Student Research Project
Layer 2 network interface for Mono

Christian Otto

April 8, 2008

Abstract

The design and functionality of a layer 2 network interface that originated within the scope of the "Extended Hot Spot" project of Deutsche Telekom Laboratories is described an explained in this document. This interface is especially designed for the use in connection with the .NET framework Mono on router platforms. The implementation developed supports TI-AR7WRD compatible Routers running with the Texas Instruments Network Support Package (NSP), v.3.7.2, and the network driver shipped with the Texas Instruments Access Point Development Kit (APDK), v.7.5.29. For this project Castlenet, AVS800W+ boxes were used for testing.

A use case for that network interface could be a mesh network consisting of a great variety of router architectures that offers several services to users of the network. Under the conditions of the use case observed services are offered by the network, i.e. not by particular devices. Thus it must be guaranteed that the same software is installed on all routers. The conventional solution if this problem would be to port the software to each architecture that might appear in the mesh network. This would require a great effort without having the guarantee that the software behaves similarly on each platform. Therefore, a new abstraction layer was introduced. This provides a unique view on a network device for the programmer.

In the implementation described below the abstraction layer was realized by porting Mono to the MIPSel architecture. Due to the requirement of having direct access to 802.11 packets, it was necessary to create a layer 2 network interface to the AP driver. The description of that interface is of the main themes of this document. The possibility to implement network protocols hardware architecture independently in user space is now given by the existence of such an interface.

Contents

1 Connection between driver and interface

To provide general interoperability with the TI AP driver an interface to the hardware abstraction layer (HAL) of the driver was created. So one is able to handle network protocol problems in a hardware independent and abstract way without the need to switch into kernel space, e.g. when routing packets. This way, implementing solutions for routing problems is possible in C or C# without the need of touching the kernel. A first test was already carried out implementing an Ad-Hoc mode driver in C#.

1.1 Interface modules

The interface is composed of two small Kernel modules, a C library and a C# wrapper library. The kernel part of the software contains a module implementing a packet queue for received packets and a module providing a communication interface for the user space world. This interface is realized as a system of proc files (see Figure 3). A packet filter that is able to accept or drop packets specified by a bit mask was implemented for performance reasons.

1.2 Usage of the proc file interface

A packet can be sent by writing the raw packet plus some meta information (e.g. the wireless bssid, transmission power) to the network interface proc file '/proc/net/llnetif'. Meta information is created by setting appropriate annotations that prefix the actual packet. The packet format required by the proc file interface is shown in Figure 1.

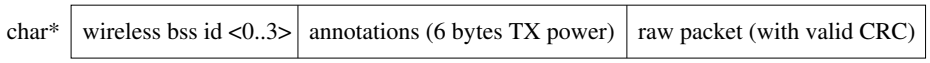

Figure 1: Packet format

The AVS800W+ has four registers that store virtual bssids. The first byte of the packet that written to /proc/net/llnetif specifies which of these bssid should be used for sending a packet. The transmission power annotation can be used for the regulation of the transmission power depending on the current data rate and the 802.11 standard used (a or b/g). If this annotation should be used, the next byte after the wireless bssid annotation should be set to

- WLAN_80211a (0x01): apply the following setting to packets that are sent out using the 802.11a standard
- WLAN_80211g (0x02): apply the following setting to packets that are sent out using the 802.11b or 802.11g standard

- WLAN_80211ag (0x03): apply the following setting to packets that are sent out not depending on the 802.11 standard that is used

Otherwise, this byte should be set to 0x00. In the case that the transmission power settings should be changed for a packet to be send out and WLAN_80211a, WLAN_80211g or WLAN_80211ag is set, the following five bytes adjust the transmission power for five predefined data rates where 0 is the lowest available transmission power an 31 the highest available one.

For performance reasons it might be useful to filter out certain packets and to only enqueue those packets that are accepted by the filter. This behaviour can be configured by writing a rule containing a policy, a bit mask and some meta information to the proc file '/proc/net/llnetfilter'. The format of that rule is shown in Figure 2.

