SSI Peripherals

Copyright © 2025 Silicon Laboratories. All rights reserved.

SSI Peripherals

Serial Synchronous Interface (SSI) Developer Guide

This section introduces the Serial Synchronous Interface (SSI) driver for the SiWx917 platform.

It describes the purpose and audience of this guide, defines its scope, and lists the required hardware and software components for development.

About This Guide

This guide provides step-by-step instructions for using the SSI peripheral driver on SiWx917 devices.

It explains how to configure, integrate, and optimize SSI functionality within your embedded application, covering implementation details, configuration options, API usage, and best practices.

Audience

This guide is intended for embedded developers who implement and integrate SSI communication features on the SiWx917 platform.
It provides practical information about the SSI peripheral architecture, initialization, configuration, low-power operation, and debugging.

Purpose

The SSI driver offers a flexible, programmable interface for synchronous serial communication.
It allows the SiWx917 device to operate as either a leader (primary) or follower (secondary). The driver supports multiple industry-standard protocols for broad interoperability across embedded systems.

Scope

This guide covers the following topics:

- Software-level implementation of SSI communication for SiWx917 devices
- Development using the WiSeConnect SDK within Simplicity Studio
- Initialization and configuration of parameters such as clock source, baud rate, frame format, data width, clock polarity (CPOL), and clock phase (CPHA)
- Usage of SSI APIs, low-power operation, error handling, and debugging
- Practical examples and best practices for integrating SSI into real-world embedded applications

```
Note: The SSI driver aligns with Silicon Labs' unified peripheral framework, ensuring
consistency and easy migration between SiWx917 peripherals.
```

Hardware and Software Components

The following hardware and software components are required to use the SSI driver effectively.

Hardware Requirements

- SiWx917 Silicon Labs evaluation kit or module

  - View supported hardware
- Serial console for runtime logging and debugging
- Logic analyzer for signal monitoring and validation

Software Requirements

- Simplicity Studio Integrated Development Environment (IDE)
  - Provides project creation, debugging, and profiling tools
- WiSeConnect SDK
  - Includes drivers, APIs, and example projects for SSI development

> Tip: The WiSeConnect SDK integrates seamlessly with Simplicity Studio, simplifying peripheral configuration through the Project Configurator and Unified Configurator (UC) tools.

> Note:
>
> - Always verify your SDK and Firmware versions match the GSPI API definitions described in this guide.
> - This guide adopts Primary/Secondary terminology in place of the legacy Master/Slave SPI terminology. However, where required for register names, signal identifiers (such as MOSI and MISO), or backward compatibility, the terms Master or Slave may still appear in technical references.

Architecture

Synchronous Serial Interface (SSI) Architecture

Peripheral Overview

The Synchronous Serial Interface (SSI) on SiWx917 is a flexible, full-duplex synchronous serial controller.

It supports Serial Peripheral Interface (SPI), Synchronous Serial Protocol (SSP), and Microwire (µWire) standards, with optional dual and quad data lines for higher throughput. SSI integrates with DMA for offloading the CPU, and supports low-power operation depending on the selected instance and power state.

SSI scales from high-bandwidth use cases, such as external flash memory and display controllers, to energy-efficient transfers where DMA minimizes CPU activity and allows the core to sleep.

General Description

The SiWx917 device includes multiple configurable SSI controllers that support common synchronous serial protocols (SPI, SSP, Microwire).
Each instance can operate as either a primary that initiates communication or as a secondary that responds to an external controller.

On this SoC:

- Two SSI instances support primary operation.
- One SSI instance supports secondary operation.

Key Features and Capabilities

The SSI peripheral on SiWx917 is a high-performance, flexible serial controller that provides the following capabilities:

- Protocol support: Operates as primary or secondary using SPI, SSP, or Microwire (µWire) protocols.
- Dual/Quad support (primary): The HP SSI primary supports single, dual, and quad data lines. The ULP SSI primary supports single-bit operation.
- Data frame size: Programmable 4–16-bit frame length. Microwire control-word length is also configurable (1–16 bits).
- DMA and FIFO integration: Supports DMA handshake signals with programmable FIFO thresholds. Each FIFO is 16 words deep.
- Interrupts: Maskable interrupts for TX empty/overflow, RX full/underflow/overflow, and multi-primary contention (if applicable).
- Programmable RX sample delay: Adjusts sampling at high-speed operation to improve reliability.
- Chip-select fan-out (primary): Up to four chip-select outputs from the HP SSI primary instance (enabled via `SER`).
- ULP SSI primary: The ULP SSI instance supports only one secondary device.
- Clock configuration: Programmable CPOL and CPHA for SPI mode control.
- High-speed operation:
  - HP SSI (Primary): Up to 40 MHz in high-speed mode or 20 MHz in full-speed mode.
  - SSI Secondary: Up to 20 MHz
  - ULP SSI Primary: Up to 10 MHz

SSI Architecture (High-Level)

The SiWx917 SSI uses a programmable synchronous-serial engine with FIFO buffering, DMA handshakes, and interrupt control to achieve high throughput with minimal CPU overhead.

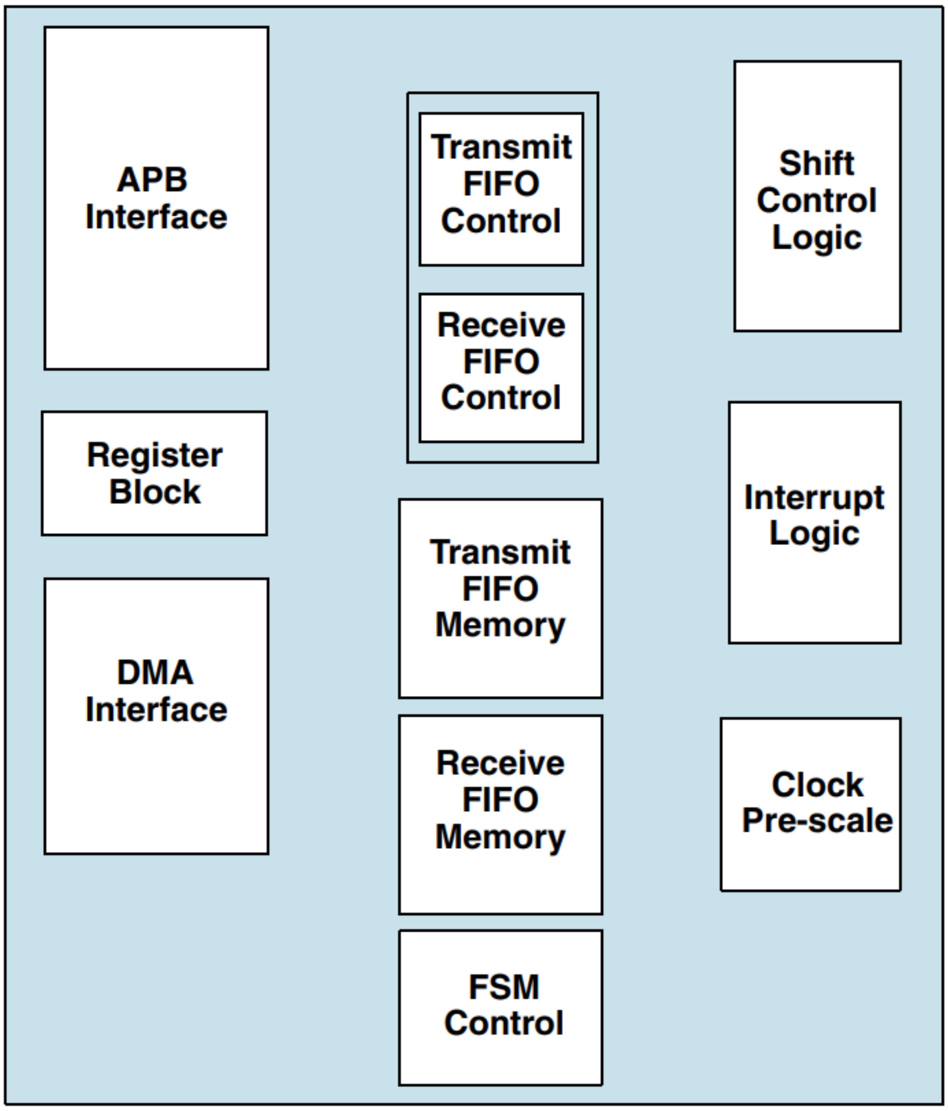


Figure: High-level SSI architecture.

Functional Blocks

Component	Description
APB Interface & Register Block	Provides access to SSI configuration and status registers via the AMBA APB bus.
DMA Interface	Supports high-speed, non-blocking data transfers for both transmit (TX) and receive (RX) paths using DMA handshakes.
Transmit/Receive FIFOs	Dual 16-word buffers for TX and RX, managed by an internal state machine for smooth data flow.
Shift Control Logic	Serializes transmit data and deserializes received data according to the selected frame format.
Interrupt Logic	Generates maskable interrupts for TX/RX thresholds, overflow/underflow conditions, and completion events.
Clock Prescaler	Programmable divider for generating the SSI serial clock, ensuring accurate timing in all modes.

Why It Matters

- High performance: DMA and FIFOs reduce CPU involvement, allowing for continuous data streaming.
- Deterministic timing: The prescaler and shift logic guarantee protocol-compliant edge control.
- Simplified debugging: Status registers and interrupts streamline validation and troubleshooting.

Software Architecture

Copyright © 2025 Silicon Laboratories. All rights reserved.

This section describes the SSI software stack and how the software layers interact during configuration and data transfer.

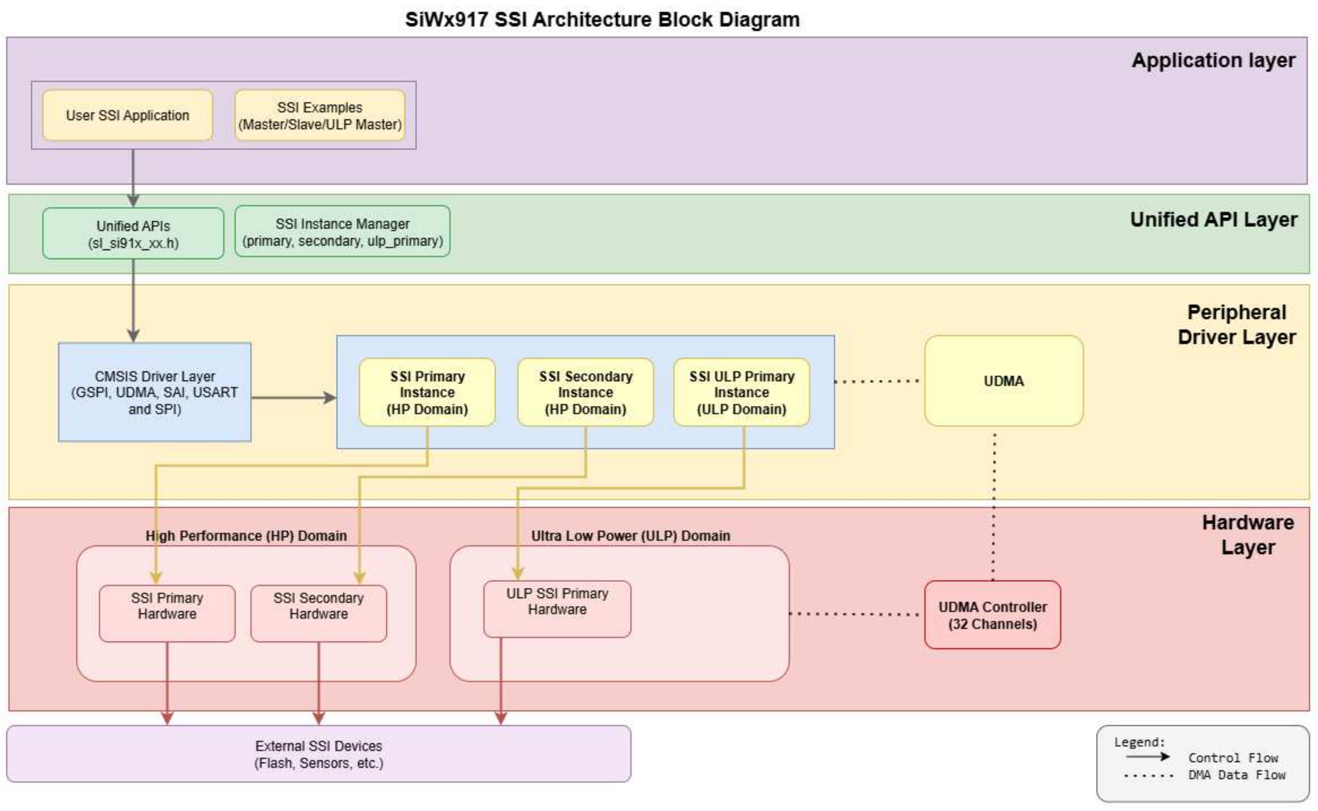

SiWx917 SSI Architecture Block Diagram
Application layer
User SSI Application
SSI Examples (Master/Slave/ULP Master)
Unified API Layer
Unified APIs (sl_si91x_xx.h)
SSI Instance Manager (primary, secondary, ulp_primary)
Peripheral Driver Layer
CMSIS Driver Layer (GSPI, UDMA, SAI, USART and SPI)
SSI Primary Instance (HP Domain)
SSI Secondary Instance (HP Domain)
SSI ULP Primary Instance (ULP Domain)
UDMA
Hardware Layer
High Performance (HP) Domain
SSI Primary Hardware
SSI Secondary Hardware
Ultra Low Power (ULP) Domain
ULP SSI Primary Hardware
UDMA Controller (32 Channels)
External SSI Devices (Flash, Sensors, etc.)
Legend:
Control Flow
DMA Data Flow

Figure: SSI software layering showing control and data paths.

