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.

Table of Contents

1. Introduction 1
2. Driver Modes 2
3. Using the Z85230 driver 3
4. Attaching Network Interfaces 4
5. Configuring And Activating The Port 5
6. Network Layer Functions 6
7. Porting The Z8530 Driver 7
8. Known Bugs And Assumptions 8
9. Public Functions Provided 9
    - z8530_interrupt 10
    - z8530_sync_open 11
    - z8530_sync_close 12
    - z8530_sync_dma_open 13
    - z8530_sync_dma_close 14
    - z8530_sync_txdma_open 15
    - z8530_sync_txdma_close 16
    - z8530_describe 17
    - z8530_init 18
    - z8530_shutdown 19
    - z8530_channel_load 20
    - z8530_null_rx 21
    - z8530_queue_xmit 22
10. Internal Functions 23
    - z8530_read_port 24
    - z8530_write_port 25
    - read_zsreg 26
    - read_zsdata 27
    - write_zsreg 28
    - write_zsctrl 29
    - write_zsdata 30
    - z8530_flush_fifo 31
    - z8530_rtsdtr 32
    - z8530_rx 33
    - z8530_tx 34
    - z8530_status 35
    - z8530_dma_rx 36
    - z8530_dma_tx 37
    - z8530_dma_status 38
    - z8530_rx_clear 39
    - z8530_tx_clear 40
    - z8530_status_clear 41
    - z8530_tx_begin 42
    - z8530_tx_done 43
    - z8530_rx_done 44
    - spans_boundary 45

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 3. Using the Z85230 driver

The Z85230 driver provides the back end interface to your board. To configure a Z8530 interface you need to detect the board and to identify its ports and interrupt resources. It is also your problem to verify the resources are available.

Having identified the chip you need to fill in a struct z8530_dev, which describes each chip. This object must exist until you finally shutdown the board. Firstly zero the active field. This ensures nothing goes off without you intending it. The irq field should be set to the interrupt number of the chip. (Each chip has a single interrupt source rather than each channel). You are responsible for allocating the interrupt line. The interrupt handler should be set to `z8530_interrupt`. The device id should be set to the z8530_dev structure pointer. Whether the interrupt can be shared or not is board dependent, and up to you to initialise.

The structure holds two channel structures. Initialise chanA.ctrlio and chanA.dataio with the address of the control and data ports. You can or this with Z8530_PORT_SLEEP to indicate your interface needs the 5uS delay for chip settling done in software. The PORT_SLEEP option is architecture specific. Other flags may become available on future platforms, eg for MMIO. Initialise the chanA.irqs to &z8530_nop to start the chip up as disabled and discarding interrupt events. This ensures that stray interrupts will be mopped up and not hang the bus. Set chanA.dev to point to the device structure itself. The private and name field you may use as you wish. The private field is unused by the Z85230 layer. The name is used for error reporting and it may thus make sense to make it match the network name.

Repeat the same operation with the B channel if your chip has both channels wired to something useful. This isn't always the case. If it is not wired then the I/O values do not matter, but you must initialise chanB.dev.

If your board has DMA facilities then initialise the txdma and rxdma fields for the relevant channels. You must also allocate the ISA DMA channels and do any necessary board level initialisation to configure them. The low level driver will do the Z8530 and DMA controller programming but not board specific magic.

Having initialised the device you can then call `z8530_init`. This will probe the chip and reset it into a known state. An identification sequence is then run to identify the chip type. If the checks fail to pass the function returns a non zero error code. Typically this indicates that the port given is not valid. After this call the type field of the z8530_dev structure is initialised to either Z8530, Z85C30 or Z85230 according to the chip found.

Once you have called z8530_init you can also make use of the utility function `z8530_describe`. This provides a consistent reporting format for the Z8530 devices, and allows all the drivers to provide consistent reporting.