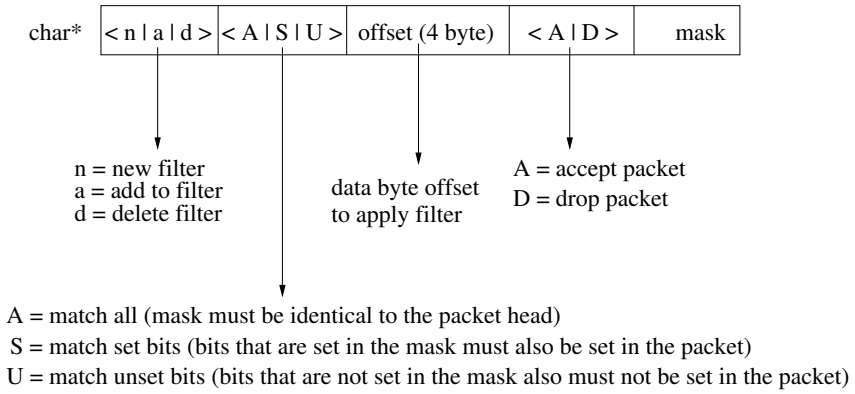

Figure 2: Filter rule

A packet is given to the user space by reading from the proc file. It contains the packet length and the raw packet itself. A poll routine for the network interface proc file was implemented to provide select requests on the proc file and thus to avoid busy wait situations. Whenever a new packet is read the file pointer of the proc file has to be reset to zero. So it is made sure that a packet is always read from its beginning. On the one hand this enables reading a packet incompletely if desired, and on the other hand it is impossible to loose borders between two packets if, for example, a packet is unintentionally read incompletely.

This functionality is comfortably accessable through the C and C# library described in section 3.

2 Kernel side internals

2.1 AP driver changes

Two small changes on the core driver were performed to facilitate sending and receiving packets via the proc file interface. Therefore the kernel symbol for the transmit function of the driver was exported to allow sending 802.11 packets from a third party module. Second a call to the enqueue function of the llnetif packet buffer was placed in the receive function of the core driver to get all the packets the driver receives. The enqueue function called as a part of an interrupt handler function has is given by the following signature:

int enqueue_packet(char data, unsigned int length);

where data is the recently received packet and length is the length of the packet. The function returns immediately when available filter rules were applied on the packet. If the packet is accepted it is put into the packet queue otherwise it is dropped.

Thus, the functionality of the driver nearly remains untouched while the new functionality works clearly separated from driver internals.

As the sources for the TI AP driver are not open source, the topic kernel and driver programming will not be treated in detail here. For further information on kernel programming for 2.4er kernels see ref. [1].

2.2 New functionality

Packets received and sent to the packet buffer are ready to be fetched by a user space process. The packet buffer has a fixed length. Thus, packets not fetched are dropped in favour of newer packets. If a user space process requests a packet by reading on the proc file '/proc/net/llnetif', information on the size of the oldest packet followed by the packet itself is given to the user space process. To read the next packet, first the user process has to reset the file pointer ('seek' the file pointer) to zero (see section 1.2).

A packet written to the proc file from user space is directly given to the transmit function of the AP driver. An own sending queue is unnecessary because the packet to be sent will be enqueued in the drivers transmission queue anyway. As one can see in Figure 3, the llnetif driver extension consists of three parts:

- the packet filter,
- the packet buffer,
- and the proc file interface itself.

