Packet Wire in Micro-C

This lab walk-through is a simple packet processing application that swaps vlan tags and counts various packets, using a single network port.

It is designed to operate in conjunction with a driving NIC that can send packets on VLAN 2, and receive packets back on VLAN 3.

The application can use any number of MEs, and the lab involves compiling the code, and changing various ways in which the packets are counted. It is a more sophisticated application than lab4_microc, but it is not really a fully fledged network application.

Steps

1. Check out the repository containing these labs and the source code.
2. Open a terminal window, and enter the directory of the repository.
3. Enter the 'apps' directory.

```
> cd apps
```

4. Enter the lab5 directory, and look at the contents. There is a default Makefile in the directory, plus some C files. The Makefile is ready for use.

```
> cd lab5
> ls
blm_custom.h  config.h  init  mac_stats.c  Makefile  README.md  wire_main.c
```

5. Look at the Makefile; in this lab the Makefile is pre-written and requires no editing. Find the section of the Makefile with the comment 'Application definition'. The lines should look something like the following:

```
#
# Application definition
```

```
#
$(eval $(call micro_c.compile_with_rtl,wire_obj,wire_main.c))
$(eval $(call micro_c.add_fw_libs,wire_obj,nfp pkt std net))
$(eval $(call micro_c.force_include,wire_obj,config))

$(eval $(call fw.add_obj,wire,wire_obj,i32.me0 i32.me1 i32.me2 i32.me3 i32.me4
$(eval $(call fw.add_obj,wire,wire_obj,i33.me0 i33.me1 i33.me2 i33.me3 i33.me4
$(eval $(call fw.add_obj,wire,stats_obj,i34.me0))
$(eval $(call fw.add_obj,wire,blm_obj,i48.me0))
$(eval $(call fw.link_with_rtsyms,wire))
$(eval $(call fw.add_ppc,wire,i8,$(PICO_CODE)))
```

There are two sections to this part of the Makefile: the first 3 lines relate to creating a micro_c program called `wire_obj`. The last six lines relate to creating a firmware called `wire`.

The first line specifies the main file compilation for the program `wire_obj`, using the source code `wire_main.c`.

The second line adds some 'C' source libraries; these supply various C functions that are required for the build. Note that these are compiled with the main program, not just linked.

The third line adds a special argument to the invocation of the C compiler; it adds an argument to *force* the inclusion of the file `config.h` at the start of preprocessing. This feature is used to provide multiple build configurations given the same source code, by including a different configuration file for different builds. In this lab a `#include "config.h"` could have been used in the `wire_main.c` file, but it is good practice in applications to permit this generic configuration.

The second section starts at the fourth line of text. This starts by adding the `wire_obj` to twelve microengines in island 32 and to twelve microengines in island 33 for firmware `wire`.

The next line adds `stats_obj` to a microengine in i34; this is a useful program to gather statistics from the MACs and accumulate them in cluster local scratch memory (and its definition in terms of C source files etc is above in the "# Gather MAC statistics" area of the Makefile).

Another line adds the `BLM` component in a microengine in island i48; this is a buffer recycler that enables the system to operate, and is built from assembler in the '# BLM' section above.

Another line forces the firmware to include run-time symbols, and then the last line indicates that the firmware `wire` should include the standard PPC code in the network interfaces - this code provides a simple packet characterization for packets that are delivered to microengines.

Effectively these lines of code define what programs should be built and included on what microengines for the firmware - they are a kind of 'project definition'.

