Linux as an SPI Slave

Adding SPI slave support to Linux

Geert Uytterhoeven
geert@linux-m68k.org

Glider bvba

FOSDEM 2018 / Hardware Enablement Devroom

© Copyright 2018 Glider bvba, CC-BY-SA-4.0

Glider.be

Table of Contents

Introduction

SPI Bus Explained

SPI Slave Challenges & Solutions

SPI Slave Prototype & Implementation

Final Words

About Me (and Computers)

Hobbyist

- 1985 Commodore 64
- 1988 Commodore Amiga 500
- 1994 Linux/m68k on Amiga
- 1997 Linux/PPC on CHRP
- 1997 Linux FBDev

Sony

- 2006 Linux on PS3/Cell

Glider bvba

- 2013 Renesas ARM-based SoCs

About Me (and FOSDEM)

2000 OSDEM
2001 FOSDEM
2002 FOSDEM
2002 FOSDEM
2003 FOSDEM
2004 FOSDEM, Embedded Track Program Committee
...
2018 FOSDEM, **Linux as an SPI Slave**

Linux as an SPI Slave

3 years ago (v3.19), Linux got I^2C Slave support

Can Linux be used as an SPI Slave, too?

Yes, but . . .

Serial Peripheral Interface

Master

MOSI
MISO
SCK
CS1
CS2
CS3

Slave 1

MOSI
MISO
SCK
CS

Slave 2

MOSI
MISO
SCK
CS

Slave 3

MOSI
MISO
SCK
CS

- Glorified shift register
- Single master is in control, multiple slaves
- Simultaneous TX and RX, high speeds (up to tens of MHz)

Serial Peripheral Interface

Example

MOSI
MISO
SCLK
nCS
4SPI 0
S
Command: 0xE9
P
10.0 µs

SPI Transfer

SPI Options

- Clock phase/polarity (`MODE_[0...3]`)
- Chip select polarity
- LSB or MSB first
- Bits per word
- Maximum transfer speed

SPI Variants

SPI Simplifications

- MOSI only
- MISO only
- 3-Wire: shared MOSI/MISO
- No chip select

SPI Extensions

- Dual: Combine MOSI & MISO
- Quad: Add 2 more wires
- DDR
- Paired QSPI

SPI Protocol (Slave Specific)

- Message consists of one or more transfers*
- Chip Select asserted for whole message (usually)
- Different types of transfers in one message:
  - Half/Full-Duplex
  - Single/Dual/Quad
  - Dummy cycles

* $\neq$ I^2C terminology!

Serial Peripheral Interface

Example: SPI FLASH Read

SPI Slave Challenges & Limitations

- Simultaneous transmit and receive
- Master has control: Hard Real-Time
- Slave must have filled TX FIFO before master starts transfer
- Slave response cannot depend on master request in the same message without using specialized hardware
  ... or very low speeds ;-)

Story of Linux SPI Slave

Request from Renesas

- **Upstream SPI Slave Support** (cfr. I^2C)
- Initial skepticism due to challenges and limitations

What did we have?

1. Patch in Renesas R-Car BSP
2. No use case? (spidev?)
3. Renesas' customer is happy!

Let's do it!

- Come up with my own use case
- Make it fit for upstreaming

Comparison with Other Buses

Overview

1. I^2C
2. UART
3. Ethernet
4. USB
5. 1-Wire

Comparison with Other Buses: I^2C

- Two-wire bus, more complex implementation
- Multiple masters, multiple slaves
- Half-Duplex
- Low speeds (100 kbps, 400 kbps, 1 Mbps, 3.4 Mbps)
- Master is in charge
- **Slow slave can use *clock stretching***
- Controller may support both master and slave mode at the same time
- Controller may support multiple slave addresses at the same time

Comparison with Other Buses: UART

- Low speeds (up to 4 MHz)
- No concept of master/slave (anymore)
- Full-Duplex
- RX FIFO can overflow
- **Each side controls its own TX**
- **Optional hardware flow control**
  - RTS/CTS (modern, symmetrical)
  - DTR/DSR (legacy, assymmetric DCE/DTE)

Comparison with Other Buses: Ethernet

- No concept of master/slave (symmetrical)
- High speed (Gbps)
- Originally Half-Duplex
- Full-Duplex when used with network switches
- Self-clocking, preamble, postamble combining TX + SCK + CS
- **Each side controls its own TX**
- **Packets may be dropped, retransmissions handled by upper layer**

Comparison with Other Buses: USB

- High speed (Mbps–Gbps)
- **Polled bus, host initiates all transactions**
- **Transactions, packets, acknowledgements**

Comparison with Other Buses: 1-Wire

- One-wire bus, more complex implementation
- Multiple masters (in theory), multiple slaves
- Half-Duplex
- Low speeds (15–125 kbps), long range (hundreds of m)
- Master is in charge
- Optional parasite power
- ! Linux supports 1-Wire masters only!