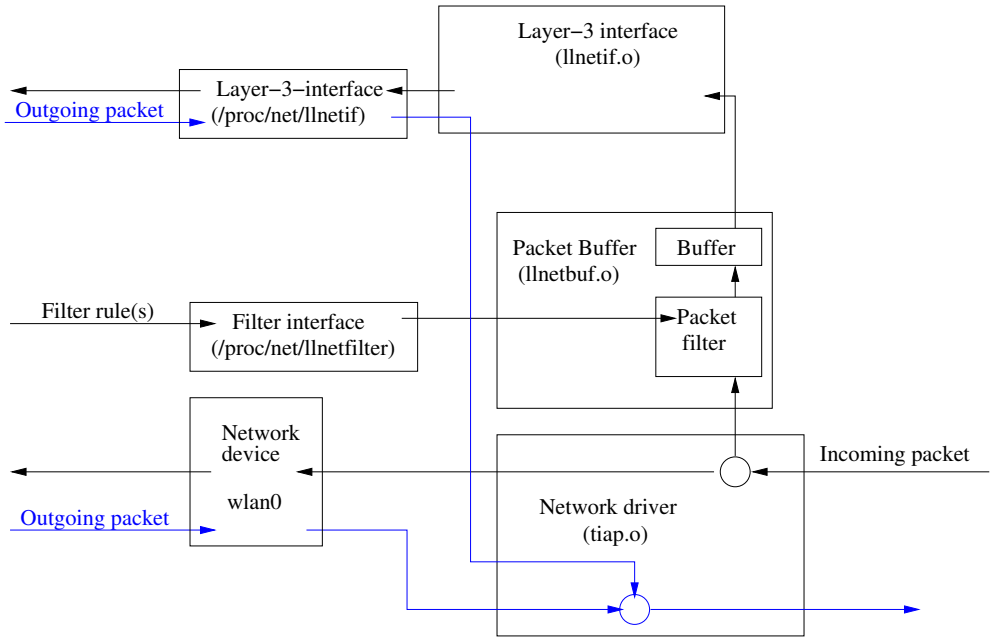

Figure 3: kernel space design

3 User space internals

The user space part of the interface (see figure 4) consists of a C library and a C# wrapper. The C library reads and writes to the proc files mentioned in section 1.2; the C# wrapper wraps the whole functionality of the C library.

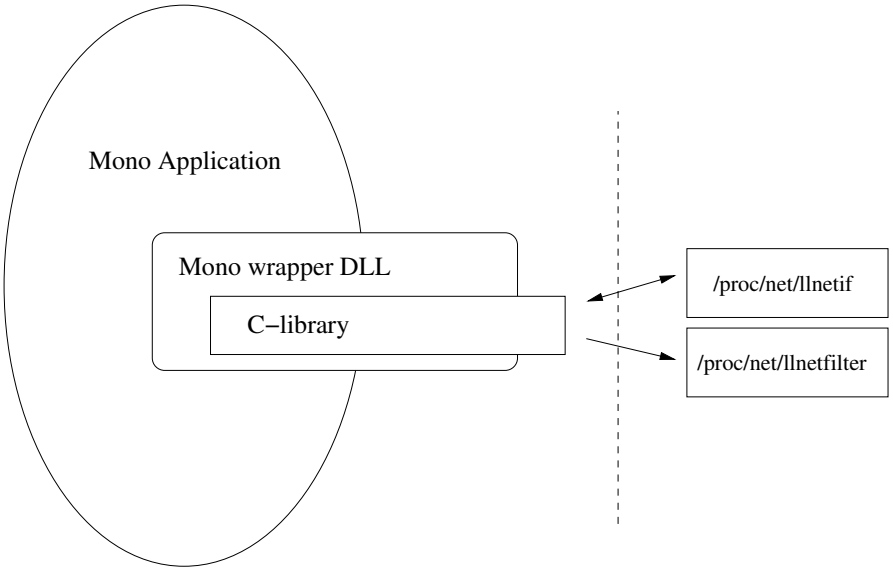

Figure 4: user space design

3.1 Functionality of the C library:

The C library provides an API for sending and receiving packets as well as setting certain values as filter rules or virtual BSS IDs. Once the library is initialized, a thread is started that performs a blocking select on the network interface proc file and calls a handler routine in case a packet is available. This handler routine has to be registered first by the user space application using this library. By this mechanism, polling is avoided for available packets. The C library has an own packet filter that works independently from the kernel space packet filter. This filter allows to register a constant number of different filter masks (128 seems to be a reasonable value for this constant). If a packet is sent to the handler routine, a 128 bit mask is given to the handler routine. It informs an application which of the 128 (possibly registered) filters match the packet and which do not. It is, for example, very easy to implement a packet distributor software in C# where programs can subscribe to a certain kind of packets specified by a particular set of filter masks.

The handler routine is registered by setting the appropriate function pointer of the type

const void(*f)(char*,char*).

This pointer can be a C function or a C# function if the function registration is performed via the C# wrapper library. The C# library contains an object that marshals the arguments for the handler function from the unmanaged to the managed .NET context. Thus, it is possible to have direct access to received wireless frames in very few lines of code and in a very comfortable way.