Software Layering Overview

Layer	Description
Application Layer	Contains user applications or example projects that use the SSI API to perform sensor communication, memory access, or display updates.
Unified API Layer (WiSeConnect)	Provides high-level driver APIs for initialization, configuration (baud rate, frame format, data width, CPOL/CPHA), and data transfer, ensuring portability across SSI instances.
CMSIS-Driver Layer	Implements the ARM® Cortex Microcontroller Software Interface Standard (CMSIS) SPI driver for RTOS compatibility and portability.
Peripheral Driver Layer	Manages low-level register programming, FIFO thresholds, and interrupt servicing.
Hardware Layer	Represents the physical SSI instances on the SiWx917 SoC: HP primary, ULP primary, and HP secondary.

Directory Structure in WiSeConnect SDK

```
wiseconnect/
├── components/
│   └── device/
│       └── silabs/
│           └── si91x/
│               └── mcu/
│                   └── drivers/
│                       ├── cmsis_driver/
│                       │   ├── CMSIS/Include/Driver_SPI.h
│                       │   ├── SPI.c
│                       │   └── SPI.h
│                       ├── peripheral_drivers/
│                       │   ├── inc/rsi_spi.h
│                        │   └── src/rsi_spi.c
│                        └── unified_api/s
│                            ├── inc/sl_si91x_ssi.h
│                            └── src/sl_si91x_ssi.c
└── examples/
    └── si91x_soc/
        └── peripheral/
            ├── sl_si91x_ssi_master
            ├── sl_si91x_ssi_slave
            └── sl_si91x_ulp_ssi_master
```

Core Components

SSI Primary (High-performance Primary)

- Purpose: Supports high-speed communication in the HP domain.
- Features: Single, dual, and quad SPI modes.
- Capabilities: Connects to up to four secondary devices.
- Performance: Operates up to 40 MHz.
- Typical Use Cases: External flash access, display interfaces, and high-rate sensor communication.

SSI ULP Primary (Ultra-low-power Primary)

- Purpose: Enables low-power communication in the ULP domain.
- Features: Single-bit SPI mode.
- Capabilities: Connects to one secondary device.
- Performance: Operates up to 10 MHz.
- Power Benefits: Supports DMA-based transfers while the MCU remains in a low-power (PS2) state.
- Typical Use Cases: Battery-powered systems requiring background or sleep-mode data exchange.

SSI Secondary

- Purpose: Operates as a programmable secondary peripheral.
- Features: Supports all SPI modes and framing protocols.
- Capabilities: Provides full-duplex communication.
- Typical Use Cases: Device-to-device communication and multi-primary networked systems.

Clock Architecture and Timing

The SSI clock system manages synchronization between the SSI input clock (`ssi_clk`), peripheral clock (`pclk`), and serial clock output (`sclk_out`).

Clock Domain Relationships

- The SSI input clock (`ssi_clk`) must not exceed the APB clock (`pclk`) for reliable synchronization.
- When `pclk` and `ssi_clk` are asynchronous, synchronization logic is automatically enabled.
- In primary mode, the maximum frequency of the serial clock output (`sclk_out`) is one-half the frequency of `ssi_clk` .The shift logic samples data on one edge and drives it on the opposite edge.

```
Fsclk_out = Fssi_clk / SCKDV
```

Parameters

Parameter	Description
`Fsclk_out`	Serial clock output frequency
`Fssi_clk`	SSI input clock frequency

Parameter	Description
`SCKDV`	A bit field in the programmable register BAUDR, holding any even value in the range 0 to 65,534. If SCKDV is 0, then sclk_out is disabled

Frequency Restrictions

- Primary Mode: `Fssi_clk ≥ 2 × (max Fsclk_out)`
- Secondary Mode: `Fssi_clk ≤ 4 × (max Fsclk_in)`

Operation Modes

This section describes the operation modes of the SSI on the SiWx917.

Primary Mode

In primary mode, the SSI peripheral initiates and controls all data transfers.

The primary generates the clock and manages communication with connected secondary devices.

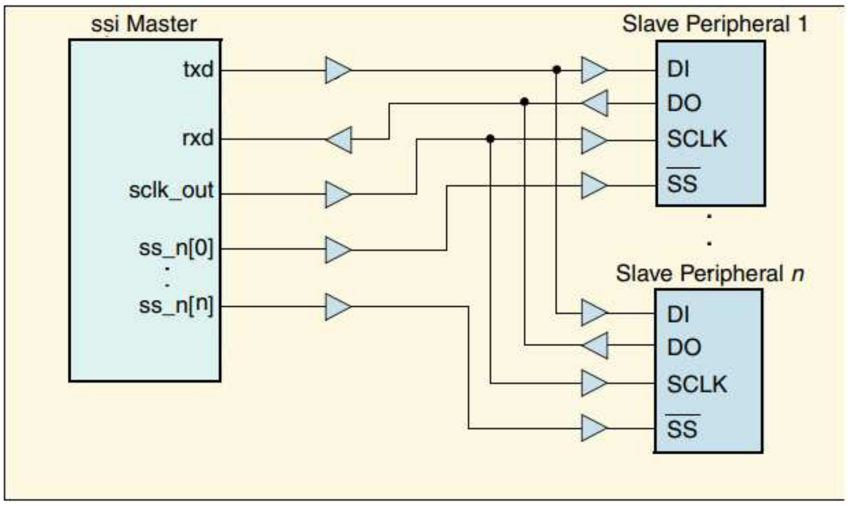


Figure: SSI primary device configuration.

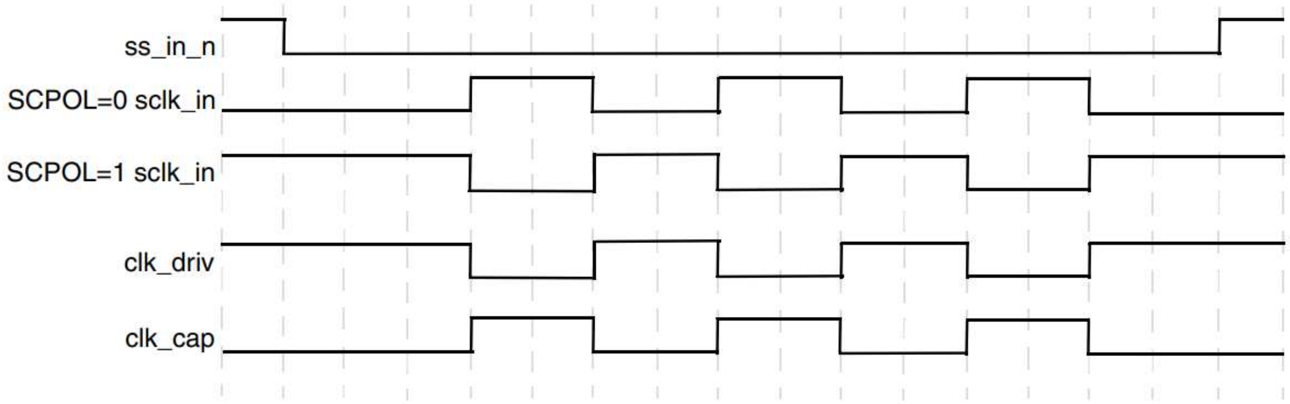
SPI Frame Format

A SPI frame asserts the selected chip-select line, shifts a programmed 4–16-bit word MSB first (`CTRLR0.DFS_32`), and maintains continuous clocking if configured for back-to-back frames.

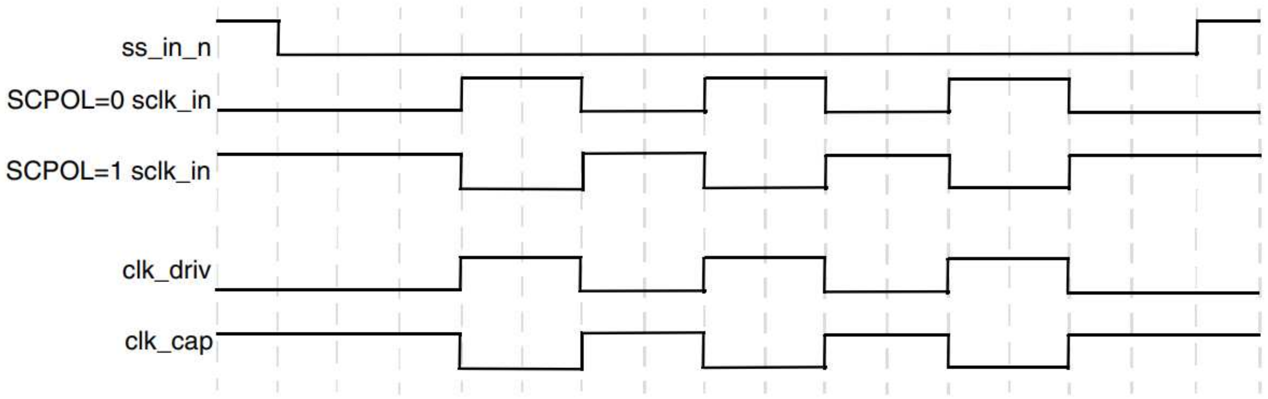
The chip-select (SS) is de-asserted after the final bit unless configured otherwise.

Internally, `clk_driv` launches transmit data (`txd`) and `clk_cap` samples received data (`rxd`).
The clock stops automatically when idle.

Frame Format for Motorola SPI with SCPH = 0



Frame Format for Motorola SPI with SCPH = 1



Clock polarity (CPOL) and phase (CPHA) define the SPI mode (0–3):

Mode	CPOL	CPHA	Idle SCLK	Data Captured On	Data Changes On
0	0	0	Low	Rising edge	Falling edge
1	0	1	Low	Falling edge	Rising edge
2	1	0	High	Falling edge	Rising edge
3	1	1	High	Rising edge	Falling edge

Dual and Quad SPI Modes

The SSI supports dual and quad SPI operation to increase data throughput by transferring multiple bits per clock cycle.

Quad SPI Frame Format

A quad SPI transaction is divided into four configurable phases:

1. Instruction Phase – Sends the command to the target device (length and width configurable).
2. Address Phase – Sends the target address using one, two, or four lines.
3. Wait-Cycle Phase – Inserts programmable wait cycles for timing at high frequencies.
4. Data Phase – Transfers data to or from memory using one, two, or four lines.

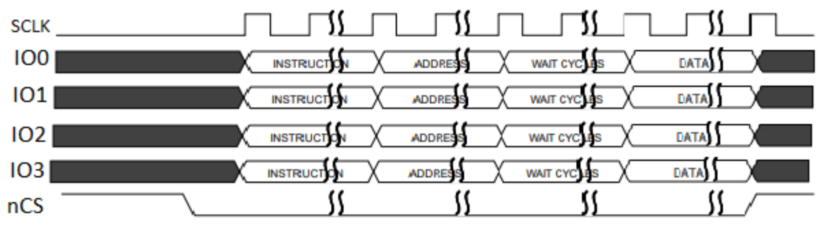


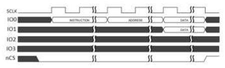
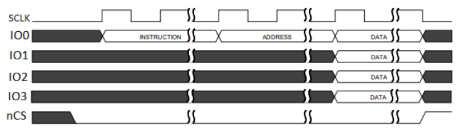
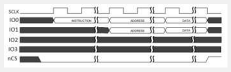
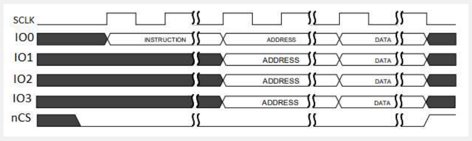
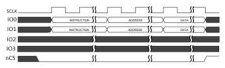
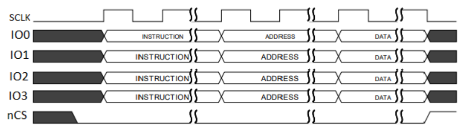
Table: Data Phase Configuration vs. Quad-SPI Functional Modes

Functional Mode	Instruction Lines	Address Lines	Data Lines
Standard	1	1	1
Dual	1 or 2	2	2
Quad	1, 2, or 4	4	4

Instruction and Address Phase Formats in Dual/Quad SPI Modes

The transfer format is controlled by the `TRANS_TYPE` field.

