```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="Z85230Guide">
 <bookinfo>
  <title>Z8530 Programming Guide</title>

  <authorgroup>
   <author>
    <firstname>Alan</firstname>
    <surname>Cox</surname>
    <affiliation>
     <address>
      <email>alan@lxorguk.ukuu.org.uk</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2000</year>
   <holder>Alan Cox</holder>
  </copyright>

  <legalnotice>
   <para>
     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.
   </para>

   <para>
     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.
   </para>

   <para>
     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
   </para>

   <para>
     For more details see the file COPYING in the source
     distribution of Linux.
   </para>
  </legalnotice>
 </bookinfo>

<toc></toc>
```

```
<chapter id="intro">
    <title>Introduction</title>
<para>
      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.
</para>
<para>
      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
</para>
</chapter>

<chapter id="Driver_Modes">
      <title>Driver Modes</title>
<para>
      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).
</para>
<para>
      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.
</para>
<para>
      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.
</para>
<para>
      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.
</para>
</chapter>

<chapter id="Using_the_Z85230_driver">
      <title>Using the Z85230 driver</title>
<para>
      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.
</para>
```

<para>
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 <function>z8530_interrupt</function>. 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.
</para>
<para>
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.
</para>
<para>
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.
</para>
<para>
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.
</para>
<para>
Having initialised the device you can then call <function>z8530_init</function>. 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.
</para>
<para>
Once you have called z8530_init you can also make use of the utility function <function>z8530_describe</function>. This provides a consistent reporting format for the Z8530 devices, and allows all the drivers to provide consistent reporting.
</para>

```
</chapter>

<chapter id="Attaching_Network_Interfaces">
      <title>Attaching Network Interfaces</title>
<para>
      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.
</para>
<para>
      The network device used for each channel should be pointed to by
      the netdevice field of each channel. The hdlc-&gt; priv field of the
      network device points to your private data - you will need to be
      able to find your private data from this.
</para>
<para>
      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.
</para>
<para>
      If you wish to use the generic HDLC then you need to register
      the HDLC device.
</para>
<para>
      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.
</para>
</chapter>

<chapter id="Configuring_And_Activating_The_Port">
      <title>Configuring And Activating The Port</title>
<para>
      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.
</para>
<para>
      <function>z8530_channel_load</function> 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.
</para>
<para>
      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.
</para>
<para>
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.
</para>
<para>
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.
</para>
<para>
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 <function>z8530_null_rx</function>
to discard the data.
</para>
<para>
To active PIO mode sending and receiving the <function>
z8530_sync_open</function> 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 <function>z8530_sync_close</function>
function shuts down a PIO channel. This must be done before the
channel is opened again and before the driver shuts down
and unloads.
</para>
<para>
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 <function>z8530_sync_dma_open</function>
function. On failure a non zero error value is returned.
Once this mode is activated it can be shut down by calling the
<function>z8530_sync_dma_close</function>. You must call the close
function matching the open mode you used.
</para>
<para>
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 <function>z8530_sync_txdma_open
</function>. This returns a non zero error code on failure. The
<function>z8530_sync_txdma_close</function> function closes down
the Z8530 interface from this mode.
</para>
</chapter>

<chapter id="Network_Layer_Functions">
<title>Network Layer Functions</title>
<para>
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.
</para>
<para>
The function <function>z8530_queue_xmit</function> 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.
</para>
<para>
The function <function>z8530_get_stats</function> 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.
</para>
</chapter>

<chapter id="Porting_The_Z8530_Driver">
<title>Porting The Z8530 Driver</title>
<para>
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.
</para>
<para>
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
<function>z8530_read_port</function> and <function>z8530_write_port
</function> is intended to be all that is required to port this
driver layer.
</para>
</chapter>

<chapter id="bugs">
<title>Known Bugs And Assumptions</title>

```
  <para>
  <variablelist>
    <varlistentry><term>Interrupt Locking</term>
    <listitem>
    <para>
        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.
    </para>
    </listitem></varlistentry>

    <varlistentry><term>Occasional Failures</term>
    <listitem>
    <para>
        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.
    </para>
    </listitem></varlistentry>
  </variablelist>

  </para>
  </chapter>

  <chapter id="pubfunctions">
     <title>Public Functions Provided</title>
!Edrivers/net/wan/z85230.c
  </chapter>

  <chapter id="intfunctions">
     <title>Internal Functions</title>
!Idrivers/net/wan/z85230.c
  </chapter>

</book>
```