ZyDAS ZD 1211 Solaris driver

Contents

1. Prologue 1
2. USB client API 1
3. GLDv3 API 2
   3.1 MAC module interface 2
   3.2 The WiFi stack (module net80211) 3
4. Device description 3
   4.1 Overview 3
   4.2 I/O Registers 3
   4.3 USB endpoints 4
   4.4 Control pipe 4
   4.5 Sending frames 5
   4.6 Receiving frames 6
   4.7 Sending commands 7
   4.8 Receiving command results and host status 8
5. Source code layout 9
6. TODO list 10
7. Resources 10

1. Prologue

This file was written as documentation for the ZyDAS ZD 1211 Solaris driver. Any information presented here is without warranty of any kind. You may use this document in any way you want as long as you keep our names on it.

2. USB client API

This is a short glimpse on USBA, the Solaris kernel USB interface. USBA has been documented extensively by Sun Microsystems (see [1] for example).

As the reader is probably aware, USB devices listen for communications on different endpoints (analogous to TCP/IP ports). To communicate with an endpoint, a pipe (connection) to the endpoint must be opened. A special case is the default control pipe, that is automatically opened by the host controller. All other pipes must be opened manually by calling `usb_pipe_open(9F)`.

The device endpoints make up an interface to the device. But a USB device can be a complex beast. It can provide multiple interfaces (each with its own set of endpoints) that expose different (and possibly independent) functionality. Moreover, each interface can be switched into different alternate configurations (having different sets of endpoints). Finally, the device as a whole may be switched into different configurations, completely changing the set of available interfaces.

The list of all configurations, interfaces, their alternate configurations and the corresponding sets of endpoints are stored in the endpoint descriptor tree. The leaves of

this tree contain sets of endpoint descriptors, which describe the endpoints address, type etc. Endpoint descriptors are key, because they are passed to `usb_pipe_open(9F)` to identify which endpoint should the function connect to.

The tree is obtained by calling `usb_get_dev_data(9F)`. The helper function `usb_lookup_alt_if(9F)` can be used to extract a specific endpoint descriptor from the tree and `usb_free_descr_tree(9F)` frees the tree from memory.

Once the pipes have been opened, transfers are issued to communicate with the device. Since multiple transfers may take place at the same time, transfer request structures are used to allow asynchronous operation. A transfer is executed as follows:

- allocate request
- submit the request using `usb_pipe_ctrl_xfer(9F)`, `usb_pipe_bulk_xfer(9F)` or `usb_pipe_intr_xfer(9F)` function (depending on endpoint type)
- wait for transfer completion
- free the request structure

Alternatively, the transfers may be executed synchronously by passing an `USB_FLAGS_SLEEP` flag to the transfer function. A synchronous call blocks until the transfer has been completed. Asynchronous calls do not block, however, they require two callback functions to be registered within the request structure. One callback is used for regular transfer termination and the other is used for exception handling.

3. GLDv3 API

The Generic LAN Driver is an OS layer that is common to all ethernet-like network devices. We must implement its interface so the system can use our driver. There are two modules that we use: The MAC layer and the WiFi stack (net80211 module). Both modules are poorly documented because they are still under development. We obtained most information we have from existing WiFi card drivers. Therefore the following information might be inaccurate or incomplete.

3.1 MAC module interface

This module defines set of callback functions which we are supposed to implement. These functions provides only the most general operations such as:

- retrieving statistic information
- starting/stopping the device
- switching the device into promiscuous mode
- enabling/disabling multicast address
- setting the MAC address (for unicast)
- transmitting a packet
- processing IOCTL operations

The Wifi plugin of the MAC layer (mac_wifi) handles encapsulation of transmitted packets (adding the IEEE 802.11 header).