Copyright © 2025 Silicon Laboratories. All rights reserved.

TRANS_TYPE	Instruction Transfer Format	Address Transfer Format	Description	Dual SPI Command Format Diagram	Quad SPI Command Format Diagram
TRANS_TYPE	Instruction Transfer Format	Address Transfer Format	Description	Dual SPI Command Format Diagram	Quad SPI Command Format Diagram
00	Standard SPI	Standard SPI	Instruction and address both use single-bit SPI.		 SCLK IO0 IO1 IO2 IO3 nCS INSTRUCTION ADDRESS DATA
01	Standard SPI	Dual/Quad (per SPI_FRF)	Instruction in Standard SPI, address in Dual/Quad.		 SCLK IO0 IO1 IO2 IO3 nCS INSTRUCTION ADDRESS DATA
10	Dual/Quad (per SPI_FRF)	Dual/Quad (per SPI_FRF)	Both instruction and address in Dual/Quad.		 SCLK IO0 IO1 IO2 IO3 nCS INSTRUCTION ADDRESS DATA

Write Operation (Dual/Quad Modes)

A write transaction includes:

1. Instruction phase
2. Address phase
3. Data phase

Configuration Fields:

- `CTRLR0.SPI_FRF` — Selects data-line width (standard/dual/quad)
- `SPI_CTRLR0.INST_L` and `SPI_CTRLR0.ADDR_L` — Define instruction and address lengths
- `CTRLR0.DFS_32` — Sets the data-frame size (4–16 bits)

Read Operation (Dual/Quad Modes)

A read transaction includes:

1. Instruction phase
2. Address phase
3. Wait cycles (`SPI_CTRLR0.WAIT_CYCLES`)
4. Data phase

Configuration Fields:

- `CTRLR0.SPI_FRF` — Selects data-line width (standard/dual/quad).
- `SPI_CTRLR0.TRANS_TYPE` — Sets instruction and address transfer types.
- `SPI_CTRLR0.WAIT_CYCLES` — Defines dummy cycles between address and data.
- `CTRLR1.NDF` — Specifies number of data frames to receive.
- `CTRLR0.DFS_32` — Sets data-frame size (4–16 bits).

Dependencies

Hardware Dependencies

- GPIO Controller: Configures SCLK, chip-select lines, and up to four data lines (D0–D3).
- DMA Controller: Enables high-throughput, non-blocking data transfers.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Software Dependencies

- `sl_si91x_ssi` – Core driver for initialization, configuration, and transfer operations.
- `sl_si91x_gpio` – Manages pin configuration and routing.
- `sl_si91x_dma` – Provides DMA support for continuous or high-speed transfers.

The Simplicity Studio Project Configurator manages all dependencies, including clock, power, GPIO, and DMA components automatically.

Initialization and Configuration

Serial Synchronous Interface (SSI) Initialization and Configuration

The SiWx917 WiSeConnect SDK provides a unified Serial Synchronous Interface (SSI) driver API.

It supports multiple configurable instances—primary (leader), secondary (follower), and ULP primary—as well as common frame formats (SPI/SSP/µWire), single/dual/quad data-line operation (where supported), and DMA-assisted transfers.

This guide describes the complete initialization and configuration flow for SSI communication on SiWx917 devices.

Initialization and Configuration

Startup Sequence

This section outlines the initialization procedure for both standard (single-line) and dual/quad Serial Peripheral Interface (SPI) modes.

It also explains project setup in Simplicity Studio.

Hardware and Software Requirements

Hardware

- Windows PC
- Si917 Evaluation Kit: WPK (BRD4002) with one of the following radio boards: BRD4338A, BRD4342A, or BRD4343A
- SiWx917 AC1 Module Explorer Kit: BRD2708A

Software

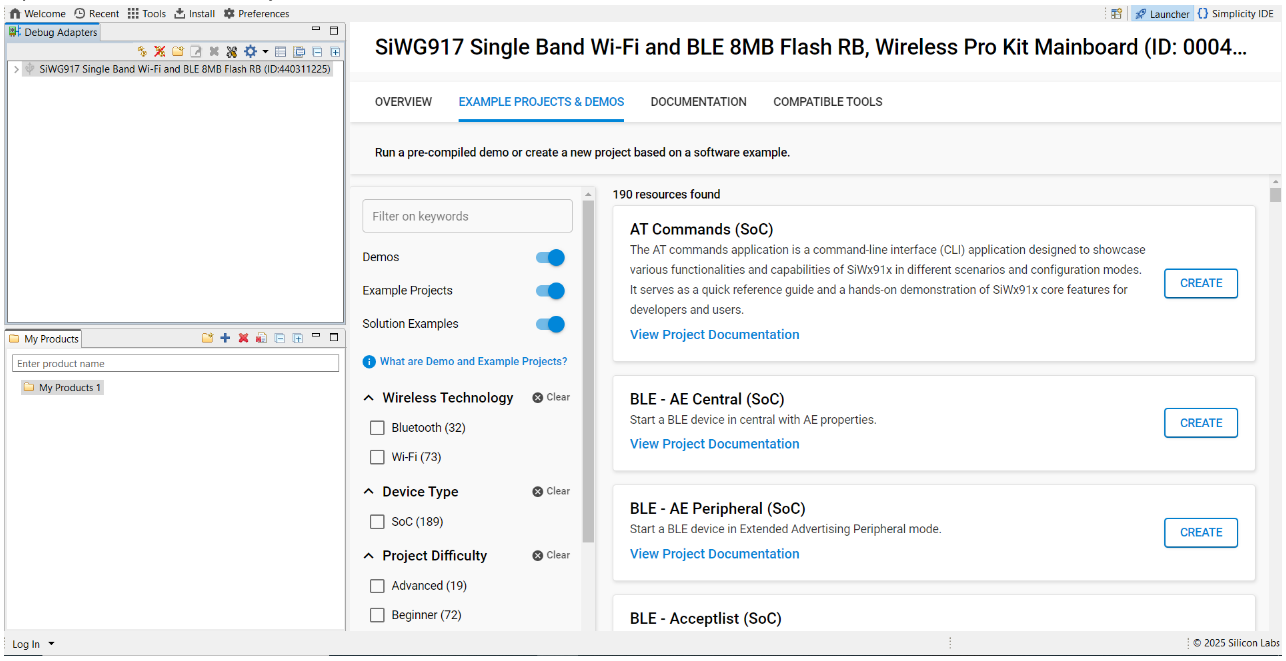
- Simplicity Studio (latest version)
- Serial console for runtime logging

Getting Started with Simplicity Studio

Follow these steps to begin development with the SSI peripheral:

1. Install Simplicity Studio.
   - Download and install from the Silicon Labs website.
2. Connect Your Hardware.
   - Attach your SiWx917 development board via USB.
3. Launch Simplicity Studio.

- Open the IDE and ensure your board is detected.



SiWG917 Single Band Wi-Fi and BLE 8MB Flash RB, Wireless Pro Kit Mainboard (ID: 0004...

Run a pre-compiled demo or create a new project based on a software example.

190 resources found

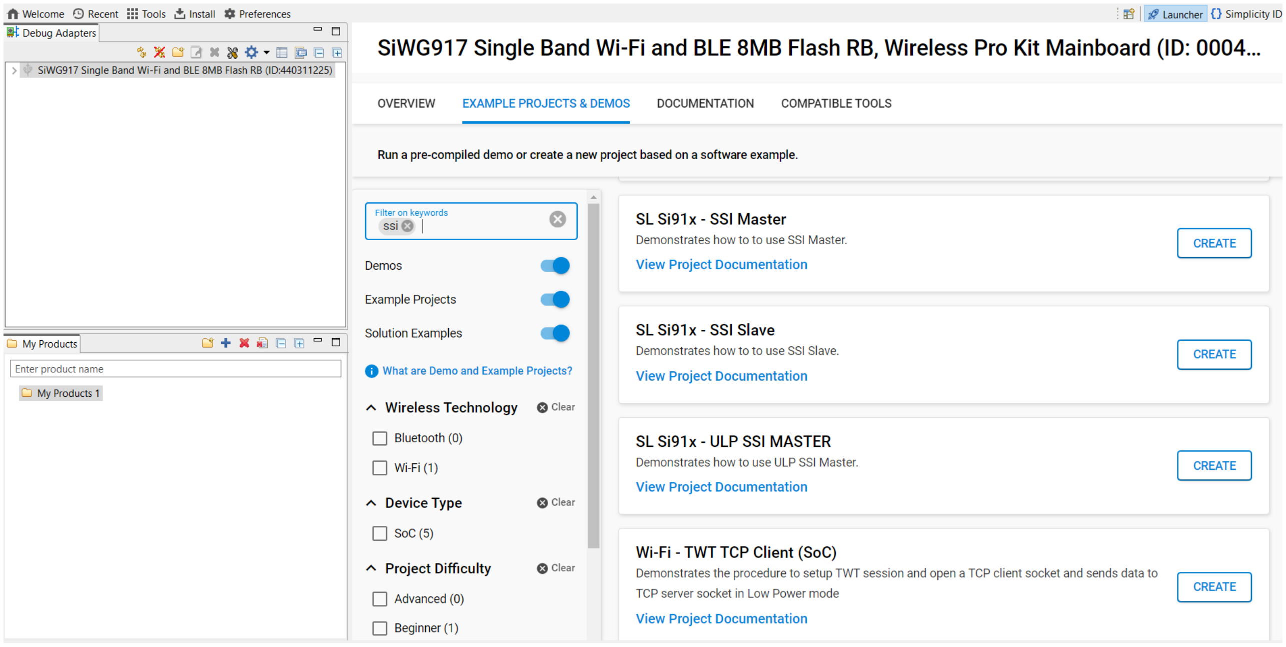
AT Commands (SoC)
The AT commands application is a command-line interface (CLI) application designed to showcase various functionalities and capabilities of SiWx91x in different scenarios and configuration modes. It serves as a quick reference guide and a hands-on demonstration of SiWx91x core features for developers and users.

BLE - AE Central (SoC)
Start a BLE device in central with AE properties.

BLE - AE Peripheral (SoC)
Start a BLE device in Extended Advertising Peripheral mode.

BLE - Acceptlist (SoC)

4. Create a New Project.
   - Select your target device or board.
   - Click Create New Project.
   - In the Examples view, search for SSI.



SiWG917 Single Band Wi-Fi and BLE 8MB Flash RB, Wireless Pro Kit Mainboard (ID: 0004...

Run a pre-compiled demo or create a new project based on a software example.

SL Si91x - SSI Master
Demonstrates how to to use SSI Master.

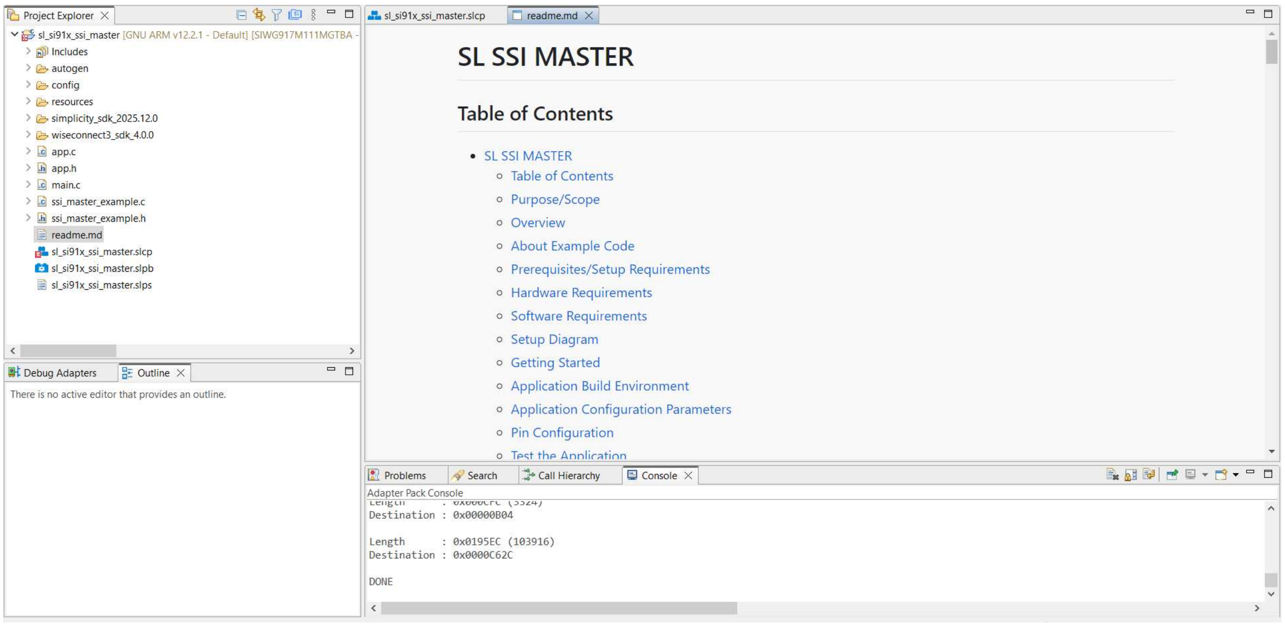
SL Si91x - SSI Slave
Demonstrates how to to use SSI Slave.