3.2 Functionality of the C# library

The library llnetif.dll mainly wraps the C library described in section 3.1 . The namespace llnetif mainly consists of two important classes:

- public class procfile, and
- public class distributor

3.2.1 public class procfile

The class procfile encapsulates very basic routines for directly accessing the proc file interface. This class contains the methods:

- **public static void init():**
  - The method procfile.init() opens the /proc/net/llnetif if it is not already open. In case the file is already open or an error occured the value -1 is returned. In case of no error the value 0 is returned.
- **public static int recv(ref byte[] buf):**
  - The method procfile.recv(...) reads a packet from /proc/net/llnetif. If an error occurred the value -1 is returned. If no packet is available the value 0 is returned else the reference **buf** contains the new packet and the return value contains the length of the packet.
- **public static int send(byte[] buf, byte wlan_bss_dev):**
  - The method procfile.send(...) writes buf to /proc/net/llnetif. In case this is a valid packet it will be sent out by the virtual bss device specified by the argument **wlan_bss_dev**. In case a write error occurs, the return value will be -1 otherwise the number of written bytes will be returned.

- **public static void add_filter_rule(byte mode, byte match_mode, byte policy, byte[] mask, int offset):**
  - The method procfile.add_filter_rule(...) adds a rule to the kernel filter writing the necessary filter information to /proc/net/llnet-filter.
  - Arguments:
    - mode:
      - llnetif.FILTER_DEFS.MODE.ADD_TO_MASK: append a new rule to an existing set of filter rules
      - llnetif.FILTER_DEFS.MODE.NEW_MASK: delete existing rules and create new set of rules where its first rule is the one specified by the next four arguments
      - llnetif.FILTER_DEFS.MODE.DELETE_MASK: delete existing rules
    - match_mode: the filter does a bitwise matching of filter masks and packets - if a packet matches a mask than a packet is either accepted or dropped (see policy)
      - llnetif.FILTER_DEFS.MATCH_MODE.MATCH_ALL: all specified bits in the mask have to be matched by the packet
      - llnetif.FILTER_DEFS.MATCH_MODE.MATCH_BIT_SET: only bits that are set to 1 have to be matched by the packet
      - llnetif.FILTER_DEFS.MATCH_MODE.MATCH_BIT_UNSET: only bits that are set to 0 have to be matched by the packet
    - policy:
      - llnetif.FILTER_DEFS.POLICY.ACCEPT_PACKET: accept a packet if the packet matches the mask
      - llnetif.FILTER_DEFS.POLICY.DROP_PACKET: drop a packet if the packet matches the mask
    - mask: bitmask that specifies class of packets that should be filtered
    - offset: packet offset that is added to a packets byte index when comparing packets with masks
- **public static void close()**
  - The method procfile.close() closes /proc/net/llnetif.

3.2.2 public class distributor

The class distributor contains the methods:

- **public static int init()**
  - The method distributor.init() opens /proc/net/llnetif and initializes some mutexes. In case of an error the value -1 otherwise the value 0 is returned.
- **public static void finish()**
  - The method distributor.finish() closes /proc/net/llnetif and finishes the packet distribution thread that was started via distributor.start_distribute().
- **public static void set_deliver_cb(CallBack cb)**
  - The method distributor.set_deliver_cb(...) sets the function pointer of the method that should be called when a packet arrives. The signature of the callback function must be
    void CallBack(byte[] packet, byte[] match_mask);
    where **packet** is the packet that has been received and **match_mask** is a 128 bit mask that specifies which of the 128 potentially registered filters (distributor.add_filter_rule(...)) match the packet.
- **public static int start_distribute()**
  - The method distributor.start_distribute() starts the packet distribution thread that calls the callback function that has been set by distributor.set_deliver_cb(...)every time a packet is read from /proc/net/llnetif.
- **public static void delete_filters()**
  - The method distributor.delete_filters() deletes all packet filters that have been registered.
- **public static void delete_filter(int filter_index)**
  - The method distributor.delete_filter(...) deletes the filter identified by index.
- **public static int add_filter_rule(int filter_index,byte mode, byte match_mode, byte policy,byte[] mask, int offset)**
  - The method distributor.add_filter_rule(...) adds a filter rule to the userspace packet filter. It works similarly to the kernel packet

filter (see section 1.2). The only difference is the argument **filter_index that** specifies one out of 128 packet filters that should be changed. The kernel filter has one filter only.

