**Allwinner A10 SPI QEMU implementation**

I am summarizing the features of the allwinner a10 spi which is very similar to the the one that we need to implement. The needed files from qemu are: hw/ssi/ssi.h, hw/ssi/ssi.c, and fifo8.c and fifo8.h. For our board we will need to include instead of fifo8 fifo32 (later it is explained).  
I am adding another word file to explain in details what ssi.c and ssi.h are doing.

**Essentially, it allows QEMU devices to:**

* Create a virtual serial bus.
* Attach multiple peripheral (slave) devices.
* Handle synchronous serial data transfers with proper chip-select handling.

1. **Full-duplex synchronous serial interface**

Implementation:

* + The SPI implementation utilizes the QEMU SSI bus, which inherently supports synchronous full-duplex transfers.
  + allwinner\_a10\_spi\_flush\_txfifo() transmits data from the TX FIFO and simultaneously receives data from connected peripherals into the RX FIFO.
  + Specifically, this occurs through calls to ssi\_transfer() provided by the generic SSI framework.

*rx = ssi\_transfer(s->bus, tx);*

*fifo8\_push(&s->rx\_fifo, rx);*

Thus, data transmission (TX) and reception (RX) occur simultaneously, demonstrating a full-duplex operation.

1. **Configurable Master/Slave**

* **Implementation**:
  + This specific implementation (allwinner-a10-spi.c) acts as an **SPI Master**. It's responsible for initiating transfers and driving chip-select lines.
  + The current provided code does **not** directly support operating as an SPI slave; it's explicitly modeled as a master controller.

*s->bus = ssi\_create\_bus(dev, "spi");*

SPI Master role is established by creating an SSI bus and actively initiating transactions.

1. **Up to four chip selects**

* **Implementation**:
  + Supported through the defined constant (AW\_A10\_SPI\_CS\_LINES\_NR), indicating 4 chip select lines.

*#define AW\_A10\_SPI\_CS\_LINES\_NR (4)*

* + The controller manages chip select lines explicitly:

*for (i = 0; i < AW\_A10\_SPI\_CS\_LINES\_NR; i++) {*

*sysbus\_init\_irq(SYS\_BUS\_DEVICE(dev), &s->cs\_lines[i]);*

*}*

* + Active chip select lines are chosen via control bits (SPI\_CTL\_SS) in the SPI\_CTL\_REG.

1. **8x64 FIFO for both transmit and receive data**

* **Implementation**:
  + Both TX and RX FIFOs are implemented explicitly as Fifo8 buffers of size AW\_A10\_SPI\_FIFO\_SIZE, which is set to 64 bytes each.

*fifo8\_create(&s->tx\_fifo, AW\_A10\_SPI\_FIFO\_SIZE);*

*fifo8\_create(&s->rx\_fifo, AW\_A10\_SPI\_FIFO\_SIZE);*

* + Data written to the TX register is queued into the TX FIFO, and received data is queued in the RX FIFO for reading by software.
  + FIFO management functions (fifo8\_push, fifo8\_pop) manage buffered data storage and retrieval.

1. **Configurable Polarity and phase of the Chip Select (SPI\_SS) and SPI Clock (SPI\_SCLK)**

* **Implementation**:
  + Clock polarity (CPOL) and phase (CPHA) configurations are provided through dedicated bits in the control register (SPI\_CTL\_REG):

*#define SPI\_CTL\_POL (1 << 3) // Clock polarity configuration bit*

*#define SPI\_CTL\_PHA (1 << 2) // Clock phase configuration bit*

* + Chip Select polarity (SPI\_CTL\_SSPOL) is also configurable:

*#define SPI\_CTL\_SSPOL (1 << 4) // Chip select polarity bit*

* + Thus, the SPI controller provides flexibility to adjust CPOL, CPHA, and chip select polarity per application requirements.

Implement an **SPI peripheral for the NXP S32K3x8** in QEMU, inspired by the provided Allwinner A10 SPI code.

A screenshot of a computer program

AI-generated content may be incorrect.