SL Si91x - ULP SSI MASTER
Demonstrates how to use ULP SSI Master.

Wi-Fi - TWT TCP Client (SoC)
Demonstrates the procedure to setup TWT session and open a TCP client socket and sends data to TCP server socket in Low Power mode

   - Choose a demonstration project (for example, SSI Primary, SSI Secondary, or ULP SSI Primary) from the WiSeConnect SDK.
   - Click Create to add the project to your workspace.
   -

Review the included readme.md for project purpose and usage.



There is no active editor that provides an outline.

SL SSI MASTER

Table of Contents

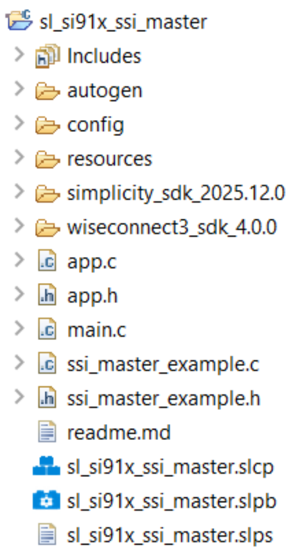
- SL SSI MASTER
  - Table of Contents
  - Purpose/Scope
  - Overview
  - About Example Code
  - Prerequisites/Setup Requirements
  - Hardware Requirements
  - Software Requirements
  - Setup Diagram
  - Getting Started
  - Application Build Environment
  - Application Configuration Parameters
  - Pin Configuration
  - Test the Application

```
Length      : 0x000CFC (3324)
Destination : 0x00000B04

Length      : 0x0195EC (103916)
Destination : 0x0000C62C

DONE
```

5. Build and Flash
   - Compile the project and flash it to your board.

> Reference: For complete setup details, see the SSI Primary Example Project on GitHub.

SSI Master Example – Folder Structure



Folder/File	Description
`autogen/`	Auto-generated configuration files (headers, linker scripts).
`config/`	Platform-specific configuration headers.
`resources/`	Documentation images and supporting resources.
`simplicity_sdk/`	Gecko SDK platform layer and third-party libraries.
`wiseconnect3_sdk_4.0.0/`	WiSeConnect SDK components and drivers.

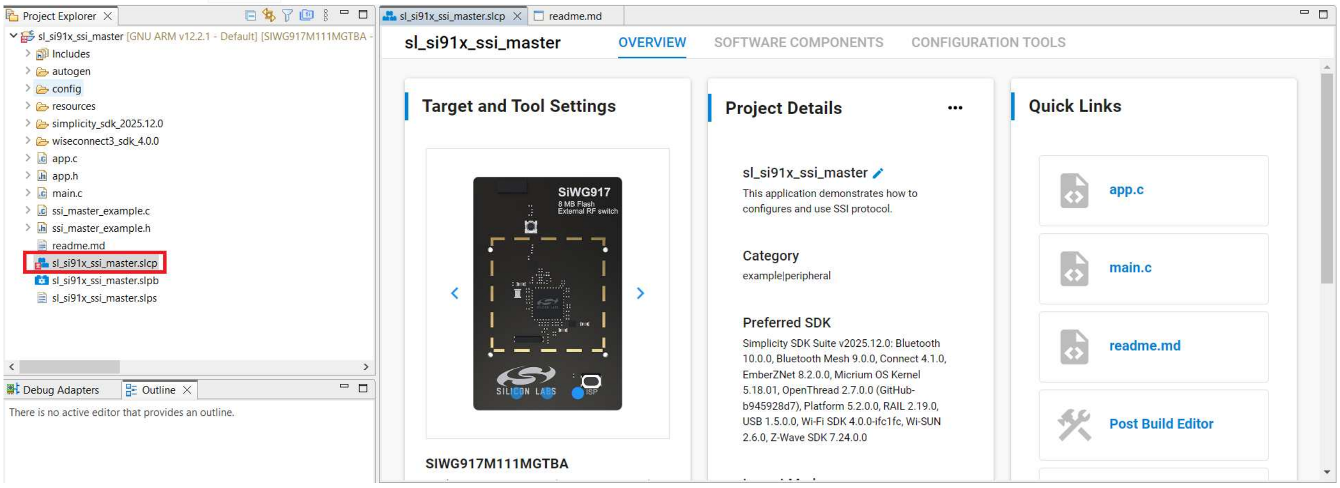
Folder/File	Description
`app.c` / `app.h`	Main application source and header files.
`main.c`	Application entry point.
`sl_si91x_ssi.slcp`	Main project configuration file for Simplicity Studio.
`sl_si91x_ssi.slps`	Project set file for managing multiple `.slcp` projects.
`sl_si91x_ssi.pintool`	Pin Tool configuration file for SSI signal mapping.
`readme.md`	Documentation and example usage guide.

Configuring SSI in Simplicity Studio

The SSI component is accessible in the Simplicity Studio Project Configurator (Component Editor). Follow the steps below to add and configure it.

Add the SSI Component

1. Open your project's `.slcp` file in Simplicity Studio.



sl_si91x_ssi_master

This application demonstrates how to configures and use SSI protocol.

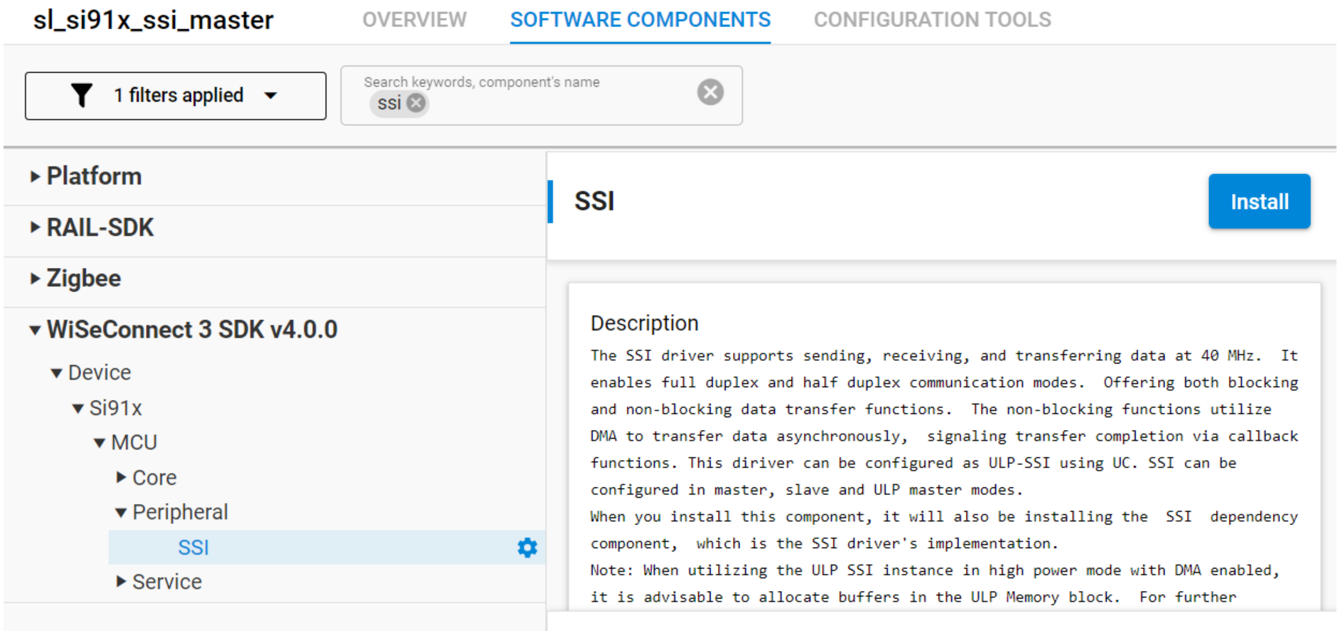
Category

example|peripheral

Preferred SDK

Simplicity SDK Suite v2025.12.0: Bluetooth 10.0.0, Bluetooth Mesh 9.0.0, Connect 4.1.0, EmberZNet 8.2.0.0, Micrium OS Kernel 5.18.01, OpenThread 2.7.0.0 (GitHub-b945928d7), Platform 5.2.0.0, RAIL 2.19.0, USB 1.5.0.0, Wi-Fi SDK 4.0.0-ifc1fc, Wi-SUN 2.6.0, Z-Wave SDK 7.24.0.0

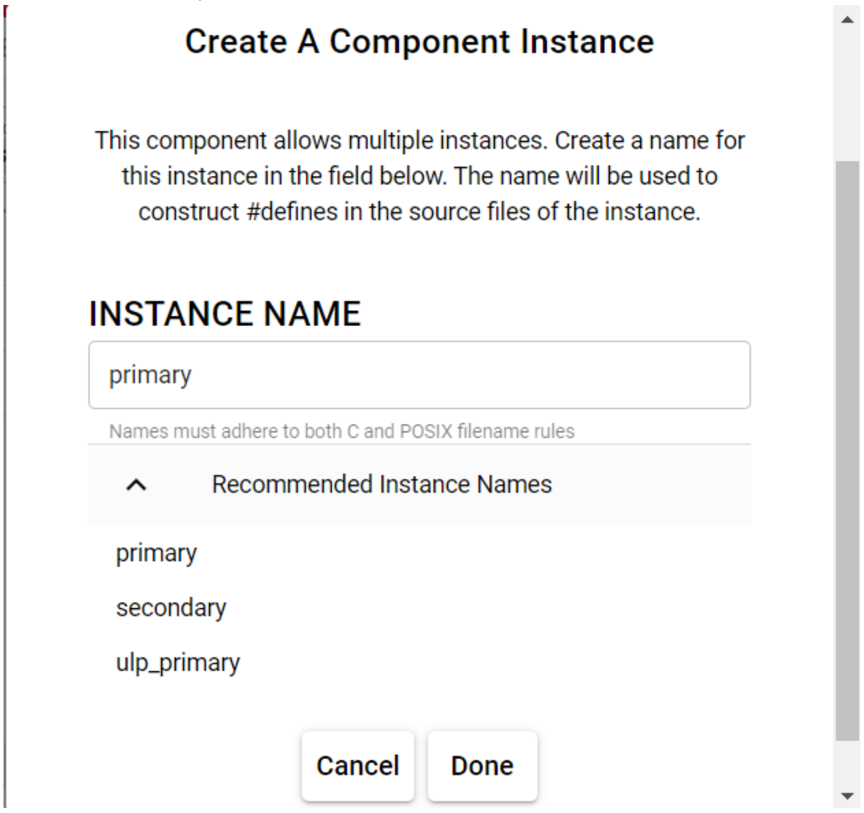
2. Go to the Software Components tab.
3. Search for SSI or navigate to WiSeConnect 3 SDK → Device → Si91x → MCU → Peripheral.



The SSI driver supports sending, receiving, and transferring data at 40 MHz. It enables full duplex and half duplex communication modes. Offering both blocking and non-blocking data transfer functions. The non-blocking functions utilize DMA to transfer data asynchronously, signaling transfer completion via callback functions. This diriver can be configured as ULP-SSI using UC. SSI can be configured in master, slave and ULP master modes.
When you install this component, it will also be installing the SSI dependency component, which is the SSI driver's implementation.
Note: When utilizing the ULP SSI instance in high power mode with DMA enabled, it is advisable to allocate buffers in the ULP Memory block. For further

4.