Summary of Features to Ease Slave Support

- Slow slave can use *clock stretching* (I^2C)
- Each side controls its own TX (UART, Ethernet)
- Hardware flow control (UART)
- Packets may be dropped, retransmissions handled by upper layer (Ethernet, USB)
- Polling (USB)
- Acknowledgements (USB)

$\Rightarrow$ Useful for designing suitable SPI slave protocols later!

Designing SPI Slave Protocols

- Unidirectional: Master to Slave
  - How to know when the slave is ready to receive data?

  ⇒ Flow control (optional)

- Unidirectional: Slave to Master
  - How to know when the slave has data to send?

  ⇒ Polling, flow control (optional)

- Bidirectional: Combination of both
  - Slave response cannot depend on master request in the same message!

  ⇒ Reply to be sent in subsequent message

Example: nRF8001 Bluetooth Low Energy Solution

- CS replaced by two signals:

  `REQN` Master Request
  `RDYN` Slave Ready (doubles as Slave Request!)
- Master sends commands: length byte, followed by data
- Slave sends events: length byte, followed by data
- Can be separate (one length is zero), or combined!

An ACI event received from the nRF8001 processor is never a reply to a command being simultaneously transmitted. For all commands, the corresponding event will always be received in a subsequent ACI transaction.†

† Source: nRF8001 Product Specification 1.3, © Nordic Semiconductor

SPI Slave Use Cases

- Receiving streams of data in fixed-size messages (e.g. from a tuner)
- Receiving and transmitting fixed-size messages of data (e.g. network frames),
- Sending commands, and querying for responses.
- …

Finding a Use Case for a Proof-of-Concept

- I^2C slave PoC: I^2C Slave Mode EEPROM Simulator
- Can we do something similar?
- What kind of SPI slaves are supported by Linux?
  - SPI FLASH
  - Ethernet
  - DACs, sensors, ...

  ⇒ Nothing suitable for Linux SPI Slave :-(
- `spi-slave-time` for querying system uptime
- `spi-slave-system-control` for remote system state control
- `spidev` from userspace

SPI Slave Changes Dissected

Overview

1. Extend SPI BUS Device Tree Bindings
2. Extend SPI Subsystem
3. Extend Renesas MSIOF SPI Master Controller Driver
4. Sample SPI Slave Handlers

SPI Bus DT Bindings: Master Controller Example

```
spi@e6e00000 {
        compatible = "renesas,rcar-gen2-msiof";
        reg = <0 0xe6e00000 0 0x0064>;
        interrupts = <GIC_SPI 158 IRQ_TYPE_LEVEL_HIGH>;
        #address-cells = <1>;
        #size-cells = <0>;

        pmic@0 {
                compatible = "renesas,r2a11302ft";
                reg = <0>;
                spi-max-frequency = <6000000>;
        };
};
```

SPI Bus DT Bindings: Slave Controller Example

- Slave controller needs empty property **`spi-slave`**
- Slave device is represented by an optional **`slave`** subnode, specifying the slave protocol

```
spi@e6e00000 {
        compatible = "renesas,rcar-gen2-msiof";
        reg = <0 0xe6e00000 0 0x0064>;
        interrupts = <GIC_SPI 158 IRQ_TYPE_LEVEL_HIGH>;

        spi-slave;

        slave { /* Optional */
                compatible = "spi-slave-time";
        }
};
```

SPI Slave Changes Dissected: SPI Subsystem

- New Kconfig symbol `CONFIG_SPI_SLAVE`
- DT parsing updates
- New `spi_slave` device class
- A mechanism to associate SPI slave handlers with an SPI slave controller:
  1. DT compatible value,
  2. `/sys/devices/.../CTLR/slave`
- New API:
  - **`spi_alloc_slave()`**
  - **`spi_slave_abort()`**
  - **`spi_controller_is_slave()`**
- Generalize SPI `master` to `controller` (+ backwards compatibility)

SPI Master Controller API

```
#include <linux/spi/spi.h>

static int probe(struct device *dev)
{
        struct spi_master *master;
        int ret;

        master = spi_alloc_master(dev, ...);

        /* Fill in capabilities */
        master->... = ...;

        /* Fill in callbacks */
        master->setup = ...;
        master->transfer_one = ...;

        ret = devm_spi_register_master(dev, master);
        if (ret < 0)
                spi_master_put(master);

        return ret;
}
```

SPI Slave Controller API

```
#include <linux/spi/spi.h>

static int probe(struct device *dev)
{
        struct spi_controller *ctlr;
        int ret;

        ctlr = spi_alloc_slave(dev, ...);

        /* Fill in capabilities */
        ctlr->... = ...;

        /* Fill in callbacks */
        ctlr->setup = ...;
        ctlr->transfer_one = ...;
        ctlr->slave_abort = ...;

        ret = devm_spi_register_controller(dev, ctlr);
        if (ret < 0)
                spi_controller_put(ctlr);

        return ret;
}
```

