Morse Micro
reaching farther™

MM6108 SDIO/SPI

Design & Troubleshooting Guide

APPNOTE-15

Table of Contents

1. Overview

The purpose of this guide is to assist in the design and troubleshooting the bring up of an MM6108 module interfaced via SDIO or SPI on Linux-based host systems. The guide addresses the key requirements for selecting parts in a design, and outlines how to diagnose issues as well as expected behaviour. With these in hand, it should be possible to quickly resolve issues in new designs.

2. SPI

A SPI interface provides a simple and common interface to a host processor. Compared to SDIO, it has fewer signals and is unidirectional instead of bidirectional. A SPI interface is specified at a physical layer and as such, the data transfer is less standardised than SDIO. Below are some key considerations from a hardware and software perspective.

2.1 Hardware

When selecting a CPU host to interface via SPI to the MM6108, consider the following requirements:

- The host should have a SPI compliant controller with support for SPI mode 0.
- The host must support level-triggered interrupts.
- The host must support full-duplex mode, as this is required by MM6108 for the SPI interface.
- The SPI interface can run at up to 50 MHz clock rate, per the specification.
- The host must support DMA backed transactions if full throughput is required on the SPI bus. Standard SPI can achieve up to 25Mbps at 50MHz, but this will reduce significantly if there is no DMA support. For example, an SPI interface with an 8-byte buffer per transaction might only achieve 2Mbps throughput on the SPI bus.
- The PCB trace length should be kept as short as possible, as extra delay can be caused which will result in lower throughput on the SPI bus.
- The MM6108 launches data on the negative edge. It is preferable to have a SPI host-side that launches on the negative edge, and samples on the positive edge in order to allow for the necessary delays.

The following pins are required for SPI:

Pin	Name	SPI mode function
31	CLK	Clock pin (input)
30	MOSI	Master data out/slave data in
29	MISO	Master data in/slave data out
28	EXT_HOST_SEL	SDIO/SPI/QSPI interface enable strap (tie high)

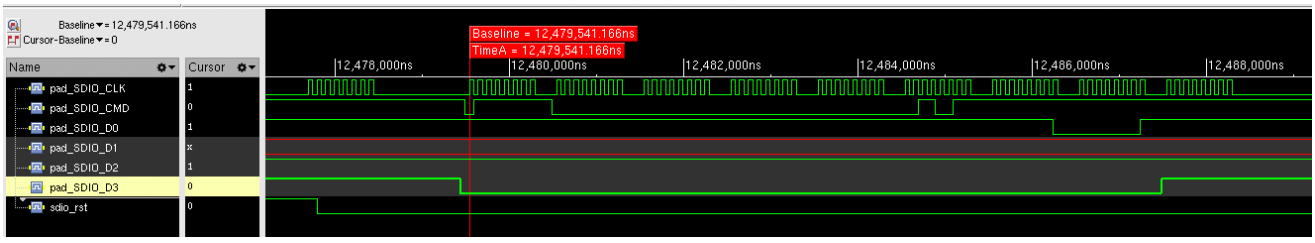
27	INT	Interrupt pin
26	Unused	Tie High
25	CS	Chip select (active low)

Note: if the waveform shape is abnormal or the timing is irregular, ensure that nothing other than the MM6108 chip and the host CPU is connected on the lines to prevent unintended interference.

2.2 Software

2.2.1 Linux-based systems

- The MM6108 SPI interface adheres to the definition in the SDIO 2.0 specification
- The implementation departs from the SDIO specification in two ways:
    1. At least 8 clocks with CS deasserted and MOSI held high immediately after reset are required to take the SPI device hardware out of reset and allow it to power up internally.
    2. CMD63 can be sent to skip the entire initialization routine of the SDIO specification. This is implemented by the Morse Micro host-side driver, so the host system does not need to handle it. The chip will return an all (8) zeros response to CMD63 indicating success of initialisation.
- CS needs to be stable after being asserted, or it will prematurely finish the transaction
- The MOSI line should NOT be toggled while waiting for a command response as this may interrupt the transaction
- Sample waveform:



- Need to ensure just one byte of clocks is sent and then immediately deassert the MOSI and then start the command transaction
- SPI mode - data clocked in on the rising edge, shifted out on the falling edge (ie SPI Mode 0)

Note: MOSI state doesn't matter if there is no clock running

2.2.2 FreeRTOS/Microcontroller-based systems

This is quite similar to the above outlined for Linux, but with a couple of differences:

1. Hard reset the device
2. Transmit at least 74 SPI clock pulses (we do this by writing `0xff` with CS deasserted 16 times)
3. Write CMD63
4. If CMD63 was not successful, write CMD0 and go to step 3 above.
5. Read chip ID to validate success.

3. SDIO

The SDIO protocol is a higher level protocol compared to SPI. It has a more well-defined specification that leads to less variance in implementation. As a result it requires less effort debugging of differences between the master and slave devices, and a faster turnaround to do the initial board bringup. It requires more signal connections which are mainly bi-directional. Below are some key considerations from a hardware and software perspective.

3.1 Hardware

- Ensure that the SDIO data and command lines are pulled up by an external pull-up resistor. The expected range of the pull-up resistance should be between 10k and 100k ohms as per the SDIO Physical Specification (section 6.6.4), with 10k providing the strongest pull-up.
- The voltage of the SDIO line should be 3.3V±0.3 ; if the voltage is too low, ensure that no internal pull-up resistors have been enabled on the host.
- It is advisable to stress test the bus (eg iperf test) and check the rise and fall time of the SDIO lines to ensure no oddities are observed.

3.1 Software (Linux)

- Check the toolchain/build environment to see if the MMC/SDIO/SDHCI interface is enabled
- Please check the MMC/SDIO/SDHCI interface that the MM6108 module is connected to is enabled in the device tree. For example:

```
&sdhci {
    status = "okay";
```

- Check that MM6108 is added to the device tree; either as an overlay or added to the DTS of the device. Templates for our DTS are provided in the porting guide and need to be modified for the target host system
- The DTS configuration can be included as a fragment if the host system supports it. For example, on the Raspberry Pi:

```
/dts-v1/;
/plugin/;

/ {
    compatible = "brcm,bcm2835";

    fragment@0 {
      target = <&mmc>;
      wifi_ovl: __overlay__ {
            pinctrl-0 = <&sdio_ovl_pins &mm_sdio_pins>;
            pinctrl-names = "default";
            non-removable;
            bus-width = <4>;
            status = "okay";
            #address-cells = <1>;
            #size-cells = <0>;
            mm6108_sdio: mm6108_sdio@0 {
                  compatible = "morse,mm610x";
                  reset-gpios = <&gpio 5 0>; /
                  power-gpios = <&gpio 3 0>, // WAKE
                        <&gpio 7 0>; // BUSY or GPIO0
                  status = "okay";
                  reg = <2>;
                  bus-width = <4>;
            };
      };
    };

    fragment@1 {
      target = <&gpio>;
      __overlay__ {
            sdio_ovl_pins: sdio_ovl_pins {
                  brcm,pins = <22 23 24 25 26 27>;
                  brcm,function = <7>; /* ALT3 = SD1 */
                  brcm,pull = <0 2 2 2 2 2>;
            };
            mm_sdio_pins: mm_sdio_pins {
                  brcm,pins = <1>; /* CHIP-IRQ */
```

```
                brcm,function = <0>;
                brcm,pull = <2>;
            };
        };
    };
};
```