6. Look at the main function in wire_main.c:

```
int
main(void)
{
    struct pkt_rxed pkt_rxed; /* The packet header received by the thread */
    __mem struct pkt_hdr *pkt_hdr;    /* The packet in the CTM */

    /*
     * Endless loop
     *
     * 1. Get a packet from the wire (NBI)
     * 2. Rewrite the packet ready for retransmission
     * 3. Count the packet as required
     * 4. Do statistics on the packet
     * 5. Send the packet back to the wire (NBI)
     */
    for (;;) {
        /* Receive a packet */
        pkt_hdr = receive_packet(&pkt_rxed, sizeof(pkt_rxed));

        /* Rewrite the packet */
        //rewrite_packet(&pkt_rxed, pkt_hdr);

        /* Count the packet */
        //count_packet(&pkt_rxed, pkt_hdr);

        /* Do stats on the packet */
        //stats_packet(&pkt_rxed, pkt_hdr);

        /* Send the packet */
```

```
            send_packet(&pkt_rxed.nbi_meta, pkt_hdr);
        }

        return 0;
    }
```

This is the main program loop for the `wire_obj` program. Its basic outline is to receive a packet, convert the VLAN tag (2 to 3 or vice versa), count the packet, do some statistics, then transmit it.

Note that the intervening functions are commented out; they will be added one at a time.

7. The firmware is ready to be compiled, using make:

```
> make
```

8. It is worth looking at the output link map file, which shows the placement of the symbols used in the program.

```
> cat wire.map
Memory Map file: /root/gavin/Bloodhound/apps/lab5/wire.map
Date: Tue Nov 17 15:30:21 2015

nfld version: 5.2.0.0,  NFFW: /root/gavin/Bloodhound/apps/lab5/wire.fw

Address       Region     ByteSize        Symbol
=================================================
0x0000000000802000    i24.emem      108             .mip
0x0000000000000000    i32.cls       32              i32._stats
0x0000000000280000    i28.imem      32              _counters
0x0000000000000000    i33.cls       32              i33._stats
0x0000000000000000    i48.me0.lm    32              i48.me0.BLQ0_DESC_LMEM_BASE
....
```

There are a lot more lines after this relating to the 'BLM' component, but there are three important symbols here. The first is `_counters`, which is in the internal memory in island 28. This is used to accumulate packet counts using atomic increment transactions - but we shall see that later.

There are then a pair of symbols `i32._stats` and `i33._stats`. These are in the cluster local scratch (CLS) for the islands, and they are used for packet and byte count statistics - again, we shall see this later.

9. It is now time to load and start the firmware that has been created onto the NFP.

```
> ./init/wire.sh start wire.fw
Starting FW:
 - Reset Islands...done
 - Loading FW (no-start)...done
 - Init DMA...done
 - Init TM...done
 - Starting FW...done
 - Init MAC for SF...done
```

This uses a shell script to restart the firmware by clearing some state in the NIC, loading the firmware, setting up network DMA, traffic management etc, and then starting the firmware and enabling the Ethernet interfaces.

The content of the shell script is not important to know - but it is worth knowing that running firmware can demand some other operations too.

10. The firmware at present is just a packet mirror - it transmits the packets it receives unchanged. So another terminal window will be required to monitor the packets on the network interface. So open a second terminal window to monitor packets.

11. In the monitoring terminal run `tshark`:

```
> tshark -x -i p4p1
tshark: Lua: Error during loading:
[string "/usr/share/wireshark/init.lua"]:46: dofile has been disabled due to ru
Running as user "root" and group "root". This could be dangerous.
Capturing on 'p4p1'
```

Ignore the error about Lua; it is dangerous to run tshark as root, as it is dangerous to run many programs. However, it is okay for this lab on a test machine.

12. Now it is time to transmit a packet from the connected NIC.

```
> tcpreplay -i p4p1 ~/udp_1pkt.pcap
sending out p4p1
processing file: /root/udp_1pkt.pcap
Actual: 1 packets (58 bytes) sent in 0.01 seconds.      Rated: 5800.0 bps, 0.0
Statistics for network device: p4p1
Attempted packets:         1
Successful packets:        1
Failed packets:            0
Retried packets (ENOBUFS): 0
Retried packets (EAGAIN):  0
```