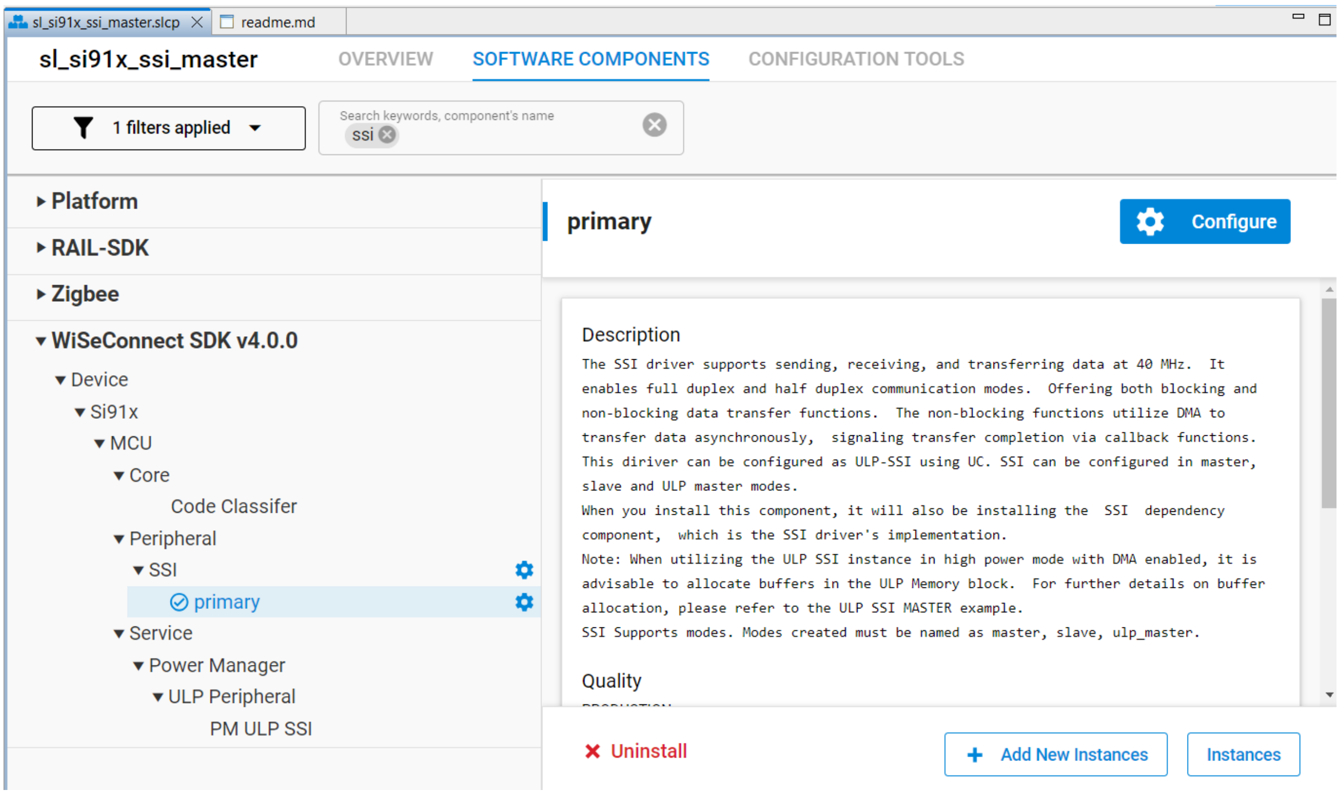
Select the SSI component. If it's not installed, click Install.


Create A Component Instance
This component allows multiple instances. Create a name for this instance in the field below. The name will be used to construct #defines in the source files of the instance.
INSTANCE NAME
primary
Names must adhere to both C and POSIX filename rules
Recommended Instance Names
primary
secondary
ulp_primary
Cancel
Done

5. Follow the prompts to install and confirm the configuration.

Configure SSI Parameters

1. Click Configure next to the SSI component to open the configuration UI.

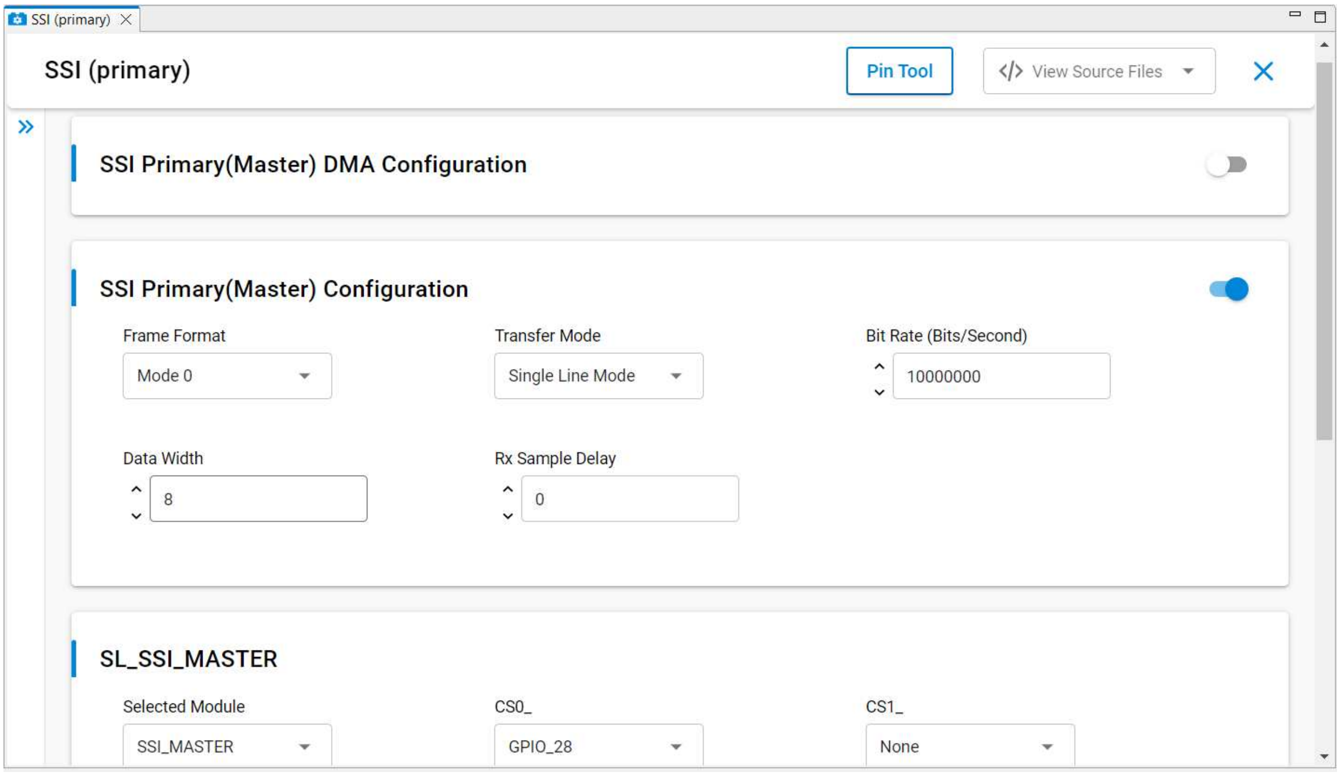


Description

```
The SSI driver supports sending, receiving, and transferring data at 40 MHz.  It
enables full duplex and half duplex communication modes.  Offering both blocking and
non-blocking data transfer functions.  The non-blocking functions utilize DMA to
transfer data asynchronously,  signaling transfer completion via callback functions.
This diriver can be configured as ULP-SSI using UC. SSI can be configured in master,
slave and ULP master modes.
When you install this component, it will also be installing the  SSI  dependency
component,  which is the SSI driver's implementation.
Note: When utilizing the ULP SSI instance in high power mode with DMA enabled, it is
advisable to allocate buffers in the ULP Memory block.  For further details on buffer
allocation, please refer to the ULP SSI MASTER example.
SSI Supports modes. Modes created must be named as master, slave, ulp_master.
```

2.

Adjust parameters such as device mode, bit rate, data width, and pin assignments.



SSI (primary)

SSI Primary(Master) DMA Configuration

SSI Primary(Master) Configuration

Frame Format: Mode 0
Transfer Mode: Single Line Mode
Bit Rate (Bits/Second): 10000000
Data Width: 8
Rx Sample Delay: 0

SL_SSI_MASTER

Selected Module: SSI_MASTER
CS0_: GPIO_28
CS1_: None

Alternatively, edit the generated configuration headers manually:

- `sl_si91x_ssi_primary_config.h`
- `sl_si91x_ssi_primary_common_config.h`

SSI Configuration Parameters

Parameter	Description	Options / Range
Device Mode	Operating role selection	Primary, Secondary, ULP Primary
Enable SSI	Enables/disables SSI instance	0 (Disabled), 1 (Enabled)
DMA Enable	Enables DMA for SSI transfers	On / Off
Frame Format	Protocol and CPOL/CPHA selection	SPI Mode 0–3, TI SSP, Microwire
Transfer Mode	Number of data lines used	Single (Standard), Dual, Quad
Bit Rate	Serial clock frequency (Hz)	500,000–40,000,000
Data Width	Number of bits per frame	4–16
Receive Sample Delay	Delay (in cycles) for sampling received data	0–63
Chip Select Outputs	Number of chip-select outputs (Primary only)	1–4

> Notes:
>
> - Match frame format and transfer mode to the connected device’s requirements.
> - Dual and quad modes support SPI Modes 0–3 only.
> - Maximum bit rates: HP SSI Primary up to 40 MHz, SSI Secondary up to 20 MHz, ULP SSI Primary up to 10 MHz.
> - Tune the Receive Sample Delay to optimize timing for high-speed operation.
> - For Quad SPI mode, define `SPI_QUAD_MODE=1` in preprocessor macros.

SSI Pin Annotation

The SSI peripheral supports multiple GPIO pin mappings, providing flexibility for hardware design.

SSI Primary – Pin Options

Signal	GPIO Options
SSI_CLK	GPIO_8, GPIO_25, GPIO_52
SSI_CS0	GPIO_9, GPIO_28, GPIO_53
SSI_CS1	GPIO_10
SSI_CS2	GPIO_15, GPIO_50
SSI_CS3	GPIO_51
SSI_MOSI/DATA0	GPIO_11, GPIO_26, GPIO_56
SSI_MISO/DATA1	GPIO_12, GPIO_27, GPIO_57
SSI_DATA2	GPIO_6, GPIO_29, GPIO_54
SSI_DATA3	GPIO_7, GPIO_30, GPIO_55

SSI Secondary – Pin Options

Signal	GPIO Options
SSI_CLK	GPIO_8, GPIO_26, GPIO_47, GPIO_52
SSI_CS	GPIO_9, GPIO_25, GPIO_46, GPIO_53
SSI_MISO	GPIO_11, GPIO_28, GPIO_49, GPIO_57
SSI_MOSI	GPIO_10, GPIO_27, GPIO_48, GPIO_56

ULP SSI (Low-Power) – Pin Options

Signal	ULP GPIO Options	SoC GPIO Options
ULP_SSI_CLK	ULP_GPIO_0, ULP_GPIO_4, ULP_GPIO_8	GPIO_6, GPIO_46
ULP_SSI_CS0	ULP_GPIO_7, ULP_GPIO_10	GPIO_48
ULP_SSI_CS1	ULP_GPIO_4	GPIO_10
ULP_SSI_CS2	ULP_GPIO_6	GPIO_12
ULP_SSI_MISO	ULP_GPIO_2, ULP_GPIO_6, ULP_GPIO_9	GPIO_8, GPIO_47
ULP_SSI_MOSI	ULP_GPIO_1, ULP_GPIO_5, ULP_GPIO_11	GPIO_7, GPIO_49

> Notes:
>
> - Map ULP SSI to either ULP GPIOs or standard SoC GPIOs depending on power design.
> - The ULP SSI instance supports only single-bit SPI mode.
> - Avoid pin conflicts between SSI instances or other peripherals.
> - If Quad SPI mode is enabled, ensure DATA2 and DATA3 are configured.
> - SSI_DATA0 = MOSI (output with respect to Primary), DATA1 = MISO (input with respect to Primary).
> - Misconfigured DATA2/DATA3 pins may trigger build warnings.

For complete pin multiplexing details, refer to the SiWx917 Device Data Sheet (PDF).