SPI Slave Changes Dissected: Renesas MSIOF

Hardware Configuration

- Do not generate clock signal
- SCK and CS become input

⇒ Flip a few bits in the SPI controller's registers

Software Layer on Top

- Register either as a master or slave, based on DT
- Replace `wait_for_completion_timeout()` by `wait_for_completion_interruptible()`
- Allow abort from the `.slave_abort()` callback
- Limitation: message size must be known in advance

SPI Slave Changes Dissected: SPI Slave Handlers

SPI Slave Handler vs. SPI Slave Driver

- **SPI Slave Driver** talks to SPI slave via SPI master controller
- **SPI Slave Handler** listens to remote SPI master via SPI slave controller NEW

SPI Slave Handler Implementation

- Most of the existing infrastructure is reused
- SPI slave controller looks almost like an ordinary SPI master controller, same API:
  - Transfer request will block on the remote SPI master
  - Transfer can be cancelled using `spi_slave_abort()`
- RDY-signal not included, but may be implemented on top (GPIO on slave tied to GPIO or IRQ on master)

SPI Slave Driver API

```
#include <linux/spi/spi.h>

static int probe(struct spi_device *spi)
{
        /* Optional configuration */
        spi->... = ...;
        spi_setup(spi);

        /* SPI transfers */
        spi_write(spi, buf, len);
        spi_read(spi, buf, len);
        spi_w8r8(spi, cmd);
        ...
        spi_sync_transfer(spi, xfers, n_xfers);
        ...
        spi_message_init_with_transfers(msg, xfers, n_xfers);
        spi_sync(spi, &msg);
        spi_async(spi, &msg);
        ...
}
```

SPI Slave Handler API

```
static int probe(struct spi_device *spi)
{
        /* Optional configuration */
        spi->... = ...;
        spi_setup(spi);

        /* Non-blocking SPI transfers */
        spi_message_init_with_transfers(msg, xfers, n_xfers);
        spi_async(spi, &msg);
        ...
}
/* Optional thread */
static void thread(struct kthread_work *work)
{
        /* Blocking SPI transfers */
        spi_write(spi, buf, len);
        ...
}
static int remove(struct spi_device *spi)
{
        spi_slave_abort(spi);
        wait_for_completion(...);
}
```

Example 1: Querying System Uptime

- Enable the `spi-slave-timer` handler:

```
# echo spi-slave-time > /sys/class/spi_slave/spi3/slave
```

- Send 8 dummy bytes and receive responce:

```
# spidev_test -D /dev/spidev2.0 -p dummy-8B
spi mode: 0x0
bits per word: 8
max speed: 500000 Hz (500 KHz)
RX | 00 00 04 6D 00 09 5B BB ...
```

- Response is uptime when **previous** message was received!
- Or all zeroes/ones when the remote system has died

Example 2: Remote System Control

- Enable the `spi-slave-system-control` handler:

```
# echo spi-slave-system-control > /sys/class/spi_slave/spi3/slave
```

- Send command:

```
# reboot='\x7c\x50'
# poweroff='\x71\x3f'
# halt='\x38\x76'
# suspend='\x1b\x1b'
# spidev_test -D /dev/spidev2.0 -p $suspend
```

Example 3: Passing Fixed Size Messages

- Enable the `spidev` handler:

```
# echo spidev > /sys/class/spi_slave/spi3/slave
```

- Transfer data:

```
# spidev_test -D /dev/spidev3.0 -p slave-hello-to-master &
# spidev_test -D /dev/spidev2.0 -p master-hello-to-slave
...
RX | 6D 61 73 74 65 72 2D 68 65 6C 6C 6F 2D 74
        6F 2D 73 6C 61 76 65 ... | master-hello-to-slave
...
RX | 73 6C 61 76 65 2D 68 65 6C 6C 6F 2D 74 6F
        2D 6D 61 73 74 65 72 ... | slave-hello-to-master
...
```

http://elinux.org/Tests:MSIOF-SPI-Slave

Mark Brown's pull request

```
spi: Updates for v4.13

There's only one big change in this release but it's a very
big change, Geert Uytterhoeven has implemented support for
SPI slave mode.  This feature has been on the cards since
the subsystem was originally merged back in the mists of
time so it's great that Geert stepped up and finally
implemented it.

 - SPI slave support, together with wholesale renaming of
   SPI controllers from master to controller which went
   surprisingly smoothly.  This is already used with Renesas
   SoCs and support is in the works for i.MX too.
 - New drivers for Meson SPICC and ST STM32
```

Future Work

- SPI Slave support for more SPI controllers
  Currently limited to Renesas MSIOF, Freescale i.MX SPI
- MSIOF: Support for variable length messages
- More SPI Slave Handlers
- IP-over-SPI
- ...

Thanks & Acknowledgements

- **Renesas Electronics Corporation**, for contracting me for upstream Linux kernel work,
- **Hisashi Nakamura (@Renesas)**, for the initial SPI slave driver implementation,
- **FOSDEM and its volunteer team**, for organizing this conference and giving me the opportunity to present here,
- The **Linux Kernel Community**, for having so much fun working together towards a common goal.