3.2 The WiFi stack (module net80211)

The WiFi stack is a module that should be used by all WiFi card drivers. It implements all general WiFi oriented operations such as:

- parsing an incoming frame
- handling all IOCTLs (that are used to configure WiFi devices through user-space applications)
- scanning, AP authentication and authorization
- encryption
- and much more

The Wifi stack was taken from the NetBSD operating system and is now being redesigned by Sun Microsystems. However the driver API is supposed not to change therefore there will be no need for rewriting all WiFi drivers if some new features are added into WiFi stack (for example creating Ad-Hoc networks which is not implemented now).

More information about WiFi stack API can be learned from code description and code itself.

4. Device description

All information about the ZD1211 was obtained from existing drivers and their documentation. Therefore, some details may be inaccurate or incomplete.

4.1 Overview

ZD1211-based devices are composed of the ZD chip and a RF chip. The ZD controls the device and communicates with the host computer. It has supposedly two interfaces for that purpose, one for USB, other for PCI, but the latter has never been seen in an actual device. The ZD chip also comes in a "B" variant (ZD1211B), which is slightly different. It is a little-endian device; all multi-byte values sent through the USB pipes must adhere to this convention.

The RF chip handles the modulation/demodulation and transmission/reception itself. Different ZD-based devices come with different RF-chips. Note that the same RF-chip needs to be programmed slightly differently when coupled with the "B" variant of ZD1211.

4.2 I/O Registers

When using the USB interface, I/O read and write requests are made using endpoints 3 and 4 (see *Sending commands* below). This gives access to a 64k-word I/O space, that exposes the ZD1211 internals. The chip's ROM, EEPROM and some of its RAM are mapped into this I/O space. In particular, some addresses serve as registers that control the device, while others can be used to retrieve information about the chip, read/write the EEPROM and so forth.

Memory map

0x0000 .. 0x8FFF	unknown or unused
0x9000 .. 0x93FF	RAM – Physical registers (undocumented, 16-bit)
0x9400 .. 0x98FF	RAM – Control Registers (documented, 32-bit)
0x9900 .. 0xA0FF	4kB EEPROM memory
0xA100 .. 0xEBFF	unknown or unused
0xEC00 .. 0xF7FF	Firmware (older devices)
0xEE00 .. 0xF7FF	Firmware (newer devices)
0xF800 .. 0xFFFF	4kB ROM Firmware

Since most registers are 32-bit, they span two consecutive memory locations. Thus, to read/write a 32-bit register, two I/O operations on consecutive memory locations must be issued.

To make things more complicated, not all internal memories of the device use the same addressing scheme. Most parts use word addressing. This means that to access a register in two consecutive (16-bit) memory locations, one needs to access `REG_ADDR` and `REG_ADDR`+1. However, the RAM in 0x9000 .. 0x98FF uses byte addressing. Thus to access two consecutive words, one needs to read `REG_ADDR` and `REG_ADDR`+2.

4.3 USB endpoints

The device has four USB endpoints (plus the default control endpoint that every USB device has). These endpoints are listed in configuration 1, interface 0, alternate 0 in the USB descriptor tree (see USB client API).

0	control pipe	loading/reading firmware, device reset
1/OUT	bulk	sending packets
2/IN	bulk	receiving packets
3/IN	interrupt	reading registers, reporting status
4/OUT	interrupt or bulk	sending control commands

4.4 Control pipe

This pipe is already open when the driver is being attached. It is used for uploading and reading the device firmware and for the *reset* command, that kicks up the firmware once it has been uploaded.

Writing firmware

Request = 0x30 (write)

Value = starting address in firmware memory space (word-wise)

Index = 0

Length = number of bytes to write (maximum 4kB)

Reset and start new firmware

Request = 0x31 (read)

Value = 0

Index = 0

Length = 1

Request returns one byte whose highest bit is set in case of any errors.

Read firmware

Request = 0x32 (read)

Value = starting address in firmware memory space (word-wise)

Index = 0

Length = number of bytes to read (maximum 4kB)

Unlike "write firmware" only 60 bytes or less should be read using a single request. Longer requests must be split.