Configuration Parameters

The following structure defines user-configurable parameters for the SSI peripheral.
Populate this structure before calling (or indirectly through) the initialization and configuration APIs.

```
typedef struct {
    uint8_t  bit_width;            // Data frame width in bits (4–16).
    uint32_t device_mode;          // Primary / Secondary / ULP.
    uint32_t clock_mode;           // SPI mode 0–3
    uint32_t baud_rate;            // Bit rate (Hz).
    uint32_t receive_sample_delay; // RX sample delay (0–63 cycles).
    uint32_t transfer_mode;        // Single / Dual / Quad.
} sl_ssi_control_config_t;
```

For detailed descriptions of `sl_ssi_control_config_t` and the latest parameter documentation, refer to the API reference:

- sl_ssi_control_config_t structure reference

API to Initialize

The SSI peripheral initialization follows a standard sequence:

```
#include "sl_si91x_ssi.h"
// Configure SSI parameters
sl_ssi_control_config_t ssi_primary_config = {
    .bit_width            = 8,
    .device_mode          = SL_SSI_MASTER_ACTIVE,
    .clock_mode           = SL_SSI_PERIPHERAL_CPOL0_CPHA0,
    .baud_rate            = 1000000,      // 10 MHz
    .receive_sample_delay = 0,
    .transfer_mode        = SL_SSI_PRIMARY_TRANSFER_MODE
};

// Step 1: Initialize SSI instance
sl_ssi_handle_t ssi_handle;
sl_si91x_ssi_init(ssi_primary_config.device_mode, &ssi_handle);

// Step 2: Set slave number
sl_si91x_ssi_set_slave_number(SSI_SLAVE_0);

// Step 3: Configure SSI settings
sl_si91x_ssi_set_configuration(ssi_handle, &ssi_primary_config, SSI_SLAVE_0);

// Step 3: Register callback (optional)
sl_si91x_ssi_register_event_callback(ssi_handle, ssi_callback_function);

// Step 4: Begin data transfers
uint8_t tx_data[10] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A};
uint8_t rx_data[10] = {0};

sl_si91x_ssi_send_data(ssi_handle, tx_data, sizeof(tx_data));
sl_si91x_ssi_receive_data(ssi_handle, rx_data, sizeof(rx_data));
```

```
Tip: Always verify the return values of SSI API calls ( sl_status_t ) to ensure successful
initialization and configuration.

For non-blocking communication, register event callbacks to handle transfer completion
and error events efficiently.
```

API Reference

Synchronous Serial Interface (SSI) API Reference

The SiWx917 (Si91x family) Synchronous Serial Interface (SSI) API in the WiSeConnect SDK provides the modules, data types (enumerations/typedefs), configuration structures, functions, and macros needed to configure and use SSI. It enables applications to operate in primary or secondary roles, select clock settings and data formats, and perform transfers using blocking, interrupt-driven, or DMA.

Public APIs

This section provides an overview of the public Synchronous Serial Interface (SSI) Application Programming Interfaces (APIs) for the SiWx917 platform. It describes the available modules, enumerations, typedefs, configuration structures, functions, and macros. It also includes usage patterns and concise examples for typical operations, such as configuring primary or secondary modes, selecting clock and data formats, and performing data transfers using interrupts or Direct Memory Access (DMA).

For the complete SSI API documentation, see the official guide: SiWx917 SSI API Reference.

For examples and configuration details, see the following resources:

- SSI Master Example
- SSI Slave Example
- SSI Ultra-Low-Power (ULP) Master Example

Callback and Notification APIs

The callback and notification APIs provide efficient, event-driven handling of SSI events. By registering a callback function, the driver automatically notifies your application when events such as transfer completion or errors occur. This event-drive design improves efficiency and responsiveness by allowing your application to process other events or tasks until it’s notified of an SSI event.

For implementation details, see Registering Event Callbacks.

Integration Notes

- Always initialize the SSI interface using `sl_si91x_ssi_init()` before performing data transfers.
- Configure communication parameters using `sl_si91x_ssi_set_configuration()` prior to enabling the interface.
- Use DMA for large or continuous data transactions to improve throughput and reduce CPU overhead.
- Register callbacks for non-blocking or asynchronous transfers.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Usage Scenarios

Synchronous Serial Interface (SSI) Usage Scenarios

The Synchronous Serial Interface (SSI) on SiWx917 supports both Primary and Secondary roles, blocking and non-blocking transfers, and advanced SPI modes (Dual and Quad).

SSI integrates with Direct Memory Access (DMA) through the Universal DMA (UDMA) controller in the WiSeConnect SDK for high throughput and minimal CPU load.

This section provides practical examples, detailed workflows, and guidance to help select the appropriate transfer mode.

Usage Scenarios

Common SSI workflows include:

- Primary transfers (non-DMA): Short, simple transfers where the CPU can wait.
- Secondary transfers (non-DMA): Device responds to primary-initiated communication.
- DMA-based transfers: Efficient high-speed data movement with minimal CPU intervention.
- Dual/Quad SPI: Enhanced throughput using two or four data lines for flash or high-bandwidth peripherals.

Use Case 1: SSI Primary (Non-DMA)

Use primary mode without DMA for short or low-frequency transfers where the CPU can wait.
This configuration is ideal for sending sensor data, reading from flash, or communicating with simple peripherals.

Implementation Steps

1. Initialize the SSI driver and configure the instance.
2. Set the secondary device number (chip select).
3. Register a callback to handle completion and error events.
4. Use transmit and receive APIs for data exchange.
5. Deinitialize the driver when the transfer is complete.

1. Include Headers and Define Buffers

```c
#include "sl_si91x_ssi.h"

uint8_t tx_buffer[256] = {0x01, 0x02, 0x03, /* ... */};
uint8_t rx_buffer[256] = {0};
```

2. Define Callback and State Variables

```
static volatile bool data_received = false;
static volatile bool transfer_error = false;

static void ssi_callback(uint32_t event)
{
    if (event & SSI_EVENT_TRANSFER_COMPLETE) {
        data_received = true;
    }
    if ((event & SSI_EVENT_DATA_LOST) || (event & SSI_EVENT_MODE_FAULT)) {
        transfer_error = true;
    }
}
```

3. Initialize the SSI Primary

```
sl_ssi_handle_t ssi_handle;
sl_ssi_control_config_t ssi_primary_config = {0};
ssi_primary_config.bit_width = 8;
ssi_primary_config.device_mode = SL_SSI_MASTER_ACTIVE;
ssi_primary_config.clock_mode = SL_SSI_PERIPHERAL_CPOL0_CPHA0;
ssi_primary_config.baud_rate = 10000000;
ssi_primary_config.receive_sample_delay = 0;

sl_status_t status = sl_si91x_ssi_init(SL_SSI_MASTER_ACTIVE, &ssi_handle);
if (status != SL_STATUS_OK) {
    // Handle initialization error
}
```

4. Configure the Secondary and SSI

```
sl_si91x_ssi_set_slave_number(SSI_SLAVE_0);
status = sl_si91x_ssi_set_configuration(ssi_handle, &ssi_primary_config, SSI_SLAVE_0);
if (status != SL_STATUS_OK) {
    // Handle configuration error
}
```

5. Register the Callback

```
status = sl_si91x_ssi_register_event_callback(ssi_handle, ssi_callback);
if (status != SL_STATUS_OK) {
    // Handle callback registration error
}
```

6. Send Data

```
status = sl_si91x_ssi_send_data(ssi_handle, tx_buffer, sizeof(tx_buffer));
if (status != SL_STATUS_OK) {
    // Handle send error
}
```

7. Receive Data

```
status = sl_si91x_ssi_receive_data(ssi_handle, rx_buffer, sizeof(rx_buffer));
if (status != SL_STATUS_OK) {
    // Handle receive error
}
```

8. Deinitialize SSI

Copyright © 2025 Silicon Laboratories. All rights reserved.

```
sl_si91x_ssi_deinit(ssi_handle);
```

> Note
>
> - In non-DMA mode, for larger transfers you must manually control chip select (CS) using General-Purpose Input/Output (GPIO) APIs.
> - Use `sl_gpio_set_pin_output()` to assert CS before sending or receiving and `sl_gpio_clear_pin_output()` to deassert CS after transfer completion. This ensures correct SPI timing.

Use Case 2: SSI Secondary (Non-DMA)

In secondary mode, the device responds to a primary's communication. This flow mirrors the primary configuration but waits for the primary device to initiate data transfer.

1. Initialize the SSI Secondary

```
sl_ssi_handle_t ssi_handle;
sl_status_t status = sl_si91x_ssi_init(SL_SSI_SLAVE_ACTIVE, &ssi_handle);
if (status != SL_STATUS_OK) {
    // Handle initialization error
}
```

2. Configure the Secondary and SSI

```
sl_si91x_ssi_set_slave_number(SSI_SLAVE_0);
sl_ssi_control_config_t ssi_secondary_config = {0};
ssi_secondary_config.bit_width = 8;
ssi_secondary_config.device_mode = SL_SSI_SLAVE_ACTIVE;
ssi_secondary_config.clock_mode = SL_SSI_PERIPHERAL_CPOL0_CPHA0;
ssi_secondary_config.baud_rate = 10000000;
status = sl_si91x_ssi_set_configuration(ssi_handle, &ssi_secondary_config, SSI_SLAVE_0);
if (status != SL_STATUS_OK) {
    // Handle configuration error
}
```

3. Start Receive Operation

```
data_received = false;
transfer_error = false;
status = sl_si91x_ssi_receive_data(ssi_handle, rx_buffer, sizeof(rx_buffer));
if (status != SL_STATUS_OK) {
    // Handle receive error
}
```

Use Case 3: DMA-Based Transfers

Direct Memory Access (DMA) reduces CPU overhead during data transfers.

In the WiSeConnect SDK, DMA operations are managed by the Universal DMA (UDMA) controller, which automatically handles channel allocation and transfer completion.

SDK Layer Interaction

Copyright © 2025 Silicon Laboratories. All rights reserved.

- The SSI driver interfaces with UDMA for transmit and receive operations.
- Applications use the following high-level SSI APIs to initiate DMA transfers:
  - `sl_si91x_ssi_transfer_data()`
  - `sl_si91x_ssi_send_data()`
  - `sl_si91x_ssi_receive_data()`

DMA Event Handling

- Completion and error events are reported through the registered callback function.
- Handle the following events for reliable DMA operation:
  - `SSI_EVENT_TRANSFER_COMPLETE`
  - `SSI_EVENT_DATA_LOST`

Use Case 4: Dual/Quad SPI Advanced Transfers

Dual and Quad SPI modes enhance throughput by transmitting multiple bits per clock cycle.

These modes support blocking and DMA transfers only; interrupt-based transfers are not supported.

```
Configuration:
In Simplicity Studio’s Project Configurator (UC), enable Dual or Quad transfer mode
under the SSI component settings to activate these features.
```

Dual/Quad SPI Flash Memory Example (Blocking Mode):

```
// Configure Quad SPI: 8-bit instruction, 24-bit address, Quad frame format
sl_si91x_ssi_command_config(ssi, SSI_INST_LEN_8_BITS,
                             SSI_ADDR_LEN_24_BITS,
                             SSI_FRF_QUAD,
                             SSI_XFER_TYPE_INST_ADDR_STD);

// Write data (Quad Input Fast Program)
sl_si91x_ssi_send_command_data(ssi, tx_buf, data_length, CMD, Address);

// Read data (Dual Output Fast Read, 8 dummy cycles)
sl_si91x_ssi_receive_command_data(ssi, rx_buf, data_length, CMD, Address, waitcycle);
```

Dual/Quad SPI Flash Memory Example (DMA Mode):
To enable DMA for SSI in Simplicity Studio, set DMA Enable to On in the SSI component configuration.

```c
#include "sl_si91x_ssi.h"

static volatile bool transfer_complete = false;

// Callback for DMA completion in Dual/Quad mode
static void quad_spi_dma_callback(uint32_t event)
{
    if (event & SSI_EVENT_TRANSFER_COMPLETE) {
        transfer_complete = true;
    }
    if ((event & SSI_EVENT_DATA_LOST) || (event & SSI_EVENT_MODE_FAULT)) {
        // Error: Data lost during Dual/Quad SPI DMA transfer
    }
}

sl_status_t SSI_Transfer(void)
{
    // Step 1: Configure Quad SPI for DMA-based flash memory operation
    sl_status_t status;
    sl_ssi_handle_t ssi_handle;
    sl_ssi_control_config_t ssi_dma_config = {0};
    ssi_dma_config.bit_width            = 8;
    ssi_dma_config.device_mode          = SL_SSI_MASTER_ACTIVE;
    ssi_dma_config.clock_mode           = SL_SSI_PERIPHERAL_CPOL0_CPHA0;
    ssi_dma_config.baud_rate            = 10000000;
    ssi_dma_config.transfer_mode        = SPI_TRANSFER_MODE_QUAD;
    ssi_dma_config.receive_sample_delay = 0;

    status = sl_si91x_ssi_init(SL_SSI_MASTER_ACTIVE, &ssi_handle);
    if (status != SL_STATUS_OK) return status;

    sl_si91x_ssi_set_slave_number(SSI_SLAVE_0);

    status = sl_si91x_ssi_set_configuration(ssi_handle, &ssi_dma_config, SSI_SLAVE_0);
    if (status != SL_STATUS_OK) return status;

    status = sl_si91x_ssi_command_config(ssi_handle,
                                         SSI_INST_LEN_8_BITS,
                                         SSI_ADDR_LEN_24_BITS,
                                         SSI_FRF_QUAD,
                                         SSI_XFER_TYPE_INST_ADDR_STD);
    if (status != SL_STATUS_OK) return status;

    // Step 2: Register DMA completion callback
    status = sl_si91x_ssi_register_event_callback(ssi_handle, quad_spi_dma_callback);
    if (status != SL_STATUS_OK) return status;

    // Step 3: Prepare data buffers
    uint8_t tx_buf[256] = {/* ... */};
    uint8_t rx_buf[256] = {0};
    uint32_t data_length = sizeof(tx_buf);
    uint8_t CMD = 0x32;       // Example command
    uint32_t Address = 0x000000; // Example address
    uint8_t waitcycle = 8;    // 8 dummy cycles

    // Step 4: Start DMA-based write (Quad Input Fast Program)
    transfer_complete = false;
    status = sl_si91x_ssi_send_command_data(ssi_handle,
                                            tx_buf,
                                            data_length,
                                            CMD,
                                            Address);
    if (status != SL_STATUS_OK) return status;
    // Wait for DMA completion
    while (!transfer_complete) {
        sl_si91x_power_manager_sleep();
    }
// Configure to dual transfer mode to read data in dual mode
    status =sl_si91x_ssi_command_config(ssi_handle,
                                     SSI_INST_LEN_8_BITS,
                                     SSI_ADDR_LEN_24_BITS,
                                     SSI_FRF_DUAL,
                                     SSI_XFER_TYPE_INST_ADDR_STD);if(status != SL_STATUS_OK)return status;// Step
5: Start DMA-based read (Dual Output Fast Read)
    transfer_complete =false;
    CMD =0x6B;
    status =sl_si91x_ssi_receive_command_data(ssi_handle, rx_buf, data_length,
                                           CMD, Address, waitcycle);if(status != SL_STATUS_OK)return
status;while(!transfer_complete){sl_si91x_power_manager_sleep();}// Step 6: Process received dataprintf("Quad/Dual
SPI DMA transfer completed: %d bytes\n", data_length);// Optional: Deinitialize SSI if no longer
neededsl_si91x_ssi_deinit(ssi_handle);return SL_STATUS_OK;}
```