In the monitoring window there should be *two* packets.

```
0000  22 33 44 55 66 77 11 22 33 44 55 66 08 00 45 00   "3DUfw."3DUf..E.
0010  00 2c 00 01 00 00 40 11 66 5c 0a 00 00 01 0a 00   .,....@.f\......
0020  00 64 0b b8 0f a0 00 18 97 c1 00 01 02 03 04 05   .d..............
0030  06 07 08 09 0a 0b 0c 0d 0e 0f                     ..........

1 0000  22 33 44 55 66 77 11 22 33 44 55 66 08 00 45 00   "3DUfw."3DUf..E.
0010  00 2c 00 01 00 00 40 11 66 5c 0a 00 00 01 0a 00   .,....@.f\......
0020  00 64 0b b8 0f a0 00 18 97 c1 00 01 02 03 04 05   .d..............
0030  06 07 08 09 0a 0b 0c 0d 0e 0f 00 00               ............

2
```

There are two packets as one is the outgoing packet, one is the returning packet. Note that they are identical.

At this point the `tshark` can be stopped.

13. Look again at the code in wire_main.c; in particular, look now at receive_packet, and the following lines:

```
__xread struct pkt_rxed pkt_rxed_in;
...
pkt_nbi_recv(&pkt_rxed_in, sizeof(pkt_rxed->nbi_meta));
```

This code consists of a first `pkt_nbi_recv` function call; this invokes a transactional memory operation with the packet engine in the local Cluster Target Memory, to

add the thread to the pool of worker threads waiting for packets. This function will not return until a packet is delivered to the thread - the contents of the packet are delivered in to special `xread` registers in the `pkt_rxed_in` structure.

The code then copies this out into non-special registers:

```
pkt_rxed->nbi_meta = pkt_rxed_in.nbi_meta;
```

This is because there are (very) limited numbers of `xread` registers.

A short time later the code has:

```
mem_read32(&(pkt_rxed_in.pkt_hdr), pkt_hdr, sizeof(pkt_rxed_in.pkt_hdr));
pkt_rxed->pkt_hdr = pkt_rxed_in.pkt_hdr;
```

This is a different memory transaction - but it is an explicit memory transaction (as opposed to the implicit transactions that occurred in lab4). This reads a number of 32-bit words in to `xread` registers holding pkt_rxed_in.pkt_hdr; the mem_read32 will wait until this completes, too. The second line copies the data again to standard registers.

So on completion the `receive_packet` function will have waited for a packet to reach the thread, and the packet header will be in structures in hand ready to be used.

Note that the C compiler can keep data structures in registers or various memories; however, the standard build options in the Makefiles force them to be held in general purpose registers.

14. The function `send_packet` is of less interest, as in this lab it only transmits the packet without any changes - except for generating the correct L3 and L4 checksums using the hardware. So we will not go into any analysis of it.

15. The next step in the lab is to add and look at the counting of packets. This shows the use of C code to analyze the packets, and to explicitly invoke transacations to update counters. Before starting this, since there is no code updating the counters currently, it is worth checking that they are zero:

```
> nfp-rtsym _counters
0x0000000000:  0x00000000 0x00000000 0x00000000 0x00000000
*
```

16. Before moving on, stop the firmware:

```
> ./init/wire.sh stop
Stopping FW:
 - Bringing down interfaces...done
 - Remove net_dev and load nfp driver...done
 - Unload FW...done
```

17. Now uncomment the packet counting (the call of `count_packet` in the main loop in `wire_main.c`), and rebuild

18. Look at the source for count_packet:

```
if (pkt_rxed->pkt_hdr.pkt.tpid!=0x8100) {
    mem_incr64(&counters.no_vlan);
} else {
    if ((pkt_rxed->pkt_hdr.pkt.tci & 0xfff)==2) {
        mem_incr64(&counters.vlan_2);
    } else if ((pkt_rxed->pkt_hdr.pkt.tci & 0xfff)==3) {
        mem_incr64(&counters.vlan_3);
    } else {
        mem_incr64(&counters.vlan_other);
    }
}
```

This code does nothing particularly surprising; it examines the Ethernet protocol type to see if a VLAN is present, and if not it increments a counter `counters.no_vlan`. If a VLAN is present then one of three other counters is incremented. This will provide four counters, then, for different types of packets.

