device 6D6DA3100AB5AD6D2B5BA1B62478652B
Parsing signing key 'm4_private_key.pem'...
Debuglock token raw data:
6d6da3100ab5ad6d2b5ba1b62478652b6d00000000000000003046022100d803b7633f74a019d7e3656f9d72a7ed7b4
fbb2c85b67bffd79e355560022100ed2b57569f2101f3a6006ea119bf1f53577e72b8374b31cce79a480be34de6b3
Debuglock token written to 'C:/Users/../../../debugtoken'.
Debug access will be unlocked after the device is reset.
DONE
```

   Note: This command will also accept a keys.json file when specifying the key used to sign the debug token.

2. Reset the part to unlock the M4 JTAG port.
3. Unlock the NWP JTAG port. The NWP port will only be unlocked if the M4 debug port is unlocked.

```
commander serial unlock ta --key ta_private_key.pem --serialport COM36 --device si917
```

```
Using serial port 'COM36' for file transfers.
Initializing debug unlock...
Nonce generated by the device 6D6DA3100AB5AD6D2B5BA1B62478652B
Parsing signing key 'ta_private_key.pem'...
Debuglock token raw data:
6d6da3100ab5ad6d2b5ba1b62478652b6d00000000000000003046022100d803b7633f74a019d7e3656f9d72a7ed7b4
fbb2c85b67bffd79e355560022100ed2b57569f2101f3a6006ea119bf1f53577e72b8374b31cce79a480be34de6b3
Debuglock token written to 'C:/Users/../../../debugtoken'.
Debug access will be unlocked after the device is reset.
DONE
```

   Note: This command will also accept a keys.json file when specifying the key used to sign the debug token.

4. Reset the part to unlock the NWP JTAG port.
5. Once the debugging session is complete, update the debug challenge nonce by locking the port as described in 5.2 Locking Debug Access.

5.5 Moving to Production

This section describes the steps recommended for moving to production. The main difference between development and production is that during development, the master boot record (MBR) is used to configure security settings so that they can be changed if needed and the JTAG ports are enabled. While in production, the eFuses (in OTP) are used to make the security settings immutable and the JTAG ports are locked.

1. Read the eFuses from a factory new device.

```
commander mfg917 read efuse --out efuse.json --device SIWG917M111MGTBA
```

```
Reading data from region: efuse
Reading 1024 bytes from 0x40012000
Writing JSON...
Manufacturing data saved to file 'efuse.json'
DONE
```

2. Before proceeding to this step, verify all other security settings are enabled. Note the required eFuses for debug locking as described in Table 1.3 Debug Lock eFuses on page 4. As a final step in production, disable the JTAG ports by making the following changes to ‘efuse.json’ in a text editor.

```
"mcu_chip_modes": {
        ...
        "disable_m4_jtag": 1,
        ...
    },
"wireless_chip_modes": {
        ...
        "disable_ta_jtag": 1,
        ...
    }
```

3. Save the file ‘efuse.json’.
4. Write the new eFuse settings back to the target device as follows:

```
commander mfg917 write efuse --data efuse.json -d SIWG917M111MBTGA
```

```
Reading JSON...
The provided data can be applied to the current Efuse.
Determining which OTP words need updating...

================================================================================
THIS IS A ONE-TIME command which PERMANENTLY writes data to the Efuse region
of your device. This command CANNOT be undone.
Type 'continue' and hit enter to proceed or Ctrl-C to abort:
```

```
continue
```

```
Writing 16 bytes to OTP...
Loading RAM code (321776 bytes)...
Starting RAM code...
Efuse was successfully written to the device's OTP.
DONE
```

5. Follow Steps 4-7 outlined in 5.2 Locking Debug Access to generate a nonce and lock each debug port.

6. Revision History

Revision 0.3

February 2025

- Updated front page
- Removed Introduction
- Added SiWx917 Security Feature
- Updated Debug Lock Overview
  - Added Sequence for Locking and Unlocking Both Cores
  - Added Secure Zone and Debug Lock
  - Added Permanent Debug Lock
- Renamed Architectural Overview to Debug Lock and Unlock Process
- Updated Debug Lock and Unlock Process
- Added Failure Analysis
- Updated examples section
  - Added Requirements
  - Updated instructions for locking and unlocking
- Replaced "TA" and "NWPSS" with NWP

Revision 0.2

May 2024

- Updated name to "SiWx917 Debug Lock"
- Replace "ThreadArch" with "Network Wireless Processor"
- Replaced "TSS" with "NWPSS"

Revision 0.1

February 2024

- Initial revision

IoT Portfolio
www.silabs.com/products

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