Example: Read Flash Status Register in Standard SPI Mode

Use `sl_si91x_ssi_receive_command_data()` to read a flash status register and check the Write In Progress (WIP) bit to verify completion of write/erase operations.

```
#define READ_STATUS_REG_CMD 0x05 // Flash status register instruction

// Assume ssi_driver_handle is initialized and configured for standard SPI

uint8_t status_reg = 0;
sl_status_t status;

// Step 1: Configure SSI for standard SPI command phase (8-bit instruction, no address)
status = sl_si91x_ssi_command_config(ssi_driver_handle,
                                     SSI_INST_LEN_8_BITS,
                                     SSI_ADDR_LEN_0_BITS,
                                     SSI_FRF_STANDARD,
                                     SSI_XFER_TYPE_INST_ADDR_STD);
if (status != SL_STATUS_OK) {
  // Handle error
  return;
}

// Step 2: Read status register (1 byte)
status = sl_si91x_ssi_receive_command_data(ssi_driver_handle,
                                           &status_reg,
                                           1,
                                           READ_STATUS_REG_CMD,
                                           0x000000,
                                           0);
if (status != SL_STATUS_OK) {
  // Handle error
  return;
}

// Step 3: Check WIP (Write In Progress) bit
if ((status_reg & 0x01) == 0) {
  // Operation complete
} else {
  // Operation still in progress, poll again if needed
}
```

Example: Send Only Command and Address (No Data Phase)

Some flash commands (for example, subsector erase or write enable) require only a command and address phase with no data phase. Use `sl_si91x_ssi_send_command_data()` as shown below.

```c
#define SUBSECTOR_ERASE_CMD 0x20 // Example: Subsector Erase command

// Assume ssi_driver_handle is initialized and configured

sl_status_t status;

// Step 1: Configure SSI for command + address phase (8-bit instruction, 24-bit address)
status = sl_si91x_ssi_command_config(ssi_driver_handle,
                                     SSI_INST_LEN_8_BITS,
                                     SSI_ADDR_LEN_24_BITS,
                                     SSI_FRF_STANDARD,
                                     SSI_XFER_TYPE_INST_ADDR_STD);
if (status != SL_STATUS_OK) {
  // Handle error
  return;
}

// Step 2: Send command and address only (no data phase)
status = sl_si91x_ssi_send_command_data(ssi_driver_handle,
                                        NULL,         // No data buffer
                                        0,            // Data length = 0
                                        SUBSECTOR_ERASE_CMD,
                                        0x000000);    // Example address
if (status != SL_STATUS_OK) {
  // Handle error
  return;
}

// The erase operation is now triggered on the flash device.
```

Advanced Features

- Dual SPI: Achieves 2× throughput using two data lines (DATA0/DATA1).
- Quad SPI: Achieves 4× throughput using four data lines (DATA0–DATA3).
- Configurable instruction and address phases: Supports both standard and enhanced transmission modes.
- Programmable wait cycles: Optimize timing to meet device-specific requirements.
- Transfer modes: Supports blocking and DMA modes for Dual/Quad SPI (interrupt mode not supported).
- Flash protocol support: Compatible with most standard SPI flash devices.

Example References

- SSI Primary Example: `sl_si91x_ssi_master`
- ULP SSI Primary Example: `sl_si91x_ulp_ssi_master`
- SSI Secondary Example: `sl_si91x_ssi_slave`

Copyright © 2025 Silicon Laboratories. All rights reserved.

Low-Power Instance

Synchronous Serial Interface (SSI) Low-Power Instance

Low-Power Instance

The low-power (ULP) instance of the Synchronous Serial Interface (SSI) provides an optimized solution for energy-constrained applications.

This capability enables serial communication during deep sleep states, helping extend battery life in portable and IoT devices.

Usage of the Low-power Instance

The Ultra-Low-Power (ULP) SSI instance allows communication during Power State 2 (PS2) with minimal energy consumption.

Battery-powered systems can maintain essential data transfers without fully waking the CPU, conserving energy and improving efficiency.

Benefits of Using ULP SSI

- Maintain communication during deep sleep modes.
- Significantly reduce overall system power consumption.
- Enable autonomous data acquisition from low-power sensors.
- Support wake-on-data or event-driven scenarios.

Configure ULP SSI for Low-power Operation (PS2)

To configure the ULP SSI instance for operation in PS2 mode:

1. Enable the ULP SSI Component
    - In Simplicity Studio, open the Software Components view.
    - Install and enable the ULP SSI component (instead of the standard SSI).
2. Configure Pin Mapping
    - Route SSI signals (SCLK, MOSI, MISO, CS) through ULP GPIOs for proper low-power operation.
3. Clock Configuration
    - Enable the ULP clock source (typically the 32 MHz RC oscillator) and gate unused clocks.
4. Power Manager Setup
    - Configure the Power Manager for PS2 mode.
    - Define the required RAM retention regions for DMA descriptors and SSI buffers.
5. Power Transition Control
    - Use `sl_si91x_power_manager_add_ps_requirement(SL_SI91X_POWER_MANAGER_PS2)` to request entry into PS2.
    - Remove the PS2 requirement when returning to active operation.
6. Peripheral Management Before Sleep
    - Disable unused peripherals.
    - Ensure all pending SSI transfers complete before entering PS2.
7. DMA Configuration (if applicable)
    - Place DMA descriptors and buffers in ULP memory to allow autonomous transfers in deep sleep.
8. Wakeup Configuration
    - Use Power Manager APIs to configure wakeup sources (GPIO, timer, wireless events, etc.).
9. Resume Operation After Wakeup
    - After exiting PS2, reinitialize and reconfigure ULP SSI as needed to resume communication.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Reinitialize After Wakeup

When the SiWx917 device exits PS2 (low-power mode), reinitialize and reconfigure the ULP SSI and GPIO peripherals to restore communication.

Steps to Reinitialize:

1. Reinitialize SSI using `sl_si91x_ssi_init()` .
2. Reconfigure ULP GPIOs for SSI signals and any active wakeup sources.
3. Reset SSI parameters—such as baud rate, frame format, FIFO thresholds, DMA, and callbacks—using `sl_si91x_ssi_set_configuration()` .
4. Resume SSI transfers or polling after verifying the first transaction.

Example – ULP SSI in Low-power Mode

```
// Init and configure (regular/ULP) SSI before entering PS2
sl_si91x_ssi_init(SL_SSI_ULP_MASTER_ACTIVE, &ssi_driver_handle);
sl_si91x_ssi_set_configuration(ssi_driver_handle,
                               &ulp_ssi_primary_config,
                               ulp_ssi_primary_slave_number);
sl_si91x_ssi_register_event_callback(ssi_driver_handle,
                                     ulp_ssi_primary_callback_event_handler);

// Before sleep
// Request transition from PS4 (active) to PS2 (deep sleep capable)
sl_si91x_power_manager_add_ps_requirement(SL_SI91X_POWER_MANAGER_PS2);

// After wakeup
// Wakeup path: reinit SSI
sl_si91x_ssi_deinit(ssi_driver_handle);
sl_si91x_ssi_init(SL_SSI_ULP_MASTER_ACTIVE, &ssi_driver_handle);
```

Implementation Example

- ULP SSI Primary Example:
  sl_si91x_ulp_ssi_master

Best Practices for Sleep/Wake Cycles

- Use ULP GPIOs for SSI signal routing whenever possible.
- Configure register retention for key control and status registers.
- Implement callback handlers to manage state transitions during sleep and wakeup.
- Validate SSI operation after wakeup before resuming communication.
- Ensure all ongoing transfers complete before entering sleep mode.

> Note: Always reinitialize the SSI peripheral after wake-up from PS2 because some configurations might not be retained in deep sleep.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Debugging and Error Handling

Synchronous Serial Interface (SSI) Debugging and Error Handling

Debugging SSI communication typically involves both hardware and software analysis.
Start by inspecting physical connections, then use debugging tools such as logic analyzers and debug logs to isolate issues.

Finally, analyze driver-level error codes to determine root causes and implement corrective actions.

Common Debugging Tips

Refer to the official Silicon Labs troubleshooting guide for application debugging:
WiSeConnect Debugging Guide

These practices provide a foundation for identifying and resolving communication issues before focusing on hardware or firmware specifics.

Visual Inspection and Connection Validation

Before running software tests, verify physical connections and signal quality.
Reliable wiring and solid PCB design are critical for stable SSI communication.

Pin Mapping

- Verify signal mappings for SCLK (CLK), CS (nCS), and data lines:
  - Single-bit mode: MOSI (IO0) and MISO (IO1)
  - Dual mode: IO0–IO1
  - Quad mode: IO0–IO3
- Confirm that pins are properly configured in the pinmux for SSI, not GPIO.

Common Reference and Voltage Levels

- Ensure a shared ground between connected boards.
- Check I/O voltage compatibility (for example, 1.8 V or 3.3 V). Use a level shifter if voltage domains differ.

Cables, Connectors, and Layout

- Inspect connectors and cables for damage or loose connections.
- Keep signal traces and wires short and direct.
- Avoid stubs or daisy-chained connections on SCLK.
- For high-speed links, add series resistors (22–49 Ω) near the driver to minimize ringing.
- Follow good PCB design practices: solid ground planes, impedance control, and minimal crosstalk near signal lines.

Power and Reset

- Confirm power rails are stable.
- Ensure any hardware reset pins are released before communication tests.

Interfacing with SiWx917 Development Kits (Primary/Secondary Setup)

You can test SSI communication between two SiWx917 development kits by configuring one as the primary and the other as the secondary.

Copyright © 2025 Silicon Laboratories. All rights reserved.

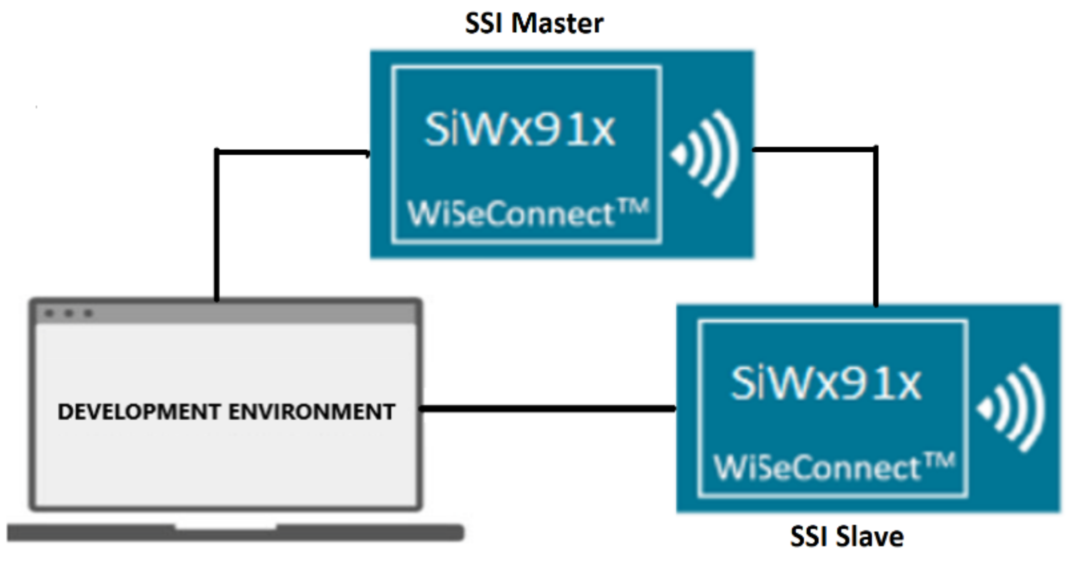
Connection Steps

1. Connect CLK, CS, MOSI, and MISO between the two boards.
2. Ensure both share a common ground.
3. Power on both boards and verify correct voltage levels.