An example for a minimal program that would be able to receive and potentially send packets is given with Listing 1 below:

```
1  using System;
2  using System.Runtime.InteropServices;
3  using llnetif;
4
5  public class receive_packets {
6    public static void handle_packet(byte[] packet,
       byte[] match_mask) {
7      System.Console.WriteLine("got packet");
8    }
9
10   public static int Main(string[] args) {
11     byte[] filter_mask = new byte[1];
12     filter_mask[0] = 0;
13     if (distributor.init() < 0) {
14       Console.WriteLine("init failed!");
15       return 0;
16     }
17
18     distributor.set_deliver_cb(handle_packet);
19     distributor.add_filter_rule(0,
20         FILTER_DEFS.MODE.NEW_MASK,
21         FILTER_DEFS.MATCH_MODE.MATCH_BIT_SET,
22         FILTER_DEFS.POLICY.ACCEPT_PACKET,
23         filter_mask,0);
24     distributor.start_distribute();
25     System.Console.WriteLine("press enter to stop
         distribution");
26     System.Console.ReadLine();
27     distributor.finish();
28     return 0;
29   }
30 }
```

Listing 1: minimal llnetif program

3.3 802.11 over llnetif:

To make C# programming the the llnetif interface more comfortable the class p80211 provides an API for building 802.11 packets. This API provides both set and get methods for all fields of an 802.11 packet and functions to convert a p80211 class into a byte array and back. Even though there exist set methods for the packet time stamp and the CRC32 checksum at the end of the packet these fields are ignored by the driver and overwritten by the firmware sending out a packet (see p80211.cs).

The conversion methods byte-array-to-packet80211 and packet80211-to-byte-array will be described more extensively now:

public byte[] ToByteArrayNetwork()

- The method ToByteArrayNetwork() converts a packet80211 class into a byte array. All address fields and the CRC32 checksum are converted to network byte order.

public byte[] ToByteArrayHost()

- The method ToByteArrayHost() converts a packet80211 class into a byte array. The byte order of all address fields and the CRC32 checksum stays untouched.

public int ReadFromByteArrayNetwork(byte[] bpacket)

- The method ReadFromByteArrayNetwork(...) converts a byte array into a packet80211 class. All address fields and the CRC32 checksum are converted to host byte order.

public int ReadFromByteArrayHost(byte[] bpacket)

- The method ReadFromByteArrayHost(...) converts a byte array into a packet80211 class. The byte order of all address fields and the CRC32 checksum stays untouched.

3.4 .Net Interoperability Details

As C# has been designed as a very high level language some of the functionality needed to execute functions that normally run in kernel space is missing. That means that this functionality has to be added by writing new C# classes. Those classes often wrap C libraries that implement the functionality needed. Therefore a mechanism is required that connects the unmanaged (native) context with the managed context. That mechanism is called marshaling.

3.4.1 Marshaling arguments in C#

Arguments that are sent from the unmanaged to the managed context have to be marshaled. In C# attributes are used to append meta information to source code. This information is put into square brackets. There is a great variety of possibilities to marshal arguments (see [2]).

Custom marshaling: The llnetif.dll library uses custom marshalling to copy function arguments or results between managed and unmanaged context. Therefore one has to extend ICustomMarshaler Object and implement the functions:

public static ICustomMarshaler GetInstance (string s)
public void CleanUpManagedData(object o)
public void CleanUpNativeData(IntPtr pNativeData)
public int GetNativeDataSize (IntPtr ptr)
public int GetNativeDataSize ()
public IntPtr MarshalManagedToNative (object obj)
public object MarshalNativeToManaged (IntPtr pNativeData)