**Analysis of Requested Features**

**Features Supported (or easily adaptable from existing Allwinner SPI):**

* **Configurable clock polarity and phase**  
  Already supported by Allwinner (SPI\_CTL\_POL, SPI\_CTL\_PHA).
* **Multiple peripheral chip selects**  
  Your peripheral requires up to **8 chip selects**, the Allwinner supports **4**. This requires increasing cs\_lines in your implementation.
* **Full-duplex transfers**  
  Fully supported in the existing Allwinner code via QEMU’s SSI bus.
* **FIFO buffers (Transmit and Receive)**  
  Allwinner already supports **64-byte FIFOs** (fifo8\_create). You need smaller FIFOs (4-word transmit & command FIFO, 4-word receive FIFO), which will simplify the FIFO implementation.
* **DMA request support**  
  DMA registers are acknowledged but unimplemented in the Allwinner code. You'd need to explicitly implement DMA interaction logic for your peripheral.

**Features Partially Supported (requiring modifications or enhancements):**

* **Flexible timing parameters** (SCK frequency, duty cycle, delays between PCS and SCK)  
  Timing and delays aren't explicitly handled in the provided code. You would need to extend your peripheral to explicitly support these timing/delay parameters. QEMU's timer utilities (QEMUTimer) could help implement realistic timing behavior.
* **Continuous transfer option (PCS asserted across multiple frames)**  
  This would require a modification to handle chip-select assertion across multiple transactions (instead of resetting chip select after each SPI transfer).

**New or unsupported features (not present in Allwinner code, requiring new logic):**

* **32-bit word size support**  
  The provided implementation is strictly byte-oriented (8-bit). You’ll need to extend the FIFO implementation to support configurable word sizes (up to 32-bit), adjusting data transfer logic accordingly.
* **Peripheral (Slave) mode**  
  The Allwinner implementation is master-only. To support peripheral (slave) mode, you must implement additional logic allowing the peripheral to respond to incoming transfers from another master.
* **Half-duplex transfer with variable bit-width transfers (1-bit, 2-bit, 4-bit, 8-bit)**  
  This requires adding significant logic to handle half-duplex modes and different bit-width transmissions. The existing implementation handles full-duplex 8-bit transfers exclusively.

**Recommended Approach to Develop Your NXP S32K3x8 SPI Peripheral:**

To efficiently develop your new peripheral using the provided Allwinner A10 SPI as a template, follow these structured steps:

**Step 1: Adapt the Core Peripheral Structure**

* Copy the Allwinner peripheral implementation as a template (e.g., nxp\_s32k\_spi.c and .h).
* Rename structures and constants (e.g., AWA10SPIState → NXPS32KSPIState).
* Modify register definitions to match your NXP S32K SPI peripheral documentation.

**Step 2: Expand Chip Select Lines**

* Change chip select count from 4 to 8:

**Step 3: FIFO Customization**

* Reduce FIFO size to match your peripheral requirements (4-word transmit/command FIFO, 4-word receive FIFO). For a 32-bit word size, each FIFO entry should be 4 bytes:

fifo32\_create(&s->tx\_fifo, 4); // 4-word FIFO

fifo32\_create(&s->rx\_fifo, 4);

* Create FIFO operations supporting 32-bit words rather than bytes, ensuring consistency with the hardware specification.

**FIFO32 is already present in qemu**

**Step 4: Add 32-bit Data Support**

* Adjust register reads and writes to handle 32-bit transfers, changing the peripheral’s behavior from byte-wise transfers (fifo8\_push/pop) to word-wise (fifo32\_push/pop).
* Modify the SSI transfer calls to handle 32-bit data:

**Step 5: Implement DMA Support**

* Clearly define DMA registers according to the peripheral’s manual.
* Handle DMA transfers by signaling DMA requests via QEMU’s DMA abstraction or interrupts, depending on your device documentation.

**Step 6: Flexible Timing and Continuous Transfers**

* Implement timers using QEMUTimer to model flexible SPI timing (SCK frequency, duty cycle, PCS delays).
* Manage continuous chip select assertion across multiple transfers by tracking state explicitly (e.g., pcs\_continuous flag).

**Step 7: Peripheral (Slave) Mode Support**

* Introduce a new mode to toggle between master and peripheral (slave) mode.
* When in slave mode, the peripheral must wait passively for incoming transfers from a connected master device.

**Step 8: Half-Duplex and Variable Bit-width Support**

* Implement state machines to handle half-duplex transfers and configurable bit widths per transfer (1-bit, 2-bit, 4-bit, 8-bit).
* Introduce additional registers for configuring transfer modes and bit widths.
* Clearly handle each transfer mode with conditional logic in the data transfer function.