

Z8530 Programming Guide

Alan Cox <alan@lxorguk.ukuu.org.uk>

Z8530 Programming Guide

by Alan Cox

Copyright © 2000 Alan Cox

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

Chapter 1. Introduction

The Z85x30 family synchronous/asynchronous controller chips are used on a large number of cheap network interface cards. The kernel provides a core interface layer that is designed to make it easy to provide WAN services using this chip.

The current driver only support synchronous operation. Merging the asynchronous driver support into this code to allow any Z85x30 device to be used as both a tty interface and as a synchronous controller is a project for Linux post the 2.4 release

Chapter 2. Driver Modes

The Z85230 driver layer can drive Z8530, Z85C30 and Z85230 devices in three different modes. Each mode can be applied to an individual channel on the chip (each chip has two channels).

The PIO synchronous mode supports the most common Z8530 wiring. Here the chip is interface to the I/O and interrupt facilities of the host machine but not to the DMA subsystem. When running PIO the Z8530 has extremely tight timing requirements. Doing high speeds, even with a Z85230 will be tricky. Typically you should expect to achieve at best 9600 baud with a Z8C530 and 64Kbits with a Z85230.

The DMA mode supports the chip when it is configured to use dual DMA channels on an ISA bus. The better cards tend to support this mode of operation for a single channel. With DMA running the Z85230 tops out when it starts to hit ISA DMA constraints at about 512Kbits. It is worth noting here that many PC machines hang or crash when the chip is driven fast enough to hold the ISA bus solid.

Transmit DMA mode uses a single DMA channel. The DMA channel is used for transmission as the transmit FIFO is smaller than the receive FIFO. it gives better performance than pure PIO mode but is nowhere near as ideal as pure DMA mode.

Chapter 4. Attaching Network Interfaces

If you wish to use the network interface facilities of the driver, then you need to attach a network device to each channel that is present and in use. In addition to use the generic HDLC you need to follow some additional plumbing rules. They may seem complex but a look at the example `hostess_sv11` driver should reassure you.

The network device used for each channel should be pointed to by the `netdevice` field of each channel. The `hdlc->priv` field of the network device points to your private data - you will need to be able to find your private data from this.

The way most drivers approach this particular problem is to create a structure holding the Z8530 device definition and put that into the private field of the network device. The network device fields of the channels then point back to the network devices.

If you wish to use the generic HDLC then you need to register the HDLC device.

Before you register your network device you will also need to provide suitable handlers for most of the network device callbacks. See the network device documentation for more details on this.

Chapter 6. Network Layer Functions

The Z8530 layer provides functions to queue packets for transmission. The driver internally buffers the frame currently being transmitted and one further frame (in order to keep back to back transmission running). Any further buffering is up to the caller.

The function `z8530_queue_xmit` takes a network buffer in `sk_buff` format and queues it for transmission. The caller must provide the entire packet with the exception of the bitstuffing and CRC. This is normally done by the caller via the generic HDLC interface layer. It returns 0 if the buffer has been queued and non zero values for queue full. If the function accepts the buffer it becomes property of the Z8530 layer and the caller should not free it.

The function `z8530_get_stats` returns a pointer to an internally maintained per interface statistics block. This provides most of the interface code needed to implement the network layer `get_stats` callback.

Chapter 7. Porting The Z8530 Driver

The Z8530 driver is written to be portable. In DMA mode it makes assumptions about the use of ISA DMA. These are probably warranted in most cases as the Z85230 in particular was designed to glue to PC type machines. The PIO mode makes no real assumptions.

Should you need to retarget the Z8530 driver to another architecture the only code that should need changing are the port I/O functions. At the moment these assume PC I/O port accesses. This may not be appropriate for all platforms. Replacing `z8530_read_port` and `z8530_write_port` is intended to be all that is required to port this driver layer.

Chapter 8. Known Bugs And Assumptions

Interrupt Locking

The locking in the driver is done via the global cli/sti lock. This makes for relatively poor SMP performance. Switching this to use a per device spin lock would probably materially improve performance.

Occasional Failures

We have reports of occasional failures when run for very long periods of time and the driver starts to receive junk frames. At the moment the cause of this is not clear.