Use Simplicity Studio to flash and debug firmware on the primary device.
Run SSI transfers and monitor activity using a logic analyzer and Simplicity Studio debug tools.

```
Note: When using custom secondary devices, ensure clock polarity (CPOL), phase (CPHA),
data width, and timing configurations match the primary’s settings.
```

Example Setup Diagram



Using a Logic Analyzer for Debugging

A logic analyzer provides protocol-level visibility for SSI signal timing and data flow.

Setup Steps

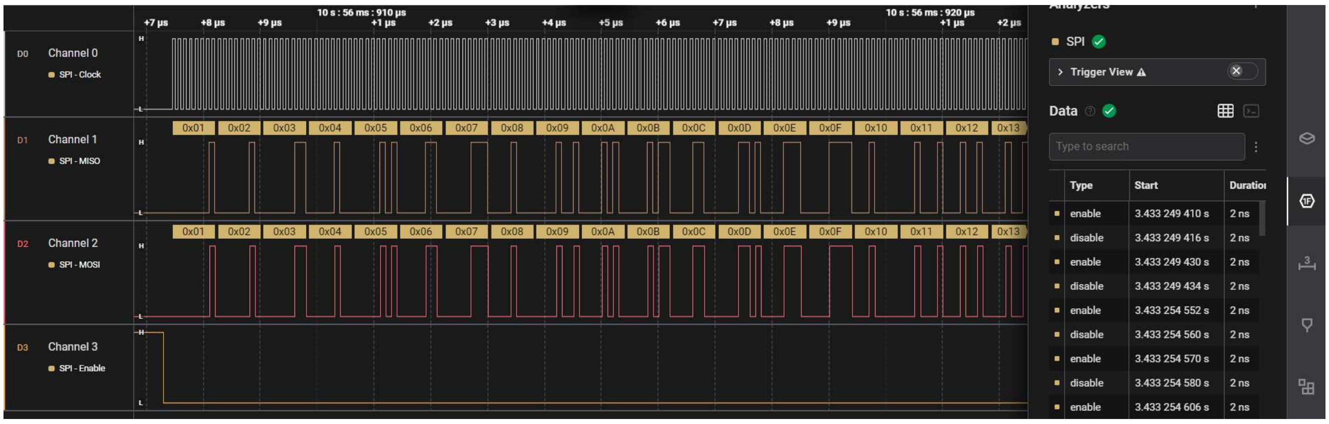
The following steps will help you capture and analyze SSI signal activity to diagnose communication issues and verify correct protocol operation.

1. Attach probes to CLK, CS, MOSI, MISO, and DATA2/DATA3 (for Dual or Quad SPI modes).
2. Confirm that the logic analyzer’s input voltage matches the board’s logic levels (for example, 3.3 V).
3. Capture SPI transactions during normal SSI operation.
4. Use the logic analyzer software’s SPI protocol decoder to analyze timing and data waveforms.

Example Capture

The following figure shows SSI communication between two SiWx917 development kits (one configured as the primary, the other as the secondary) using DMA-assisted data transfers.

Copyright © 2025 Silicon Laboratories. All rights reserved.



Type	Start	Duration
enable	3.433 249 410 s	2 ns
disable	3.433 249 416 s	2 ns
enable	3.433 249 430 s	2 ns
disable	3.433 249 434 s	2 ns
enable	3.433 254 552 s	2 ns
disable	3.433 254 560 s	2 ns
enable	3.433 254 570 s	2 ns
disable	3.433 254 580 s	2 ns
enable	3.433 254 606 s	2 ns

Software Debugging Techniques

In addition to hardware validation, software-level tools provide critical insight into SSI behavior.

Enable Debug Logging

Enable debug logging to monitor SSI operations and print error messages or status codes.

Example:

```
if (status != SL_STATUS_OK) {
    DEBUGOUT("SSI Error: 0x%02X
", status);
}
```

Error Code Handling

SSI driver APIs return standardized status codes (`sl_status_t`) that help identify communication issues and guide recovery actions.
Always check for `SL_STATUS_OK` after each API call to ensure successful operation.

Common Error Codes and Recovery Actions

Error Code	Description	Recommended Recovery Action
`SL_STATUS_OK`	Operation successful.	Continue normal operation.
`SL_STATUS_INVALID_PARAMETER`	Invalid configuration or argument.	Verify and correct parameters.
`SL_STATUS_BUSY`	Peripheral is busy.	Wait for transfer completion or reset the peripheral.
`SL_STATUS_TIMEOUT`	Operation timed out.	Check signal integrity, chip-select timing, and retry.
`SL_STATUS_TRANSMIT`	Transmission error detected.	Verify CS, SCLK, and MOSI connections.
`SL_STATUS_RECEIVE`	Reception error detected.	Check MISO signal integrity and ensure secondary readiness.
`SL_STATUS_NOT_INITIALIZED`	SSI not initialized.	Call `sl_si91x_ssi_init()` before performing transfers.

```
Tip: For persistent errors, reset and reinitialize the SSI using sl_si91x_ssi_deinit()
followed by sl_si91x_ssi_init() to restore the peripheral to a known state.
```

Interrupt Error Handling

The SiWx917 SSI generates interrupts to signal events that require software handling.

Event Type	Description	Recommended Action
Transfer Complete	Indicates that a data transfer finished successfully.	Clear interrupt flags and prepare for the next transaction.
Data Lost (Overflow/Underflow)	Signals FIFO overrun or underrun conditions.	Reset the peripheral and verify buffer handling.

```
Note: Always clear interrupt status bits after handling an event to prevent repeated
triggers.
```

Appendix

Serial Synchronous Interface (SSI) Appendix

This appendix provides reference material for the Serial Synchronous Interface (SSI) peripheral. It includes a glossary of terms, a list of acronyms, extended usage examples, and frequently asked questions (FAQs). Use this section as a quick lookup guide when developing with SSI.

Appendix Overview

The appendix is organized into the following sections:

- Glossary: Defines important SSI-related terms.
- Acronyms: Expands abbreviations commonly used in SSI development.
- Extended Examples: Provides links to example projects in the WiSeConnect software development kit (SDK).
- FAQs: Answers common questions about SSI usage, performance, and troubleshooting.

Glossary

The glossary defines key terms related to SSI and its supported protocols.

Term	Description
CMSIS	Cortex Microcontroller Software Interface Standard – ARM's standardized interface for Cortex-M microcontrollers.
CPHA	Clock Phase – Determines when data is sampled relative to the clock edge.
CPOL	Clock Polarity – Determines the idle state of the serial clock.
DMA	Direct Memory Access – Transfers data without central processing unit (CPU) intervention.
Dual SPI	Dual Serial Peripheral Interface – Uses two data lines for increased throughput.
FIFO	First In, First Out – A buffer structure used in the SSI peripheral.
HAL	Hardware Abstraction Layer – A software layer that provides a uniform interface to hardware peripherals.
Microwire	A half-duplex serial protocol developed by National Semiconductor.
PLL	Phase-Locked Loop – A control system that generates a stable, high-frequency clock from a lower-frequency reference.
Primary	The device that initiates and controls serial transfers.
PS1	Power State 1 – Ultra-low power or standby state. The CPU and most peripherals are halted. Fast wake-up is possible. Used for deep energy savings.
PS2	Power State 2 – Low performance (up to 20–32 MHz CPU, 0.75 V supply). Many features and peripherals enter low-power mode. Maintains limited processing capability while conserving power.
PS3	Power State 3 – Medium performance (up to 90 MHz CPU, 1.05 V supply). Some features and peripherals may be disabled or limited. Consumes less power than PS4 but remains responsive.
PS4	Power State 4 – Maximum performance (up to 180 MHz CPU, 1.15 V supply). All features and peripherals are available. Used for full-speed operations and during startup after reset.
Quad SPI	Quad Serial Peripheral Interface – Uses four data lines for maximum throughput.
RTOS	Real-Time Operating System – An operating system designed for real-time applications with deterministic timing.
SDK	Software Development Kit – A collection of software tools and libraries for application development.
Secondary	The device that responds to transfers initiated by the Primary.
SPI	Serial Peripheral Interface – A synchronous serial communication protocol developed by Motorola.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Term	Description
SSI	Serial Synchronous Interface – A configurable peripheral that supports multiple protocols.
SSP	Synchronous Serial Protocol – A communication protocol developed by Texas Instruments.
ULP	Ultra-Low Power – A power domain designed for minimal energy consumption.

Acronyms

Acronym	Description
APB	Advanced Peripheral Bus
CPHA	Clock Phase
CPOL	Clock Polarity
CS	Chip Select
DMA	Direct Memory Access
FIFO	First In, First Out
GPIO	General Purpose Input/Output
MISO	Master In Slave Out
MOSI	Master Out Slave In
PMU	Power Management Unit
PS1–PS4	Power Save Modes
RX	Receive
SCK	Serial Clock
SLC	Simplicity Studio Component
SPI	Serial Peripheral Interface
SSI	Serial Synchronous Interface
SSP	Synchronous Serial Protocol
TX	Transmit
UC	Universal Configurator
ULP	Ultra-Low Power

Extended Examples

This section provides links to example projects that demonstrate SSI features and integration within the WiSeConnect SDK. Use these examples to accelerate development and validate configurations.

Basic SSI Examples

- SSI Master: examples/si91x_soc/peripheral/sl_si91x_ssi_master/
- SSI Slave: examples/si91x_soc/peripheral/sl_si91x_ssi_slave/
- ULP SSI Master Example: examples/si91x_soc/peripheral/sl_si91x_ulp_ssi_master/

Sensor Integration Examples

- ICM40627 Inertial Measurement Unit (IMU) with SPI and DMA: examples/si91x_soc/peripheral/sl_si91x_icm40627/ -- Demonstrates configuring SSI for high-speed sensor data acquisition using Direct Memory Access (DMA) and burst reads.
- Memory LCD (Baremetal, SPI): examples/si91x_soc/peripheral/memlcd_baremetal/ -- Demonstrates driving a memory LCD with SSI and frame buffer updates.

FAQ

This section answers frequently asked questions about SSI functionality, modes, and troubleshooting.

Copyright © 2025 Silicon Laboratories. All rights reserved.

Q: What is the maximum Serial Peripheral Interface (SPI) clock frequency supported?

A: The SSI peripheral supports up to 40 MHz for standard SPI mode and up to 10 MHz for Dual/Quad SPI modes.

Q: Can I use both Direct Memory Access (DMA) and interrupts simultaneously?

A: Yes. When DMA is enabled, the driver uses DMA for data transfer and interrupts for completion notification and error handling.

Q: How do I switch between different SPI modes (Clock Polarity and Clock Phase)?

A: Configure the frame format in the Universal Configurator (UC) or set the appropriate parameters in the `sl_si91x_ssi_set_configuration()` function.

Q: What happens to SSI configuration during sleep modes?

A: For Ultra-Low-Power (ULP) SSI, configuration is retained during Power State 2 (PS2) and Power State 1 (PS1). For regular SSI, you must reinitialize the peripheral after waking from deep sleep.

Q: Can I communicate with multiple slave devices?

A: Yes. The SSI Primary supports up to four chip select (CS) lines for addressing multiple slave devices.

Q: How do I troubleshoot timing issues in Dual or Quad SPI?

A: Use a logic analyzer to verify clock phase, data setup and hold times, and ensure adequate wait cycles between command/address and data phases.

Q: What is the difference between blocking and non-blocking transfers?

A: Blocking transfers wait until completion before returning. Non-blocking transfers (using DMA) return immediately and use callbacks to signal completion.

Q: How do I optimize power consumption with SSI?

A: Use the ULP SSI instance with ULP General Purpose Input/Output (GPIO), enable Power State 2 (PS2), and leverage DMA for autonomous operation during sleep.

Q: Can I use SSI with external flash memory?

A: Yes. The SSI peripheral works well for flash memory communication, particularly with Dual or Quad SPI modes for higher throughput.

Q: What should I do if I encounter persistent communication errors?

A: Check hardware connections, verify clock settings, ensure proper impedance matching, and implement error recovery mechanisms as described in the debugging section.

Additional Resources

Use these resources for further reference and support:

- SSI API Documentation: WiSeConnect SSI API
- WiSeConnect SDK: WiSeConnect GitHub Repository
- Community Support: Silicon Labs Community
- Technical Support: Silicon Labs Support

Copyright © 2025 Silicon Laboratories. All rights reserved.