What is of interest here is the mem_incr64 function calls. These are actually in one of the microc libraries under `microc/lib`, and is included with `nfp/mem_atomic.h`. The increment is a transaction to the transactional memory that owns the counters, and it invokes a command to that memory with an embedded argument (an *immediate atomic*). Since this transaction does not require additional argument data

the thread does not need to deschedule while the transaction takes place. This cannot be seen in the code, but it is useful to understand that some library calls require waiting for completion (such as the calls in receive_packet) whereas others are "fire and forget".

19. Now the firmware is ready to load (since it was rebuilt after editing).

```
> ./init/wire.sh start wire.fw
Starting FW:
 - Reset Islands...done
 - Loading FW (no-start)...done
 - Init DMA...done
 - Init TM...done
 - Starting FW...done
 - Init MAC for SF...done
```

20. Now when packets flow through the counters should update.

```
> tcpreplay -i p4p1 ~/udp_1pkt.pcap
    ...
> nfp-rtsym _counters
0x0000000000:  0x00000007 0x00000000 0x00000000 0x00000000
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
```

Note that the numbers here may not be the same; this is because some Ethernet interfaces in Linux may generate spurious packets on their own, such as IPv6 auto-discovery - and these will be counted too!

Also note that these counters are 64-bit counters, as they are specified in the code as `uint64_t` and they are incremented with mem_incr64. The data is stored in memory in LWBE format - a somewhat bizarre cross between little- and big-endian formats. LWBE stands for little-word-big-endian - the first word is then the lowest 32 bits, and the second word is the upper 32 bits, for each 64-bit pair. This is why the first word increments for non-vlan packets.

21. Try sending a packet with a VLAN tag of 2

```
> tcpreplay -i p4p1 ~/udp_v2.pcap
    ...
```

```
> nfp-rtsym _counters
0x0000000000:  0x0000001a 0x00000000 0x00000001 0x00000000
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
```

Note again that the first number may be different.

The second counter is the 'VLAN 2' counter - there has now been one VLAN 2 packet seen. The counters are working.

22. Before moving on, stop the firmware:

```
> ./init/wire.sh stop
Stopping FW:
 - Bringing down interfaces...done
 - Remove net_dev and load nfp driver...done
 - Unload FW...done
```

23. Now look at the code for rewriting the packet, in `rewrite_packet`.

```
int vlan;

vlan = 0;
if (pkt_rxed->pkt_hdr.pkt.tpid==0x8100) {
    vlan = pkt_rxed->pkt_hdr.pkt.tci & 0xfff;
    if ((vlan==2) || (vlan==3)) {
        pkt_hdr->pkt.tci = pkt_rxed->pkt_hdr.pkt.tci ^ 1;
    }
}
```

This code is again relatively simple; it checks for a VLAN tag, and if the VLAN is 2 or 3 then it toggles the bottom bit of the tag in the packet data itself.

To change the actual packet the code uses an *implicit* memory transaction. The structure `pkt_hdr` is the packet header in the cluster target memory - it is a `__mem struct pkt_hdr *`. This address was generated in `receive_packet`, and it has provided for the ability to update the packet for transmission (indeed, the `send_packet` function needs to modify metadata ahead of the packet itself for transmit, and that function is already using this address).

What this code shows is that a memory transaction can be hidden if desired, for

ease of coding or for ease of reading. The line `pkt_hdr->pkt.tci = pkt_rxed->pkt_hdr.pkt.tci ^ 1;` is just standard C for updating a data structure. The C compiler itself will generate a transcational memory write, and the thread will deschedule while this transaction completes - but it is invisible to the code reader.