Chapter 9. Public Functions Provided

Name

`z8530_interrupt` — Handle an interrupt from a Z8530

Synopsis

```
irqreturn_t z8530_interrupt (int irq, void * dev_id);
```

Arguments

irq Interrupt number

dev_id The Z8530 device that is interrupting.

Description

A Z85[2]30 device has stuck its hand in the air for attention. We scan both the channels on the chip for events and then call the channel specific call backs for each channel that has events. We have to use callback functions because the two channels can be in different modes.

Locking is done for the handlers. Note that locking is done at the chip level (the 5uS delay issue is per chip not per channel). `c->lock` for both channels points to `dev->lock`

Name

`z8530_sync_open` — Open a Z8530 channel for PIO

Synopsis

```
int z8530_sync_open (struct net_device * dev, struct z8530_channel * c);
```

Arguments

dev The network interface we are using

c The Z8530 channel to open in synchronous PIO mode

Description

Switch a Z8530 into synchronous mode without DMA assist. We raise the RTS/DTR and commence network operation.

Name

`z8530_sync_close` — Close a PIO Z8530 channel

Synopsis

```
int z8530_sync_close (struct net_device * dev, struct z8530_channel *  
c);
```

Arguments

dev Network device to close

c Z8530 channel to disassociate and move to idle

Description

Close down a Z8530 interface and switch its interrupt handlers to discard future events.

Name

`z8530_sync_dma_open` — Open a Z8530 for DMA I/O

Synopsis

```
int z8530_sync_dma_open (struct net_device * dev, struct z8530_channel  
* c);
```

Arguments

dev The network device to attach

c The Z8530 channel to configure in sync DMA mode.

Description

Set up a Z85x30 device for synchronous DMA in both directions. Two ISA DMA channels must be available for this to work. We assume ISA DMA driven I/O and PC limits on access.

Name

`z8530_sync_txdma_open` — Open a Z8530 for TX driven DMA

Synopsis

```
int z8530_sync_txdma_open (struct net_device * dev, struct z8530_channel  
* c);
```

Arguments

dev The network device to attach

c The Z8530 channel to configure in sync DMA mode.

Description

Set up a Z85x30 device for synchronous DMA transmission. One ISA DMA channel must be available for this to work. The receive side is run in PIO mode, but then it has the bigger FIFO.

Name

`z8530_sync_txdma_close` — Close down a TX driven DMA channel

Synopsis

```
int    z8530_sync_txdma_close    (struct net_device * dev, struct
z8530_channel * c);
```

Arguments

dev Network device to detach

c Z8530 channel to move into discard mode

Description

Shut down a DMA/PIO split mode synchronous interface. Halt the DMA, and free the buffers.

Name

`z8530_shutdown` — Shutdown a Z8530 device

Synopsis

```
int z8530_shutdown (struct z8530_dev * dev);
```

Arguments

dev The Z8530 chip to shutdown

Description

We set the interrupt handlers to silence any interrupts. We then reset the chip and wait 100uS to be sure the reset completed. Just in case the caller then tries to do stuff.

This is called without the lock held

Name

z8530_channel_load — Load channel data

Synopsis

```
int z8530_channel_load (struct z8530_channel * c, u8 * rtable);
```

Arguments

c Z8530 channel to configure

rtable table of register, value pairs

FIXME

ioctl to allow user uploaded tables

Load a Z8530 channel up from the system data. We use +16 to indicate the “prime” registers. The value 255 terminates the table.

Name

z8530_null_rx — Discard a packet

Synopsis

```
void z8530_null_rx (struct z8530_channel * c, struct sk_buff * skb);
```

Arguments

c The channel the packet arrived on

skb The buffer

Description

We point the receive handler at this function when idle. Instead of processing the frames we get to throw them away.

Name

`z8530_queue_xmit` — Queue a packet

Synopsis

```
netdev_tx_t z8530_queue_xmit (struct z8530_channel * c, struct sk_buff  
* skb);
```

Arguments

c The channel to use

skb The packet to kick down the channel

Description

Queue a packet for transmission. Because we have rather hard to hit interrupt latencies for the Z85230 per packet even in DMA mode we do the flip to DMA buffer if needed here not in the IRQ.