Uploading firmware

Before the firmware is uploaded only generic USB requests (such as `GET_DEVICE_DESCRIPTOR`) can be made.

The firmware should be uploaded during driver attach. It is too long to be loaded in one request, one cannot write more than 4 kB at a time. A typical firmware upload takes three control requests:

- Write firmware request (address=0xEE00, length=0x1000)
- Write firmware request (address=0xF600, length=0x400)
- Reset request (will also check for errors)

When the new firmware runs regular USB requests can be done.

4.5 Sending frames

Frames are sent through pipe 1/OUT. Each transfer should have the following format.

Offset	Len	Description
0x00	1 B	Rate, modulation type and flags
0x01	2 B	Frame size (including CRC and ICV fields)
0x03	1 B	Misc. flags
0x04	2 B	“Packet size” (not straightforward)
0x06	2 B	Duration of this frame in microseconds
0x08	1 B	Service number (Frame Length Extension Service or 0)
0x09	2 B	Length of the following frame in μs (if multiple frames are to be transmitted)
0x0B	n B	802.11 frame (including headers)

4.6 Receiving frames

Incoming frames are read from pipe 2/IN.

Offset	Len	Description
0x00	1 B	Received frame rate
0x01	4 B	?
0x05	n B	802.11 frame (already decrypted, but ICV and IV headers are still in)
n+0x5	4 B	CRC 32
n+0x9	1 B	Received signal strength indicator
n+0xA	1 B	Signal quality for CCK modulation
n+0xB	1 B	Signal quality for OFDM modulation
n+0xC	1 B	Cipher type (Which type of decryption has been applied)
n+0xD	1 B	Flags

The *flags* byte indicates either modulation type or error type, if an error occurred:

0x80 Frame error

0x40 CRC 32 error

0x20 Addresses do not match (packet could still be good in promiscuous mode)

0x10 CRC 16 error

0x08 Decryption not possible (wrong key)

0x04 Overrun

0x02 Time-out

0x01 OK – OFDM modulation

0x00 OK – DSSS modulation

0x69 Special case – merged transfer (see below)

If the frame was received and no error occurred, its modulation type can be determined. Depending on whether this was OFDM or DSSS, the *received frame rate* field should be interpreted as follows:

OFDM		**DSSS**	
0x0B	6 Mb/s	0x0A	1 Mb/s
0x0F	9 Mb/s	0x14	2Mb/s
0x0A	12 Mb/s	0x37	5 Mb/s
0x0E	18 Mb/s	0x6E	11 Mb/s
0x09	24 Mb/s		
0x0D	36 Mb/s		
0x08	48 Mb/s		
0x0C	54 Mb/s		

Merged transfers

The device can merge up to three 802.11 frames into a single USB transfer in order to lower USB overhead. If the last two bytes of the USB transfer are 0x7E 0x69 then the transfer has been merged and has the following structure: (Here, packet is a 802.11 frame including the ZD header and footer)

Packet 1 (+ alignment up to first multiple of 4 bytes)

Packet 2 (+ alignment up to first multiple of 4 bytes)

Packet 3 (+ alignment up to first multiple of 4 bytes)

Length of packet 1 (in bytes)

Length of packet 2 (in bytes), or 0 if packet 2 is not present

Length of packet 3 (in bytes), or 0 if packet 3 is not present

0x7E 0x69 (special signature)

If the last byte is not 0x69 then the USB transfer contains a single packet.

4.7 Sending commands

Commands are sent through pipe 4/OUT. This pipe can be either an interrupt or a bulk pipe (this varies greatly, even within the same device model). The driver must determine the type of endpoint 4/OUT and act accordingly.

