CONFIDENTIAL

TC | APPLIED TECHNOLOGIES

How to create a high speed SPI slave handler.

File:	SPI Slave Handler.doc
Printed:	3/20/09 11:22:00 AM
Updated:	3/18/09 4:49:00 PM

Revision history

When	Who	What
2008-06-21	M Lave	Document Started
2009-03-18	B Karr	Added 2.3 regarding using SPI slave with MixFix
2009-03-20	B Karr	Various edits

Table of Contents

1 Introduction

This document describes how to make an SPI slave handler which can respond to high speed SPI.

The hardware SPI module is basically a shift register and one Rx and Tx holding register. For an SPI slave the firmware must be able to empty the Rx holding register within one word interval and provide new data for the Tx as well. If the SPI interface is running at 1MHz and the word size is 8 and assuming there is one extra clock period in each end of the transfer this will result in the handler routine having to respond within 10us.

The eCos kernel usually handles the IRQ and it does disable the IRQ when in critical regions so it is not possible to guarantee IRQ latencies under 100us.

Further more the ARM is normally executing code out of SDRAM and as there is no cache in the DICE system code would normally be executed at rates which are ¼ of the clock speed of 50MHz.

This document will show how to change the interrupt mux to make sure the SPI is routed to the FIQ. How to install lthe handler and some other hints on how to make the load on the system minimal.

2 Using the FIQ

The interrupt mux registers are explained in the General Purpose Control and Status register section of the users guide. We will just program the FIQ mux to route the SPI interrupt to the first FIQ line.

```
//Prepare the FIQ handler
static void spiFIQInitialize (void)
{
    //we need to set-up the FIQ mux for SPI int.
    *((volatile uint32*)GPCSR_FIQ_MUX_5_TO_0) = 0x0; //FIQ0=SPI_INT
    //let's mask the FIQ0 so we can use the mask bit to turn this
      //functionality off and on
    *((volatile uint32*)ICTL_FIQ_MASK) = 0x1;
    //now it is safe to install the handler, it will not run now
    TCInterruptCreateFIQ(&spi_fiq_handler);
}
```

You would need to include these header files for the various definitions"

```
#include "TCTypes.h"
#include "TCTasking.h"
#include "TCInterrupt.h"
#include "coreDefs.h"
```

The interrupt is enabled but still masked. As the SPI hardware has functions to mask the individual interrupts we might just unmask the FIQ right away.

```
*((volatile uint32*)ICTL_FIQ_MASK) = 0x0;
```

We also need to make sure the default SPI firmware handler for master mode is not installed. Make sure that 'spi' does not appear in any of the module definitions in Make.params:

```
BASE_MODULE := main target cli debug
APP_MODULE := 1394lal misc fis frmwload
CHIP_MODULE := dice 1394llc ~~spi~~ sram
```

2.1 The FIQ handler

The FIQ handler is called with all state saved on the FIQ stack. This stack is 128 DWORDS deep and by the time the handler is called 9 DWORDS has been used. By default the stack is in SDRAM but it can be moved to internal SRAM for optimizations by modifying the kernel files.

```
static void spi_fiq_handler(void) __attribute__((section
(".sram.spi_isr")));

static void spi_fiq_handler(void)
{
      //make sure the interrupt condition is cleared.
      //in case of Tx int, write something to the SPI DATA port
      //In case of Rx int read from the SPI DATA port
}
```

Above we saw how to install the handler but we need to enable the interrupts we want to get through in the SPI module itself.

In order to work with the SPI module we can steal the defines from spiDefs.h. This file is in the spi/src directory which is part of the master handler so simply copy this file to a directory of your choice or just copy the content into your implementation file. We will be using those defines below:

Let's assume that we are using clkphase=1 and clkpl=1 as that is by far the most common:

Set the format, slave, word size etc:

```
MPTR(SPI_CTRL) = 0x34;
MPTR(SPI_INT_MASK) = 0x03;
//right here we will get our first Tx Empty interrupt
```

So what should the handler do? It depends on the application. We could just implement a simple circular buffer for receive and transmit. We would do that in internal SRAM as well so things are efficient:

```
typedef struct
{
      uint8 rxbuf[256];
      uint8 rxPut;
      uint8 rxGet; //convention put==get => empty
      uint8 txbuf[256];
      uint8 txPut;
      uint8 txGet; //convention put==get => empty
      uint8 rxOverrun;
} SPI_SLAVE_DATA;

static volatile SPI_SLAVE_DATA      spiData     __attribute__((section
(".sram.spi_data")));
```

If we are writing C we need to declare these volatile to make sure the order of operations is kept by the optimizer.

The handler could then look like this:

```
static void spi_fiq_handler(void) __attribute__((section
(".sram.spi_isr")));

static void spi_fiq_handler(void)
{
      uint32 stat = MPTR(SPI_STAT);
      if (stat & SPI_RX_FULL)
      {
            //we got data, read it to clear Int
            uint32 d = MPTR(SPI_DATA);
            if (spiData.rxPut+1 == spiData.rxGet)
            {
                  //no more spece in buffer
                  spiData.rxOverrun = 1;
            }
            else
            {
                  spiData.rxbuf[spiData.rxPut] = d;
                  spiData.rxPut++;
            }
      }
      if (stat & SPI_TX_EMPTY)
      {
            //tx holding register has space, put something there
            if (spiData.txPut == spiData.txGet)
            {
                  //no data to send, put a zero
                  MPTR(SPI_DATA) = 0;
            }
            else
            {
                  MPTR(SPI_DATA) = spiData.txbuf[spiData.txGet];
                  spiData.txGet++;
            }
      }
}
```

2.2 Potential pitfalls

The implementation in the previous section has some flaws. Let's say that it is okay to send '0' between packets but we can't accept zeroes in the middle of packets.