Called from the network code. The lock is not held at this point.

Chapter 10. Internal Functions

Name

z8530_read_port — Architecture specific interface function

Synopsis

```
int z8530_read_port (unsigned long p);
```

Arguments

p port to read

Description

Provided port access methods. The Control SV11 requires no delays between accesses and uses PC I/O. Some drivers may need a 5uS delay

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

The caller must hold sufficient locks to avoid violating the horrible 5uS delay rule.

Name

z8530_write_port — Architecture specific interface function

Synopsis

```
void z8530_write_port (unsigned long p, u8 d);
```

Arguments

p port to write

d value to write

Description

Write a value to a port with delays if need be. Note that the caller must hold locks to avoid read/writes from other contexts violating the 5uS rule

In the longer term this should become an architecture specific section so that this can become a generic driver interface for all platforms. For now we only handle PC I/O ports with or without the dread 5uS sanity delay.

Name

`read_zsreg` — Read a register from a Z85230

Synopsis

```
u8 read_zsreg (struct z8530_channel * c, u8 reg);
```

Arguments

c Z8530 channel to read from (2 per chip)

reg Register to read

FIXME

Use a spinlock.

Most of the Z8530 registers are indexed off the control registers. A read is done by writing to the control register and reading the register back. The caller must hold the lock

Name

`read_zsdata` — Read the data port of a Z8530 channel

Synopsis

```
u8 read_zsdata (struct z8530_channel * c);
```

Arguments

c The Z8530 channel to read the data port from

Description

The data port provides fast access to some things. We still have all the 5uS delays to worry about.

Name

`write_zsreg` — Write to a Z8530 channel register

Synopsis

```
void write_zsreg (struct z8530_channel * c, u8 reg, u8 val);
```

Arguments

c The Z8530 channel

reg Register number

val Value to write

Description

Write a value to an indexed register. The caller must hold the lock to honour the irritating delay rules. We know about register 0 being fast to access.

Assumes `c->lock` is held.

Name

`write_zsctrl` — Write to a Z8530 control register

Synopsis

```
void write_zsctrl (struct z8530_channel * c, u8 val);
```

Arguments

c The Z8530 channel

val Value to write

Description

Write directly to the control register on the Z8530

Name

`write_zsdata` — Write to a Z8530 control register

Synopsis

```
void write_zsdata (struct z8530_channel * c, u8 val);
```

Arguments

c The Z8530 channel

val Value to write

Description

Write directly to the data register on the Z8530

Name

z8530_flush_fifo — Flush on chip RX FIFO

Synopsis

```
void z8530_flush_fifo (struct z8530_channel * c);
```

Arguments

c Channel to flush

Description

Flush the receive FIFO. There is no specific option for this, we blindly read bytes and discard them. Reading when there is no data is harmless. The 8530 has a 4 byte FIFO, the 85230 has 8 bytes.

All locking is handled for the caller. On return data may still be present if it arrived during the flush.

Name

`z8530_rtsdtr` — Control the outgoing DTS/RTS line

Synopsis

```
void z8530_rtsdtr (struct z8530_channel * c, int set);
```

Arguments

c The Z8530 channel to control;

set 1 to set, 0 to clear

Description

Sets or clears DTR/RTS on the requested line. All locking is handled by the caller. For now we assume all boards use the actual RTS/DTR on the chip. Apparently one or two don't. We'll scream about them later.

Name

z8530_rx — Handle a PIO receive event

Synopsis

```
void z8530_rx (struct z8530_channel * c);
```

Arguments

c Z8530 channel to process

Description

Receive handler for receiving in PIO mode. This is much like the async one but not quite the same or as complex

Note

Its intended that this handler can easily be separated from the main code to run realtime. That'll be needed for some machines (eg to ever clock 64kbits on a sparc ;)).

The RT_LOCK macros don't do anything now. Keep the code covered by them as short as possible in all circumstances - clocks cost baud. The interrupt handler is assumed to be atomic w.r.t. to other code - this is true in the RT case too.