24. Uncomment the call of `rewrite_packet` , rebuild and start the firmware:

```
> make
...
> ./init/wire.sh start wire.fw
...
```

25. Back in the monitoring terminal restart `tshark`

```
> tshark -x -i p4p1
...
```

26. Transmit a packet that is not on VLAN 2:

```
> tcpreplay -i p4p1 ~/udp_1pkt.pcap
...
```

The monitoring window should have something like:

```
0000  22 33 44 55 66 77 11 22 33 44 55 66 08 00 45 00   "3DUfw."3DUf..E.
0010  00 2c 00 01 00 00 40 11 66 5c 0a 00 00 01 0a 00   .,....@.f\......
0020  00 64 0b b8 0f a0 00 18 97 c1 00 01 02 03 04 05   .d..............
0030  06 07 08 09 0a 0b 0c 0d 0e 0f                     ..........

0000  22 33 44 55 66 77 11 22 33 44 55 66 08 00 45 00   "3DUfw."3DUf..E.
0010  00 2c 00 01 00 00 40 11 66 5c 0a 00 00 01 0a 00   .,....@.f\......
0020  00 64 0b b8 0f a0 00 18 97 c1 00 01 02 03 04 05   .d..............
0030  06 07 08 09 0a 0b 0c 0d 0e 0f 00 00               ............
```

The received packet is unchanged compared to the transmitted packet.

27. Transmit a packet that is on VLAN 2:

```
> tcpreplay -i p4p1 ~/udp_1pkt.pcap
...
```

The monitoring window should now have something more, looking like:

```
0000  22 33 44 55 66 77 11 22 33 44 55 66 81 00 90 02   "3DUfw."3DUf....
0010  08 00 45 00 00 2c 00 01 00 00 40 11 66 5c 0a 00   ..E..,....@.f\..
0020  00 01 0a 00 00 64 0b b8 0f a0 00 18 97 c1 00 01   .....d..........
0030  02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f         ..............

0000  22 33 44 55 66 77 11 22 33 44 55 66 81 00 80 03   "3DUfw."3DUf....
0010  08 00 45 00 00 2c 00 01 00 00 40 11 66 5c 0a 00   ..E..,....@.f\..
0020  00 01 0a 00 00 64 0b b8 0f a0 00 18 97 c1 00 01   .....d..........
0030  02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00 00   ................
```

Here it can be seen that the VLAN tag has been changed from 2 to three - this is the last byte on the first line.

At this point the `tshark` can be stopped.

28. Before moving on, stop the firmware:

```
> ./init/wire.sh stop
Stopping FW:
 - Bringing down interfaces...done
 - Remove net_dev and load nfp driver...done
 - Unload FW...done
```

29. In the NFP architecture there are usually a number of ways to reach a desired goal, and what decides the "best" option may be dependent on system performance goals, bus bandwidths, memory bandwidths, and so on. In general, though, below e.g. 40Gbps, all options are equally good.

    Another option for maintaining statistics is to use the cluster local scratch - and this is what `stats_packet` does.

```
__intrinsic void
stats_packet( struct pkt_rxed *pkt_rxed,
              __mem struct pkt_hdr *pkt_hdr )
```

```
{
    __xwrite uint32_t bytes_to_add;
    SIGNAL   sig;
    int address;

    bytes_to_add = pkt_rxed->nbi_meta.pkt_info.len;

    if (pkt_rxed->pkt_hdr.pkt.tpid!=0x8100) {
        address = (uint32_t) &(stats.no_vlan);
    } else {
        if ((pkt_rxed->pkt_hdr.pkt.tci & 0xfff)==2) {
            address = (uint32_t) &(stats.vlan_2);
        } else if ((pkt_rxed->pkt_hdr.pkt.tci & 0xfff)==3) {
            address = (uint32_t) &(stats.vlan_3);
        } else {
            address = (uint32_t) &(stats.vlan_other);
        }
    }

    __asm {
        cls[statistic, bytes_to_add, address, 0, 1], ctx_swap[sig]
    }
}
```

This function is declared in a novel fashion - it is declared as an `__intrinsic` function. These functions are special in that they are always inlined, and they are inlined with the view to optimizations at compile time such as constant elimination and code elimination; they may also contain in-line assembler. Many of the library functions are written as `__intrinsic` .