The functions that are important for the actual marshaling are MarshalManagedToNative and MarshalNativeToManaged. At the example beyond byte arrays are marshaled between native and managed context. In native context the array length is always saved in the first four bytes of the array. Without that information it would be impossible to marshal from unmanaged to managed context because type safeness could not be guaranteed (the other direction of course would work).

```
1  public class ByteArrayMarshaler : ICustomMarshaler {
2    private static ByteArrayMarshaler Instance = new
        ByteArrayMarshaler();
3    public static ICustomMarshaler GetInstance (string
        s) {
4      return Instance;
5    }
6    public void CleanUpManagedData(object o){ }
7    public void CleanUpNativeData(IntPtr pNativeData){
8      UnixMarshal.FreeHeap(pNativeData);
9    }
10   public int GetNativeDataSize (IntPtr ptr) {
11     return Marshal.ReadInt32(ptr);
12   }
13   public int GetNativeDataSize () {
```

```
14      return Marshal.SizeOf(typeof(byte));
15    }
16
17    public IntPtr MarshalManagedToNative (object obj) {
18      if (obj == null) return IntPtr.Zero;
19      if (obj.GetType() != typeof(byte[]))
20        throw new ArgumentException("Argument must be
            byte[]");
21      byte[] array = obj as byte[];
22      IntPtr ptr = Marshal.AllocHGlobal(
23        array.Length + Marshal.SizeOf(typeof(int)));
24      Marshal.WriteInt32(ptr,array.Length);
25      ptr = (IntPtr)((int)ptr + Marshal.SizeOf(typeof(
           int)));
26      Marshal.Copy(array,0,ptr,array.Length);
27      return ptr;
28    }
29
30    public object MarshalNativeToManaged (IntPtr
         pNativeData) {
31      if(pNativeData == IntPtr.Zero) return null;
32      int size = GetNativeDataSize(pNativeData);
33      byte[] array = new byte[size];
34      IntPtr data_ptr = (IntPtr)((int)pNativeData +
           Marshal.SizeOf(typeof(int)));
35      Marshal.Copy(data_ptr,array,0,size);
36      return array;
37    }
38  }
```

Listing 2: ByteArrayMarshaler

3.4.2 Using a custom marshaler

To use a certain marshaler the appropriate attribute for each parameter of a function has to be set. The following example shows how to marshal a byte array that is returned by a C library function with the ByteArrayMarshaler:

```
[DllImport("libllaccess.so")]
[return : MarshalAs(UnmanagedType.CustomMarshaler,
    MarshalTypeRef=typeof(ByteArrayMarshaler))]
public static extern byte[] some_function();
```

If arguments of a function have to be marshaled, i.e. if a Mono function is called from a C library (callback) one would have to set a marshaler for each argument:

```
private static void _cb([MarshalAs(UnmanagedType.
    CustomMarshaler,
  MarshalTypeRef=typeof(ByteArrayMarshaler))] byte[]
     packet,
  [MarshalAs(UnmanagedType.CustomMarshaler,
  MarshalTypeRef=typeof(ByteArrayMarshaler))] byte[]
     match_mask) {
  //...
}
```

4 Conclusion

Originally the intention of that project was to create an interface not only to directly send and receive 802.11 frames via a proc file interface but to to send arbitrary packets.

The first part of these plans has been put into practice. It is now possible to create and send valid packets in userspace and to send them out using the C# library, the C library and the proc file interface that are described above. As a proof of concept the annotation for the transmission power regulation that is also described in section 1.2 was added. In case there will be a requirement for more annotations they can be added easily in a similar way. When introducing new annotations one has to bear in mind that the TI AP driver mostly needs to be restarted when changing driver settings e.g. the maximum data rate. Thus, at least when using the TI AP driver, changing these parameters within a session would be inefficient because the router would become unusable.

One of the intended goals was to facilitate the possibility to send completely arbitrary packets. This could not be reached. The most important reason was that the firmware sources of the TI AP driver were inaccessible. Important validity checks when sending packets are executed at this level. Thus it was not possible do deactivate those checks to make sending of arbitrary packets possible.

References

[1] Jonathan Corbet Alessandro Rubini. *Linux Device Drivers.* O'Reilly, 2nd edition, jun 2001.

[2] Hisham Mardam Bey. Mono documentation - interop with native libraries. `http://www.mono-project.com/Interop_with_Native_Libraries`.