Every transfer must have the following format:

Offset	**Len**	**Description**
0x00	1 B	Command code (lower byte)
0x01	1 B	0x00 (higher byte of command - reserved)
0x02	? B	Data (depending on command type)

Write register or memory

Command = 0x21

Data Address1(16 b), value1(16 b), address2(16 b), value2(16 b), ...

Read register or memory

Command = 0x22

Data Address1(16 b), address2(16 b), ...

Write RF registers

Command = 0x23

Data Type(16 b), count(16 b), value1(16 b), ... , value_count(16 b)

The value of *type* should be 1 for the 3683-A RF chip and 2 for other RF chips. *Count* represents how many values will come. It must equal to the width of the RF configuration register, which depends on RF chip type. (For the AL2230 it is 24.)

Each value (*value1* to *value_count*) carries only one bit of useful information. First value contains the most significant bit and the last value contains the least significant bit. The proper way to prepare these values is to the value of a special register at address 0x932C and use it as template. Bits 1 and 2 (where 0 is the least significant bit) must be cleared and bit 3 must be set to the desired value (that is actually sent into the RF register).

4.8 Receiving command results and HW status

Command results and HW status are received from pipe 3/IN. Every transfer has the following format:

Offset	Len	Description
0x00	1 B	Packet type (lower byte)
0x01	1 B	Packet type (higher byte)
0x02	? B	Data (depending on packet type)

Read from memory/register response

This packet will be generated in response to a “Read register or memory” command.

Packet type = 0x9001

Data address1(16b), value1(16b), address2(16b), value2(16b), ...

Retry Fail message

Following message is sent to host if the packet could not be transmitted.

Packet type	= 0xA001
Data	new_rate(16b), mac_addr(48b), number(16b), (and some other stuff that we do not understand)

HW interrupt messages

Packet type	= 0x9001 (the same as read response, but no read request was sent)
Data	address(16b), value(16b)

If an interrupt occurred, the device sends a surplus "read response" message. The address and value of the *interrupt control register* (0x9510) is sent in the data storage area. The cause of the interrupt is encoded in the IC register value:

bit 3	Wake up
bit 5	DTIM notify
bit 6	Configure next beacon

Other bits seem to be used only in the PCI device variant (not in the USB version) or not to be used at all.

5. Source code layout

The code is split into several files. We will discuss them in following paragraphs.

zyd_impl.h, zyd.c

This is the main source file. It contains all entry-point functions for MAC layer and WiFi stack module. It also contains general driver entry points (such as `_init`, `_fini`).

zyd_fw.h, zyd_fw.c

The device's firmware is stored in these files. We keep firmware simply in source code as a huge array of constant values. This approach is simple however change of firmware inevitably leads to recompilation of the whole driver.

zyd_hw.h, zyd_hw.c

Device specific (hardware) functions are kept here such as device initialization, channel setting and frame sending.

zyd.h

Public header file. (Nothing really interesting there).

zyd_reg.h

Declarations of ZD register addresses, field bitmasks and configuration sequences used by zyd_hw. We keep this in a separate file since there is quite a lot of them.

zyd_usb.h, zyd_usb.c

Everything related to communication over USB can be found here. This module takes care of finding USB endpoints, opening pipes to them and encapsulating all USB-transfers. It also installs callbacks to handle incoming USB transfers.

zyd_util.h, zyd_util.c

Contains supporting functions and structures. The most important thing here is a USB-safe locking primitive. Since it is forbidden to hold a mutex during calls to USBA we had to implement an extra lock on top of mutexes and conditional variables that is USB-safe.

6. TODO list

Following features are not implemented but they should be implemented in the future in order to secure complete functionality of the device.

- frames are always transmitted at 11Mbit/s in CCK modulation
- merged multi-frame transfers are dropped by receiving handler
- no encryption support
- no promiscuous mode support
- MAC address can not be changed
- no multicast support

7. References

[1] Sun Microsystems: writing Device Drivers, November 2006
(http://docs.sun.com/app/docs/doc/816-4854)