Z8530 Programming Guide

Alan Cox

alan@redhat.com

Z8530 Programming Guide

by Alan Cox

Table of Contents

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

The support code handles most common card configurations and supports running both Cisco HDLC and Synchronous PPP. With extra glue the frame relay and X.25 protocols can also be used with this driver.

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 SyncPPP and Cisco 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 dev- priv field of the network device points to your private data - you will need to be able to find your ppp device from this. In addition to use the sync ppp layer the private data must start with a void * pointer to the syncppp structures.

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

If you wish to use the synchronous ppp then you need to attach the syncppp layer to the network device. You should do this before you register the network device. The `sppp_attach` requires that the first void * pointer in your private data is pointing to an empty struct ppp_device. The function fills in the initial data for the ppp/hdlc layer.

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 syncppp 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

z8530_interrupt

Name

`z8530_interrupt` — Handle an interrupt from a Z8530

Synopsis

```
void z8530_interrupt  (int irq); void * dev_id); struct pt_regs
* regs);
```

Arguments

`irq`

Interrupt number

`dev_id`

The Z8530 device that is interrupting.

`regs`

unused

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

z8530_sync_open

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.

z8530_sync_close

Name

`z8530_sync_close` — Close a PIO Z8530 channel

Synopsis

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

Arguments