We only cover the sync cases for this. If you want 2Mbit async do it yourself but consider medical assistance first. This non DMA synchronous mode is portable code. The DMA mode assumes PCI like ISA DMA

Called with the device lock held

Name

z8530_tx — Handle a PIO transmit event

Synopsis

```
void z8530_tx (struct z8530_channel * c);
```

Arguments

c Z8530 channel to process

Description

Z8530 transmit interrupt handler for the PIO mode. The basic idea is to attempt to keep the FIFO fed. We fill as many bytes in as possible, its quite possible that we won't keep up with the data rate otherwise.

Name

`z8530_status` — Handle a PIO status exception

Synopsis

```
void z8530_status (struct z8530_channel * chan);
```

Arguments

chan Z8530 channel to process

Description

A status event occurred in PIO synchronous mode. There are several reasons the chip will bother us here. A transmit underrun means we failed to feed the chip fast enough and just broke a packet. A DCD change is a line up or down.

Name

`z8530_dma_rx` — Handle a DMA RX event

Synopsis

```
void z8530_dma_rx (struct z8530_channel * chan);
```

Arguments

chan Channel to handle

Description

Non bus mastering DMA interfaces for the Z8x30 devices. This is really pretty PC specific. The DMA mode means that most receive events are handled by the DMA hardware. We get a kick here only if a frame ended.

Name

z8530_dma_tx — Handle a DMA TX event

Synopsis

```
void z8530_dma_tx (struct z8530_channel * chan);
```

Arguments

chan The Z8530 channel to handle

Description

We have received an interrupt while doing DMA transmissions. It shouldn't happen. Scream loudly if it does.

Name

`z8530_dma_status` — Handle a DMA status exception

Synopsis

```
void z8530_dma_status (struct z8530_channel * chan);
```

Arguments

chan Z8530 channel to process

A status event occurred on the Z8530. We receive these for two reasons when in DMA mode. Firstly if we finished a packet transfer we get one and kick the next packet out. Secondly we may see a DCD change.

Name

z8530_rx_clear — Handle RX events from a stopped chip

Synopsis

```
void z8530_rx_clear (struct z8530_channel * c);
```

Arguments

c Z8530 channel to shut up

Description

Receive interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

Name

z8530_tx_clear — Handle TX events from a stopped chip

Synopsis

```
void z8530_tx_clear (struct z8530_channel * c);
```

Arguments

c Z8530 channel to shut up

Description

Transmit interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

Name

`z8530_status_clear` — Handle status events from a stopped chip

Synopsis

```
void z8530_status_clear (struct z8530_channel * chan);
```

Arguments

chan Z8530 channel to shut up

Description

Status interrupt vectors for a Z8530 that is in 'parked' mode. For machines with PCI Z85x30 cards, or level triggered interrupts (eg the MacII) we must clear the interrupt cause or die.

Name

z8530_tx_begin — Begin packet transmission

Synopsis

```
void z8530_tx_begin (struct z8530_channel * c);
```

Arguments

c The Z8530 channel to kick

Description

This is the speed sensitive side of transmission. If we are called and no buffer is being transmitted we commence the next buffer. If nothing is queued we idle the sync.

Note

We are handling this code path in the interrupt path, keep it fast or bad things will happen.

Called with the lock held.

Name

z8530_tx_done — TX complete callback

Synopsis

```
void z8530_tx_done (struct z8530_channel * c);
```

Arguments

c The channel that completed a transmit.

Description

This is called when we complete a packet send. We wake the queue, start the next packet going and then free the buffer of the existing packet. This code is fairly timing sensitive.

Called with the register lock held.

Name

z8530_rx_done — Receive completion callback

Synopsis

```
void z8530_rx_done (struct z8530_channel * c);
```

Arguments

c The channel that completed a receive

Description

A new packet is complete. Our goal here is to get back into receive mode as fast as possible. On the Z85230 we could change to using ESCC mode, but on the older chips we have no choice. We flip to the new buffer immediately in DMA mode so that the DMA of the next frame can occur while we are copying the previous buffer to an sk_buff

Called with the lock held

Name

`spans_boundary` — Check a packet can be ISA DMA'd

Synopsis

```
int spans_boundary (struct sk_buff * skb);
```

Arguments

skb The buffer to check

Description

Returns true if the buffer cross a DMA boundary on a PC. The poor thing can only DMA within a 64K block not across the edges of it.