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 5. Configuring And Activating The Port

The Z85230 driver provides helper functions and tables to load the port registers on the Z8530 chips. When programming the register settings for a channel be aware that the documentation recommends initialisation orders. Strange things happen when these are not followed.

`z8530_channel_load` takes an array of pairs of initialisation values in an array of u8 type. The first value is the Z8530 register number. Add 16 to indicate the alternate register bank on the later chips. The array is terminated by a 255.

The driver provides a pair of public tables. The z8530_hdlc_kilostream table is for the UK 'Kilostream' service and also happens to cover most other end host configurations. The z8530_hdlc_kilostream_85230 table is the same configuration using the enhancements of the 85230 chip. The configuration loaded is standard NRZ encoded synchronous data with HDLC bitstuffing. All of the timing is taken from the other end of the link.

When writing your own tables be aware that the driver internally tracks register values. It may need to reload values. You should therefore be sure to set registers 1-7, 9-11, 14 and 15 in all configurations. Where the register settings depend on DMA selection the driver will update the bits itself when you open or close. Loading a new table with the interface open is not recommended.

There are three standard configurations supported by the core code. In PIO mode the interface is programmed up to use interrupt driven PIO. This places high demands on the host processor to avoid latency. The driver is written to take account of latency issues but it cannot avoid latencies caused by other drivers, notably IDE in PIO mode. Because the drivers allocate buffers you must also prevent MTU changes while the port is open.

Once the port is open it will call the rx_function of each channel whenever a completed packet arrived. This is invoked from interrupt context and passes you the channel and a network buffer (struct sk_buff) holding the data. The data includes the CRC bytes so most users will want to trim the last two bytes before processing the data. This function is very timing critical. When you wish to simply discard data the support code provides the function `z8530_null_rx` to discard the data.

To active PIO mode sending and receiving the `z8530_sync_open` is called. This expects to be passed the network device and the channel. Typically this is called from your network device open callback. On a failure a non zero error status is returned. The `z8530_sync_close` function shuts down a PIO channel. This must be done before the channel is opened again and before the driver shuts down and unloads.

The ideal mode of operation is dual channel DMA mode. Here the kernel driver will configure the board for DMA in both directions. The driver also handles ISA DMA issues such as controller programming and the memory range limit for you. This mode is activated by calling the `z8530_sync_dma_open` function. On failure a non zero error value is returned. Once this mode is activated it can be shut down by calling the `z8530_sync_dma_close`. You must call the close function matching the open mode you used.

The final supported mode uses a single DMA channel to drive the transmit side. As the Z85C30 has a larger FIFO on the receive channel this tends to increase the maximum speed a little. This is activated by calling the `z8530_sync_txdma_open` . This returns a non zero error code on failure. The `z8530_sync_txdma_close` function closes down the Z8530 interface from this mode.

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_dma_close — Close down DMA I/O

Synopsis

```
int z8530_sync_dma_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 mode synchronous interface. Halt the DMA, and free the buffers.

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_describe — Uniformly describe a Z8530 port

Synopsis

```
void z8530_describe (struct z8530_dev * dev, char * mapping, unsigned long io);
```

Arguments

dev	Z8530 device to describe
mapping	string holding mapping type (eg “I/O” or “Mem”)
io	the port value in question

Description

Describe a Z8530 in a standard format. We must pass the I/O as the port offset isn't predictable. The main reason for this function is to try and get a common format of report.

Name

z8530_init — Initialise a Z8530 device

Synopsis

```
int z8530_init (struct z8530_dev * dev);
```

Arguments

dev Z8530 device to initialise.

Description

Configure up a Z8530/Z85C30 or Z85230 chip. We check the device is present, identify the type and then program it to hopefully keep quite and behave. This matters a lot, a Z8530 in the wrong state will sometimes get into stupid modes generating 10Khz interrupt streams and the like.

We set the interrupt handler up to discard any events, in case we get them during reset or setp.

Return 0 for success, or a negative value indicating the problem in errno form.

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 Comtrol 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.