When the producer thread is putting data into the tx buffer it might not be able to keep up with the interrupt routine. As soon as the first byte is put into the Tx buffer the master might read it and continue to read faster than data can be put in. If the SPI is running fast it might consume quite a lot of CPU power not starving the producer thread.

This all depends on the application but let's look at an example:

Assume a system where all packets start with a certain character, say 0xf0 and end with another character say 0xf7. We can't allow any zeroes to be stuffed into a packet. We also assume that an 'interrupt' GPIO line is used to signal to the master that we have data.

We will keep the receiver side using a ring buffer but we will skip characters which are not inside a packet.

We change the transmit side to use a packet pointer instead.

```
typedef struct
{
      uint8 rxbuf[256];
      uint8 rxPut;
      uint8 rxGet; //convention put==get => empty
      uint8 rxskip; //initialize to 1 in init code
      uint8 txbuf[256];
      uint8 txPut;
      uint8 txGet; //convention put==get => empty
      uint8 * txPtr;
      uint8 rxOverrun;
} SPI_SLAVE_DATA;

static volatile SPI_SLAVE_DATA      spiData     __attribute__((section
(".sram.spi_data")));

static void spi_fiq_handler(void) __attribute__((section
(".sram.spi_isr")));

static void spi_fiq_handler(void)
{
      uint32 stat = MPTR(SPI_STAT);
      if (stat & SPI_RX_FULL)
      {
            //we got data, read it to clear the Int
            uint32 d = MPTR(SPI_DATA);
            if (d == 0xf0) spiData.rxskip = 0;
            if (!spiData.rxskip)
            {
                  if (spiData.rxPut+1 == spiData.rxGet)
                  {
                        //no more space in buffer
                        spiData.rxOverrun = 1;
                        spiData.rxskip = 1; //better skip rest

                  }
                  else
                  {
                        spiData.rxbuf[spiData.rxPut] = d;
                        spiData.rxPut++;
                  }
            }
            if (d == 0xf7) spiData.rxskip = 1;
      }
      if (stat & SPI_TX_EMPTY)
      {
            //tx holding register has space, put something there
            if (spiData.txPtr)
            {
                  if (*spiData.txPtr == 0xf0) GPIO=0; //start of packet
                  MPTR(SPI_DATA) = *spiData.txPtr++;
                  if (*spiData.txPtr == 0xf7)
                  {
                        spiData.txPtr = 0; //done packet
                        GPIO=1; // about to send the last data
                        }
                }
                else
                {
                        //no data to send, put a zero
                        MPTR(SPI_DATA) = 0;
                }
        }
}
```

The producer thread will have to set the txPtr to point to the beginning of a complete packet in the ring buffer. It will then wait (`TCTaskWait(1)`) in a loop and keep checking for txPtr to be 0. Then search the buffer for the next complete packet.

This is of course just an example and it is only pseudo code, it has not been compiled. Depending on the application a smarter approach can be developed from this.

2.3 DICE-Jr and DICE-Mini at 192KHz.

Finally, a note for products which use the “MixFix” workaround, which is a software patch for an errata in the internal hardware mixer at 192KHz. In this case the MixFix code also uses the FIQ, and you will need to multiplex the MixFix and SPI handling in order to use them together.

For more information about the MixFix mixer workaround, see the document ‘tcd22x0MixerBug.pdf’ in your subversion tag.

3 Closing remarks

When working with these FIQ handlers there are a few things to look out for. It is not possible to call any kernel function from the FIQ as it can interrupt at any time and the kernel might not be in a safe and unlocked place. There are ways for an FIQ to post a ‘fake’ IRQ which can then be handled by an ISR/DSR pair and signal semaphores etc.

Furthermore the kernel can be optimized for FIQ handling by putting part of the stub and the FIQ stack into SRAM. That requires some minor modifications to the kernel, specifically Vectors.s.

Another issue to watch out for is using the GPIO from the FIQ. Let’s say that there are code at thread level which is doing read-modify-write to the GPIO register. If the FIQ interrupts in the middle of such an operation the resulting state might not be correct.

In that case it might be necessary to disable FIQ around this read-modify-write.

```
#define SAFE_GPIO_OFF   0
#define SAFE_GPIO_ON    1
#define SAFE_GPIO_TGL   2

void safeSetGpio(uint8 gpio, uint8 state)
{
        uint32 msk = 1<<gpio;

        *((volatile uint32*)ICTL_FIQ_MASK) = 0x1;
        if (state==SAFE_GPIO_OFF)
                *pGpio &= ~msk;
        else if (state==SAFE_GPIO_ON)
                *pGpio |= msk;
```

```
      else if (state==SAFE_GPIO_TGL)
            *pGpio ^= msk;
      *((volatile uint32*)ICTL_FIQ_MASK) = 0x0;
}
```