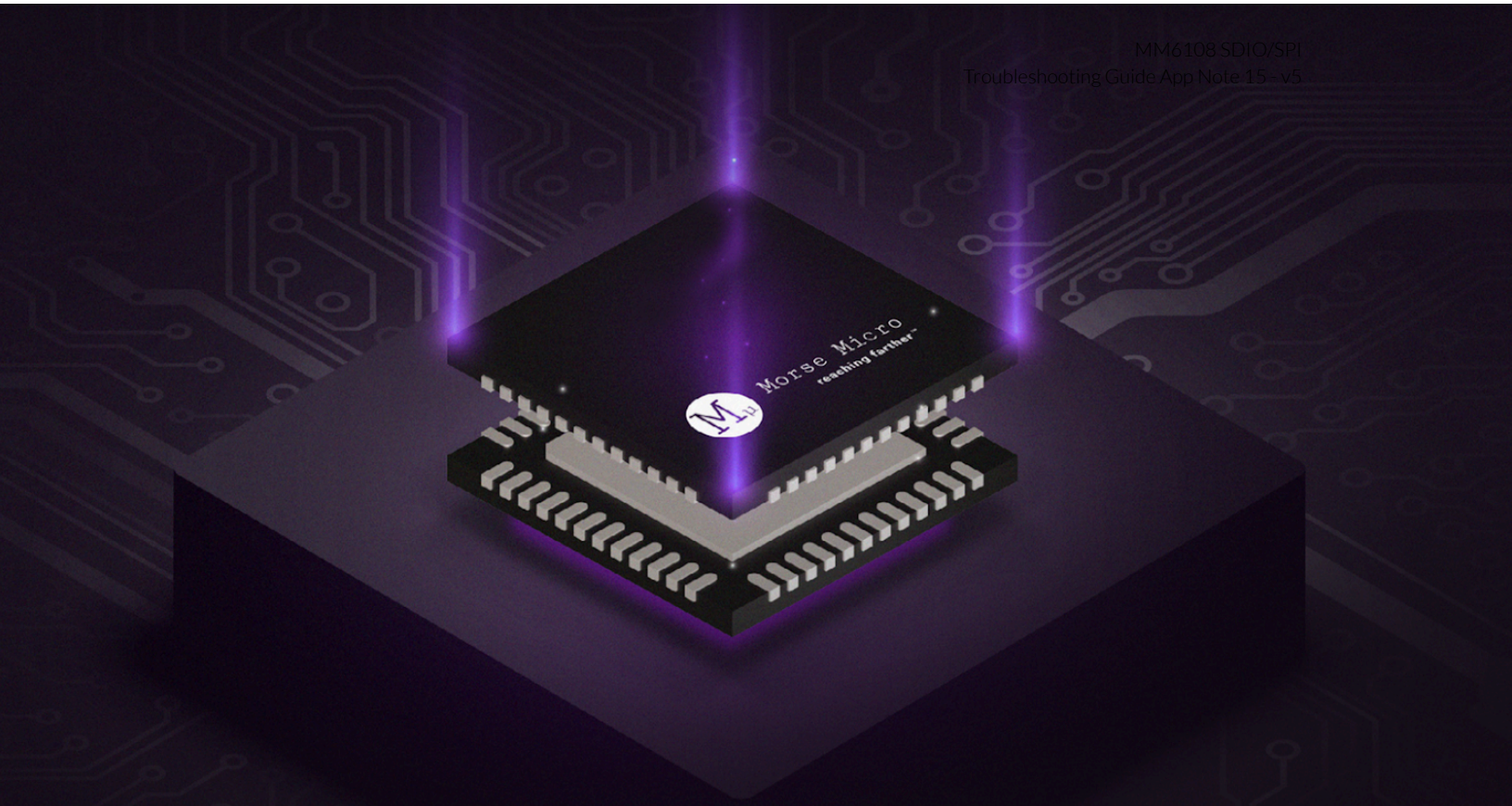
- The module can also be integrated into the device tree directly without the use of overlays. For example, the below integration with a Mediatek host:

```
&sdhci {
     status = "okay";
     mediatek,cd-poll;

     mm6108_sdio: mm6108_sdio@0 {
                compatible = "morse,mm610x";
                reset-gpios = <&gpio 40 0>;
                power-gpios = <&gpio 39 0>,
                          <&gpio 41 0>;
           status = "okay";
                reg = <2>;
                bus-width = <4>;
     };
};
```

4. Revision History

Release Number	Release Date	Release Notes
1	22/11/2022	Initial release
2	27/04/2023	Added section on FreeRTOS debugging
3	22/09/2023	Revised pin assignments in code example
4	03/11/2023	General revision and updates - Doc Control
5	28/11/2024	Make Pin26 explicitly tied high



Morse Micro
reaching farther™