This function is only `__intrinsic` so that it may contain the in-line assmbler at the end, and so that it may handle the transaction completion `SIGNAL sig` .

The function is very similar to `count_packet` - it determines which statistic needs to be updated depending on the incoming VLAN tag. In this case two values are generated - `bytes_to_add` (which is the length of the packet in bytes) and `address` (the address of the statistic to update).

With these two values in hand a transaction to the cluster local scratch (CLS) is invoked. Normally these transactions are hidden within libraries, but this function "takes the lid off" so we can see the explicit transaction. These transactions have to

be invoked with assembler instructions - in this case it is a `cls[]` command instruction. The particular transaction it is invoking is the `statistic` transaction. This is a special atomic-add-and-increment operation which the CLS can perform - it reads 64-bits of memory, adds a byte value to the bottom 35 bits and it increments the top 29 bits, effectively treating the memory location as a byte and packet count with a 35-bit byte counter and a 29-bit packet counter.

To perform the transaction the CLS has to know which address to use for the transaction - this is provided as `address` - and it needs to know the number of bytes. The number of bytes is presented in the "pull" registers, which are the special `__xwrite` registers. Hence `bytes_to_add` is declared as `__xwrite`.

The last special part of the assembler instruction is `ctx_swap[sig]`. This tells the thread that it can swap out, until the transaction completes - and the transaction will use the signal `SIGNAL sig`.

30. Edit `wire_main.c` to uncomment the call of `stats_packet`, make the firmware, and start it.

```
> make
...
> ./init/wire.sh start wire.fw
...
```

31. Now when packets flow through the statistics in the CLS should update.

```
> tcpreplay -i p4p1 ~/udp_1pkt.pcap
sending out p4p1
processing file: /root/udp_1pkt.pcap
Actual: 1 packets (58 bytes) sent in 0.01 seconds.      Rated: 5800.0 bps, 0.04
Statistics for network device: p4p1
Attempted packets:         1
Successful packets:        1
Failed packets:            0
Retried packets (ENOBUFS): 0
Retried packets (EAGAIN):  0

> nfp-rtsym i32._stats
0x0000000000:  0x00000155 0x00000020 0x00000000 0x00000000
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
```

```
> nfp-rtsym i33._stats
0x0000000000:  0x00000111 0x00000018 0x00000000 0x00000000
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
```

The statistics are again LWBE, to make 64-bit quantities. In the capture above the first statistic is 0x0000002000000155; splitting this out gives a 35-bit byte count (0x155) and packet count (4), for example.

```
> tcpreplay -i p4p1 ~/udp_v2.pcap
sending out p4p1
processing file: /root/udp_v2.pcap
Actual: 1 packets (62 bytes) sent in 0.01 seconds.      Rated: 6200.0 bps, 0.0
Statistics for network device: p4p1
Attempted packets:         1
Successful packets:        1
Failed packets:            0
Retried packets (ENOBUFS): 0
Retried packets (EAGAIN):  0

> nfp-rtsym i32._stats
0x0000000000:  0x00000155 0x00000020 0x00000000 0x00000000
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
> nfp-rtsym i33._stats
0x0000000000:  0x00000111 0x00000018 0x00000046 0x00000008
0x0000000010:  0x00000000 0x00000000 0x00000000 0x00000000
```

Here we can see the VLAN 2 packet went through island 33 and yielded 0x0000000800000046 - or a byte count of 70 and packet count of 1.

32. Finally stop the firmware:

```
> ./init/wire.sh stop
Stopping FW:
 - Bringing down interfaces...done
 - Remove net_dev and load nfp driver...done
 - Unload FW...done
```